Software Project Management Laboratory

1. Git Introduction

Andrea Morichetta, Phd

Computer Science Division

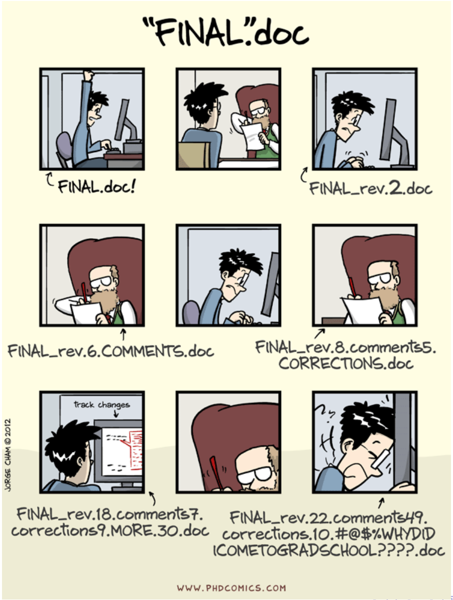
`http://swcarpentry.github.io/git-novice/`

Pro Git Book

October 3rd, 2018

IN CASE OF FIRE
git commit
git push
exit building

Why Git is useful?



Use Case

Consider this scenario:

- You have a homework submission for today and the assignment is ready for submission
- While testing it you discovered a minor bug and decided to fix it
- After attempting to do so, you accidentally changed a working code and got yourself in a big mess
- You no longer remember what was and what wasn't there
- It is 23:58 PM

Use Case

Consider this scenario:

- You have a homework submission for today and the assignment is ready for submission
- While testing it you discovered a minor bug and decided to fix it
- After attempting to do so, you accidentally changed a working code and got yourself in a big mess
- You no longer remember what was and what wasn't there
- It is 23:58 PM

Then you realize that Ctrl + Z won't solve your problem

What is Git?

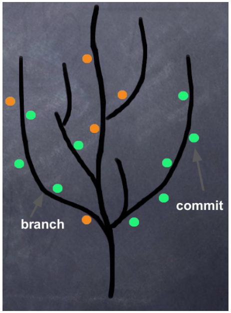
- **Open source** project originally developed in 2005 by Linus Torvalds
- A **command line** utility
- You can imagine git as something that sits on top of your **file system** and **manipulates files**.
- A **distributed version control** system - DCVS

What is distributed version control system?

- **Version control system** is a system that records changes to a file or set of files over time so that you can recall specific versions later
- **Distributed** means that there is no main server and all of the full history of the project is available once you cloned the project.

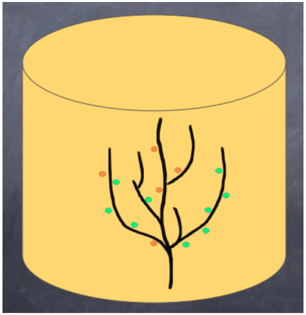
Git

- You can imagine git as something that sits on top of your file system and manipulates files.
- This "something" is a **tree structure** where each commit creates a new node in that tree.
- Nearly all git commands actually serve to navigate on this tree and to manipulate it accordingly.



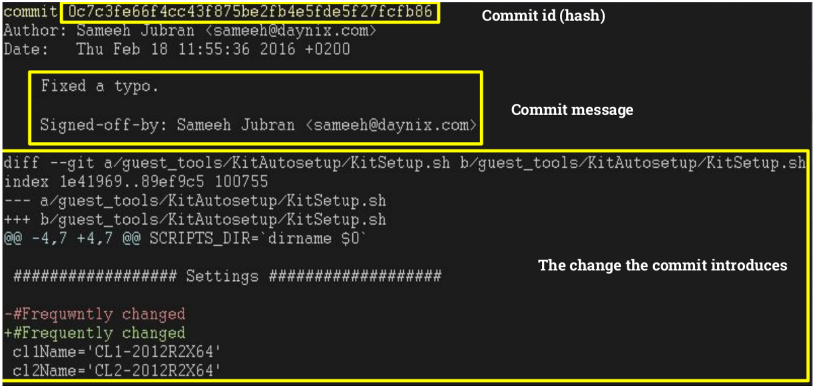
Git repository

- The purpose of git is to **manage a project**, or a set of **files**, as they **change over time**. Git stores this information in a data structure called a **repository**
- A git repository contains, mainly:
  - A set of commits



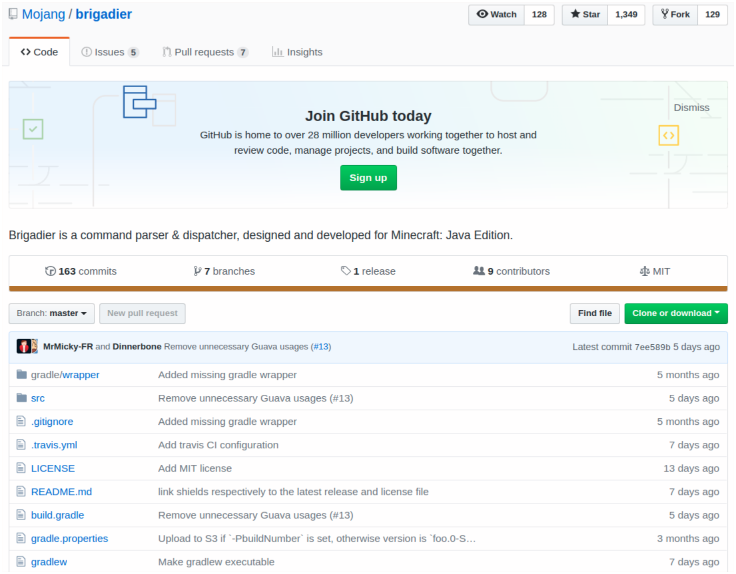
Commit

- A **commit** object mainly contains three things:
  - A set of **changes** the **commit** introduces
  - **Commit message** describing the changes
  - A **hash**, a 40-character string that uniquely identifies the commit object



```
commit 0c7c3fe66f4cc43f875be2fb4e5fde5f27fcfb86
Author: Sameeh Jubran <sameeh@daynix.com>
Date:   Thu Feb 18 11:55:36 2016 +0200

    Fixed a typo.

    Signed-off-by: Sameeh Jubran <sameeh@daynix.com>
diff --git a/guest_tools/KitAutosetup/KitSetup.sh b/guest_tools/KitAutosetup/KitSetup.sh
index 1e41969..89ef9c5 100755
--- a/guest_tools/KitAutosetup/KitSetup.sh
+++ b/guest_tools/KitAutosetup/KitSetup.sh
@@ -4,7 +4,7 @@ SCRIPTS_DIR=`dirname $0`

 ################# Settings ##################

-#Frequwntly changed
+#Frequently changed
 cl1Name='CL1-2012R2X64'
 cl2Name='CL2-2012R2X64'
```

GitHub


Mojang / brigadier

Watch 128 | Star 1,349 | Fork 129

Code | Issues 5 | Pull requests 7 | Insights

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Dismiss

Brigadier is a command parser & dispatcher, designed and developed for Minecraft: Java Edition.

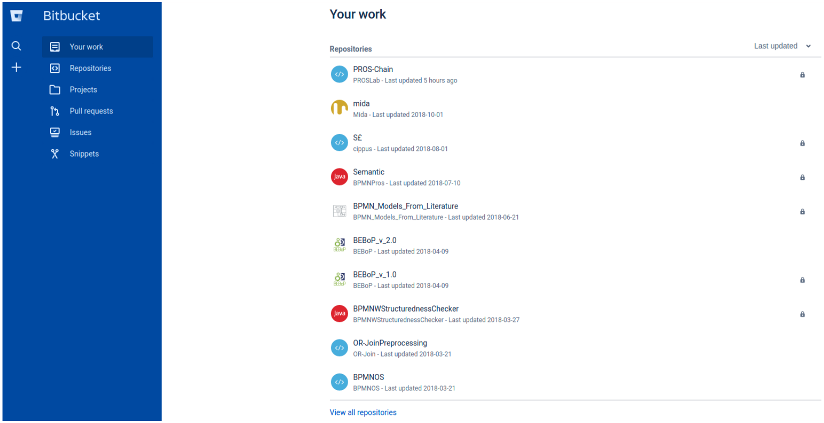
163 commits | **7** branches | **1** release | **9** contributors | MIT

Branch: **master** | New pull request | Find file | Clone or download

MrMicky-FR and **Dinnerbone** Remove unnecessary Guava usages (#13) | Latest commit 7ee589b 5 days ago

gradle/wrapper	Added missing gradle wrapper	5 months ago
src	Remove unnecessary Guava usages (#13)	5 days ago
.gitignore	Added missing gradle wrapper	5 months ago
.travis.yml	Add travis CI configuration	7 days ago
LICENSE	Add MIT license	13 days ago
README.md	link shields respectively to the latest release and license file	7 days ago
build.gradle	Remove unnecessary Guava usages (#13)	5 days ago
gradle.properties	Upload to S3 if `-PbuildNumber` is set, otherwise version is `foo.0-S…	3 months ago
gradlew	Make gradlew executable	7 days ago

Atlassian



Your work

Repositories — Last updated

PROS-Chain
PROSLab - Last updated 5 hours ago

mida
Mida - Last updated 2018-10-01

S£
cippus - Last updated 2018-08-01

Semantic
BPMNPros - Last updated 2018-07-10

BPMN_Models_From_Literature
BPMN_Models_From_Literature - Last updated 2018-06-21

BEBoP_v_2.0
BEBoP - Last updated 2018-04-09

BEBoP_v_1.0
BEBoP - Last updated 2018-04-09

BPMNWStructurednessChecker
BPMNWStructurednessChecker - Last updated 2018-03-27

OR-JoinPreprocessing
OR-Join - Last updated 2018-03-21

BPMNOS
BPMNOS - Last updated 2018-03-21

View all repositories

Git Cheat Sheet

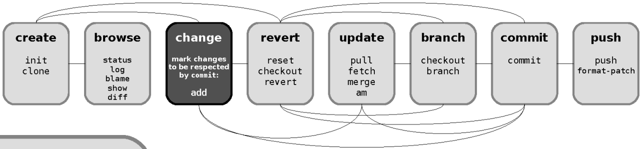
by Jan Krüger <jk@jk.gs>, http://jan-krueger.net/git/
Based on work by Zack Rusin

Basics

Use git help [*command*] if you're stuck.

`master`	default devel branch
`origin`	default upstream branch
`HEAD`	current branch
`HEAD^`	parent of HEAD
`HEAD~4`	great-great grandparent of HEAD
`foo..bar`	from branch *foo* to branch *bar*

(left to right) Command Flow



Create

From existing files
```
git init
git add .
```
From existing repository
```
git clone ~/old ~/new
git clone git://...
git clone ssh://...
```

Publish

In Git, `commit` only respects changes that have been marked explicitly with `add`.

```
git commit [-a]
```
(-a: add changed files automatically)
```
git format-patch origin
```
(create set of diffs)
```
git push remote
```
(push to origin or *remote*)
```
git tag foo
```
(mark current version)

Useful Tools

```
git archive
```
Create release tarball
```
git bisect
```
Binary search for defects
```
git cherry-pick
```
Take single commit from elsewhere
```
git fsck
```
Check tree
```
git gc
```
Compress metadata (performance)
```
git rebase
```
Forward-port local changes to remote branch
```
git remote add URL
```
Register a new remote repository for this tree
```
git stash
```
Temporarily set aside changes
```
git tag
```
(there's more to it)
```
gitk
```
Tk GUI for Git

Tracking Files

```
git add files
git mv old new
git rm files
git rm --cached files
```
(stop tracking but keep files in working dir)

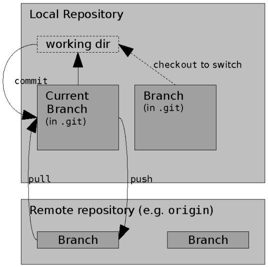
View

```
git status
git diff [oldid newid]
git log [-p] [file|dir]
git blame file
git show id
```
(meta data + diff)
```
git show id:file
git branch
```
(shows list, * = current)
```
git tag -l
```
(shows list)

Update

```
git fetch
```
(from def. upstream)
```
git fetch remote
git pull
```
(= fetch & merge)
```
git am -3 patch.mbox
git apply patch.diff
```

Structure Overview



Revert

In Git, `revert` usually describes a new commit that undoes previous commits.

```
git reset --hard (NO UNDO)
```
(**reset** to last commit)
```
git revert branch
git commit -a --amend
```
(**replaces** prev. commit)
```
git checkout id file
```

Branch

```
git checkout branch
```
(switch working dir to *branch*)
```
git merge branch
```
(merge into current)
```
git branch branch
```
(branch current)
```
git checkout -b new other
```
(branch *new* from *other* and switch to it)

Conflicts

Use add to mark files as resolved.

```
git diff [--base]
git diff --ours
git diff --theirs
git log --merge
gitk --merge
```

Installing Git

Before to start:

`https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`

Git Configuration

When we use Git on a new computer for the first time, we need to **configure** a few thing:

- our name and email address,
- what our preferred text editor is,

```
$ git config --global user.name "Andrea Morichetta"
$ git config --global user.email "andrea.morichetta@unicam.it"
```

```
$ git config --global core.editor "gedit"
```

This user name and email will be associated with your subsequent Git activity, which means that any changes pushed to GitHub, BitBucket, GitLab or another Git host server in a later lesson will include this information.

Git Config List

The commands run above only need to be run once: the flag –global **tells Git to use the settings for every project**, in your user account, on this computer.
You can check your settings at any time:

```
$git config --list
```

Creating a Repository

First, let's create a directory in Desktop folder for our work and then move into that directory:

```
$ cd ~/Desktop
$ mkdir thesis
$ cd thesis
```

Then we tell Git to make project a repository; a place where Git can store versions of our files:

```
$ git init
```

It is important to note that **git init** will create a repository that includes subdirectories and their files; there is no need to create separate repositories nested within the **thesis** repository, whether subdirectories are present from the beginning or added later. Also, note that the creation of the **thesis** directory and its initialization as a repository are completely separate processes.

Git Repository

If we use ls to show the directory's contents, it appears that nothing has changed:

```
$ ls
```

But if we add the -a flag to show everything, we can see that Git has created a hidden directory within planets called .git:

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

Git uses this special sub-directory to store all the information about the project, including all files and sub-directories located within the project's directory. If we ever delete the **.git** sub-directory, we will lose the project's history.

Git Status

We can check that everything is set up correctly by asking Git to tell us the status of our project:

```
$ git status
```

Git exercise

Create a new chapter folder in the **thesis** repository

```
$ mkdir chapter     # make a sub-directory thesis/chapter
$ cd chapter        # go into chapter sub-directory
$ ls -a             # ensure that the  sub-directory is present
```

Git exercise

Create a new chapter folder in the **thesis** repository

```
$ mkdir chapter    # make a sub−directory thesis/chapter
$ cd chapter       # go into chapter sub−directory
$ ls −a            # ensure that the  sub−directory is present
```

Is the git init command, run inside the **chapter** sub-directory, required for tracking files stored in the thesis sub-directory?

Git exercise

Create a new chapter folder in the **thesis** repository

```
$ mkdir chapter     # make a sub-directory thesis/chapter
$ cd chapter        # go into chapter sub-directory
$ ls -a             # ensure that the  sub-directory is present
```

Is the git init command, run inside the **chapter** sub-directory, required for tracking files stored in the thesis sub-directory?

No. We do not need to make the **chapter** sub-directory a Git repository because the **thesis** repository will track all files, sub-directories, and sub-directory files under the **thesis** directory. Thus, in order to track all information about **thesis**, we only needed to add the **chapter** sub-directory to the **thesis** directory.
Additionally, Git repositories can **interfere with each other if they are "nested"**: the outer repository will try to version-control the inner repository. Therefore, it's best to create each new **Git repository in a separate directory**.

Tracking Changes

Questions

- How do I record changes in Git?
- How do I check the status of my version control repository?
- How do I record notes about what changes I made and why?

Objectives

- Go through the modify-add-commit cycle for one or more files.
- Explain where information is stored at each stage of that cycle.
- Distinguish between descriptive and non-descriptive commit messages.

Create file

First let's make sure we're still in the right directory. You should be in the **thesis** directory.

```
$ pwd
```

Let's create a file called **title.txt** that contains some notes about the title and authors. We'll use nano to edit the file; you can use whatever editor you like. In particular, this does not have to be the core.editor you set globally earlier. But remember, the bash command to create or edit a new file will depend on the editor you choose (it might not be nano).

```
$ nano title.txt
```

Type some text into the **title.txt** file.

```
$ git status
```

The "untracked files" message means that there's a file in the directory that Git isn't keeping track of.

Git Add

We can tell Git to track a file using git add:

```
$ git add title.txt
```

and then check that the right thing happened:

```
$ git status
```

Git Commit

Git now knows that it's supposed to keep track of title.txt, but it hasn't recorded these changes as a commit yet. To get it to do that, we need to run one more command:

```
$ git commit –m "Start notes on title file"
```

When we run git commit, Git takes everything we have told it to save by using git add and stores a copy permanently inside the special .git directory. This permanent copy is called a commit (or revision) and its short identifier is f22b25e. Your commit may have another identifier.

We use the -m flag (for "message") to record a short, descriptive, and specific comment that will help us remember later on what we did and why. If we just run git commit without the -m option, Git will launch nano so that we can write a longer message.

Git Log

If we run git status now:

```
$ git status

On branch master
nothing to commit, working directory clean
```

t tells us everything is up to date. If we want to know what we've done recently, we can ask Git to show us the project's history using git log:

```
$ git log


commit 26be42751f20cbcc113e8b2dab00c3b2b661ee7e (HEAD -> master)
Author: Andrea M <andrea.morichetta@unicam.it>
Date:   Wed Oct 10 17:05:34 2018 +0200

first commit
```

git log lists all commits made to a repository in reverse chronological order. The listing for each commit includes the commit's full identifier, the commit's author, when it was created, and the log message Git was given when the commit was created.

Change the title file

Now suppose that we want adds more information to the file title.txt.

```
$ nano title.txt

 This is a new line in the title.txt file
```

When we run git status now, it tells us that a file it already knows about has been modified:

```
$ 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 di

 modified:    title.txt

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

The last line is the key phrase: "no changes added to commit". We have changed this file, but we haven't told Git we will want to save those changes (which we do with git add) nor have we saved them (which we do with git commit)

Git Diff

It is good practice to always review our changes before saving them. We do this using git diff. This shows us the differences between the current state of the file and the most recently saved version:

```
$ git diff

diff --git a/title.txt b/title.txt
index d3e2104..81bc58d 100644
--- a/title.txt
+++ b/title.txt
@@ -1 +1,2 @@
 This is the first line
+This is the second line
```

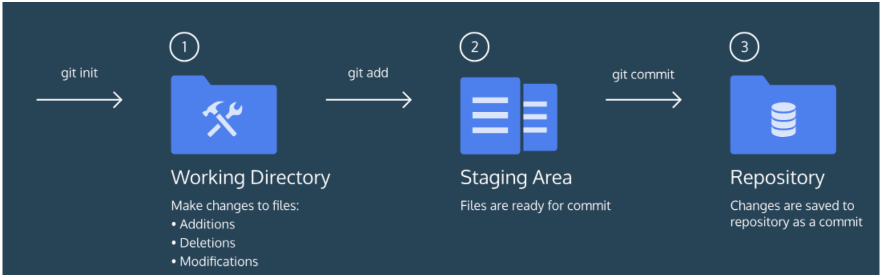
- The first line tells us that Git is comparing the old and new versions of the file.
- The second line tells which versions of the file Git is comparing; d3e2104 and 81bc58d are unique computer-generated labels for those versions.
- The third and fourth lines once again show the name of the file being changed.
- The remaining lines show the actual differences and the lines on which they occur. In particular, the + marker in the first column shows where we added a line.

Git Add & Commit

Git insists that we add files to the set we want to commit before actually committing anything. This allows us to commit our changes in stages and capture changes in logical portions rather than only large batches. For example, suppose we're adding a few citations to relevant research to our thesis. We might want to commit those additions, and the corresponding bibliography entries, but not commit some of our work drafting the conclusion (which we haven't finished yet).

```
$ git add title.txt
$ git commit -m "Added the second line in the tile file"
```

Where are my changes?



Git take snapshots of changes over the life of a project,
git add specifies what will go in a snapshot (putting things in the staging area), **git commit** then actually takes the snapshot, and makes a permanent record of it (as a commit).

Memory status changes

Add a new line in the title file and then:

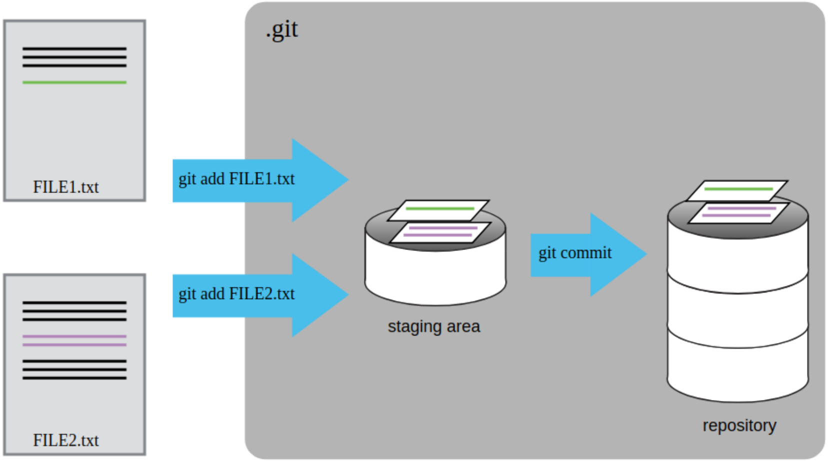
```
$ nano thesis.txt
$ git diff

$ git add thesis.txt
$ git diff //no output
$ git diff --staged

$ git commit -m "Add third line"
$ git diff --staged

$ git log
```

Git Log

To avoid having git log cover your entire terminal screen, you can limit the number of commits that Git lists by using -N, where N is the number of commits that you want to view. For example, if you only want information from the last commit you can use:

```
$ git log -1

$ git log --oneline

$ git log --oneline --graph --all --decorate
```

Multiple Add



Exploring Hystory

You can refer to the most recent commit of the working directory by using the identifier HEAD

```
$ git diff HEAD title.txt
```

If we want to see the differences between older commits we can use git diff again, but with the notation HEAD˜1 , HEAD˜2, and so on, to refer to them:

```
git diff HEAD"tilde"2 mars.txt

Or

$git log

$ git diff f22b25e3233b4645dabd0d81e651fe074bd8e73b title.txt
```

Exploring Hystory

We could also use **git show** which shows us what changes we made at an older commit as well as the commit message, rather than the differences between a commit and our working directory that we see by using **git diff**.

```
$ git show HEAD~2 mars.txt
```

Git Checkout

So we can save changes to files and see what we've changed

- **git status** tells us that the file has been changed, but those changes haven't been staged.
- now how can we restore older versions of things? Let's suppose we accidentally overwrite our 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 di
18:40
modified:    title.txt

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

We can put things back the way they were by using git checkout:

```
$ git checkout HEAD title.txt

$ cat title.txt
```

Git Checkout to an Older Version

If we want to go back even further, we can use a commit identifier instead:

```
 $ git checkout f22b25e title.txt
```

Ignoring Things

What if we have files that we do not want Git to track for us, like backup files created by our editor or intermediate files created during data analysis?
Let's create a few dummy files:

```
$ mkdir newFolder
$ touch a.dat b.dat c.dat results/a.out results/b.out
```

and see what Git says:

```
$ git status
```

Putting these files under version control would be a waste of disk space. Having them all listed could distract us from changes that actually matter, so let's tell Git to ignore them.

.gitignore

We can ignore dummy files creating in the root directory of our project a file called **.gitignore**

```
$ nano .gitignore
$ cat .gitignore
```

```
*.dat
results/
```

These patterns tell Git to ignore any file whose name ends in **.dat** and everything in the **newFolder** directory. (If any of these files were already being tracked, Git would continue to track them.)
Once we have created this file, the output of **git status** is much cleaner:

```
$ git status
```

We should track .gitignore?

everyone we're sharing our repository with will probably want to ignore the same things that we're ignoring. Let's add and commit **.gitignore**:

```
$ git add .gitignore
$ git commit -m "Ignore data files and the results folder."
$ git status
```

Using **.gitignore** helps us avoid accidentally adding to the repository files that we don't want to track:

```
$ git add a.dat
```

If we really want to override our ignore settings, we can use git add **-f** to force Git to add something. For example, **git add -f a.dat**. We can also always see the status of ignored files if we want:

```
$ git status --ignored
```

Remotes in Git

Version control really comes into its own when we begin to collaborate with other people. We already have most of the machinery we need to do this; the only thing missing is to copy changes from one repository to another.

Systems like **Git** allow us to move work between any two repositories. In practice, though, it's easiest to use one copy as a central hub, and to keep it on the web rather than on someone's laptop.

Most programmers use hosting services like **GitHub, BitBucket or GitLab** to hold those master copies.

GitHub

- GitHub is the single largest host for Git repositories, and is the central point of collaboration for millions of developers and projects.
- A large percentage of all Git repositories are hosted on GitHub, and many open-source projects use it for Git hosting, issue tracking, code review, and other things.

Create a New Account

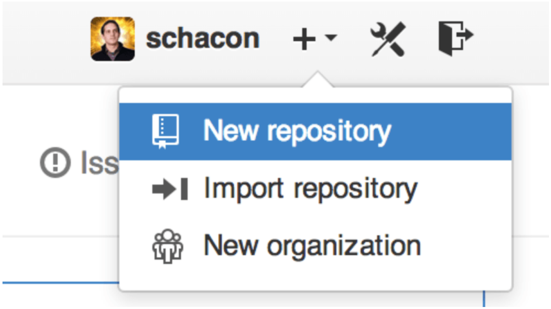
The first thing you need to do is set up a free user account. Simply visit `https://github.com`, choose a user name that isn't already taken, provide an email address and a password, and click the big green "Sign up for GitHub" button.

Maintaining a Project

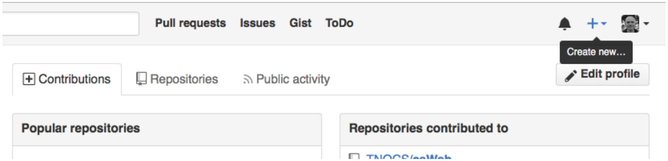
Creating a New Repository

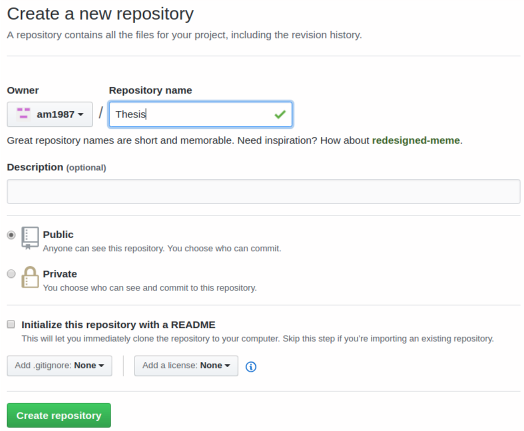
Let's create a new repository to share our project code with.

Start by clicking the **"New repository"** button on the right-hand side of the dashboard, or from the + button in the top toolbar next to your username as seen in The "New repository" dropdown.



Create a new repository on GitHub





Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: am1987 / Repository name: Thesis

Great repository names are short and memorable. Need inspiration? How about **redesigned-meme**.

Description (optional)

[✓ Public] Anyone can see this repository. You choose who can commit.

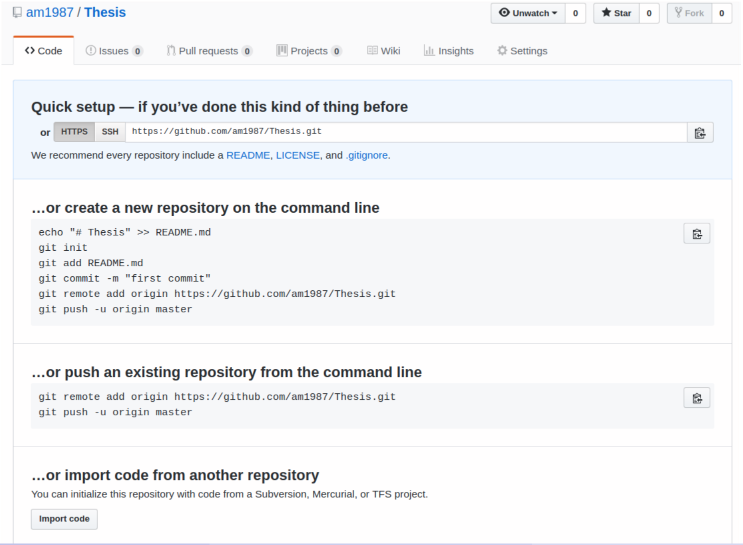
Private — You choose who can see and commit to this repository.

Initialize this repository with a README — This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None

New Repository

As soon as the repository is created, GitHub displays a page with a URL and some information on how to configure your local repository:



Quick setup — if you've done this kind of thing before

```
https://github.com/am1987/Thesis.git
```

We recommend every repository include a README, LICENSE, and .gitignore.

…or create a new repository on the command line

```
echo "# Thesis" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/am1987/Thesis.git
git push -u origin master
```

…or push an existing repository from the command line

```
git remote add origin https://github.com/am1987/Thesis.git
git push -u origin master
```

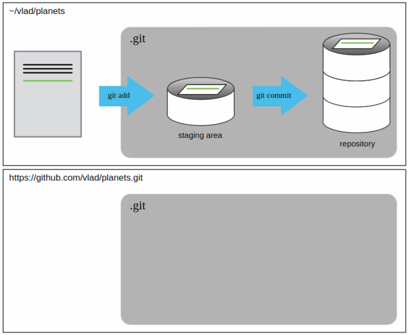
…or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

What happened in the Server?

This effectively does the following on GitHub's servers:

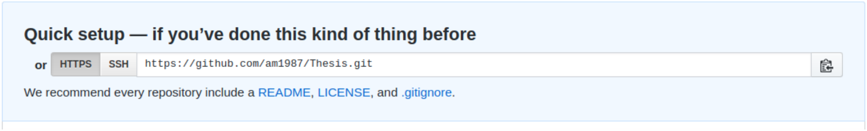
```
$ mkdir Thesis
$ cd Thesis
$ git init
```



Note that our local repository still contains our earlier work, but the remote repository on GitHub appears empty as it doesn't contain any files yet.

Connect Local and Remote Repositories

We do this by making the GitHub repository a remote for the local repository. The home page of the repository on GitHub includes the string we need to identify it:



Quick setup — if you've done this kind of thing before

https://github.com/am1987/Thesis.git

We recommend every repository include a README, LICENSE, and .gitignore.

Click on the **HTTPS**' link to change the protocol from SSH to HTTPS. Copy that URL from the browser, go into the local **thesis** repository, and run this command:

```
$ git remote add origin https://github.com/am1987/Thesis.git
```

We can check that the command has worked by running **git remote -v**:

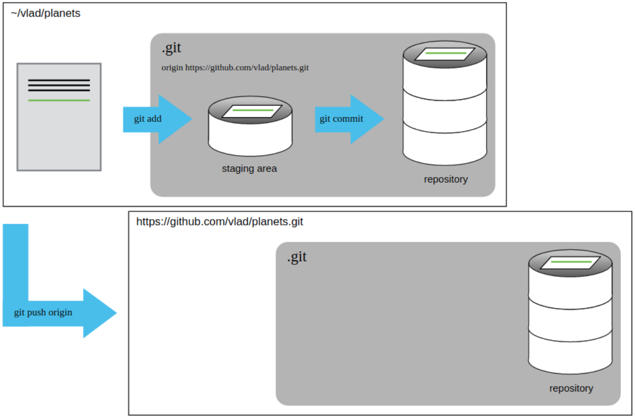
```
$ git remote -v
```

The name origin is a local nickname for your remote repository. We could use something else if we wanted to, but origin is by far the most common.

Git Push

Once the nickname origin is set up, this command will push the changes from our local repository to the repository on GitHub:

```
$ git push origin master
```



Curiosity

Push vs. Commit

When we push changes, we're interacting with a remote repository to update it with the changes we've made locally (often this corresponds to sharing the changes we've made with others). Commit only updates your local repository.

Fixing Remote Repository

The command **git remote set-url** allows us to change the remote's URL to fix it.

Git Pull

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

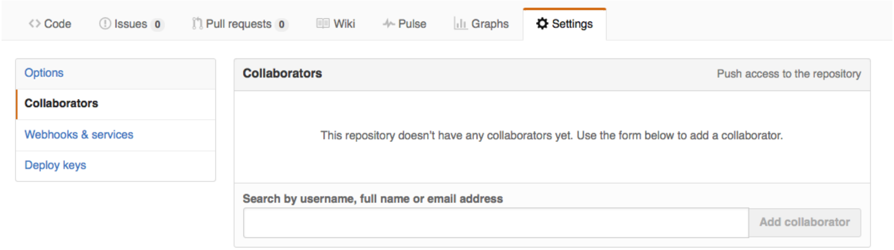
```
$ git pull origin master
```

While the git **fetch** command will fetch down all the changes on the server that you don't have yet, it will not modify your working directory at all. It will simply get the data for you and let you merge it yourself. However, there is a command called **git pull** which is essentially a **git fetch immediately followed** by a **git merge** in most cases.

Collaborating in Git

One person will be the "Owner" and the other will be the "Collaborator". The goal is that the Collaborator add changes into the Owner's repository.

The Owner needs to give the **Collaborator access**. On GitHub, click the settings button on the right, then select Collaborators, and enter your partner's username.



Collaborators

Push access to the repository

This repository doesn't have any collaborators yet. Use the form below to add a collaborator.

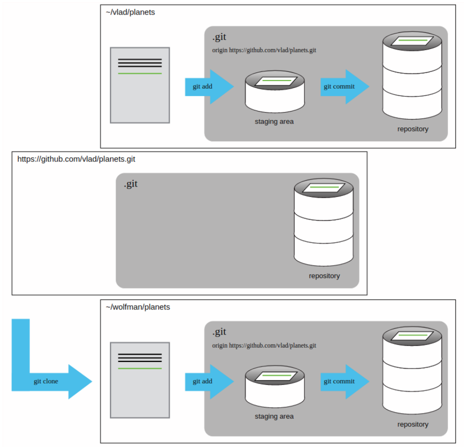
Search by username, full name or email address

To accept access to the Owner's repo, the Collaborator needs to go to https://github.com/notifications. Once there she can accept access to the Owner's repo.

Clone a project

The Collaborator needs to download a copy of the Owner's repository to her machine. This is called "**cloning a repo**". To clone the Owner's repo into her Desktop folder, the Collaborator enters:

```
$ git clone https://github.com/am1987/Thesis.git ~/Desktop/thesis
```



Collaborators Change

The Collaborator can now make a change in her clone of the Owner's repository, exactly the same way as we've been doing before:

```
$ cd ~/Desktop/thesis
$ nano title.txt
$ cat title.txt

$ git add title.txt
$ git commit -m "Add notes about new part"
```

Then push the change to the Owner's repository on GitHub:

```
$ git push origin master
```

Note that **we didn't have to create a remote called origin**: Git uses this name by default when we clone a repository. (This is why origin was a sensible choice earlier when we were setting up remotes by hand.)

Take a look to the Owner's repository on its GitHub website now (maybe you need to refresh your browser.) You should be able to see the new commit made by the Collaborator.

Download Collaborators Changes

```
$ git pull origin master
```

Now the three repositories (Owner's local, Collaborator's local, and Owner's on GitHub) are back in sync.

Summarizing

In practice, it is good to be sure that you have an updated version of the repository you are collaborating on, **so you should git pull before making our changes**.

The basic collaborative workflow would be:

- update your local repo with **git pull origin master**,
- make your changes and stage them with **git add**,
- commit your changes with **git commit -m**,
- upload the changes to GitHub with **git push origin master**

It is better to make many commits with smaller changes rather than of one commit with massive changes: **small commits are easier to read and review**.

Review Changes

On the command line, the Collaborator can use **git fetch origin master** to get the remote changes into the local repository, but without merging them.
Then by running **git diff master origin/master** the Collaborator will see the changes output in the terminal.

```
$ git fetch origin/master
$ git diff master origin/master
$ git merge origin/master
```

Conflict

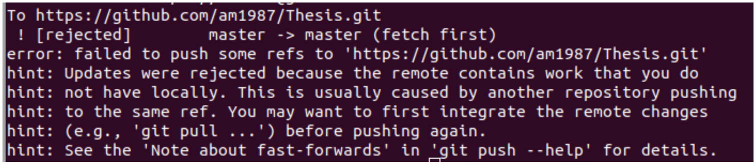
As soon as people can work in parallel, they'll likely step on each other's toes. This will even happen with a single person: if we are working on a piece of software on both our laptop and a server in the lab, we could make different changes to each copy. Version control helps us manage these conflicts by giving us tools to resolve overlapping changes.
Let's create a conflict modifing the same file between collaborators:

```
$ nano title.txt
$ git add title.txt
$ git commit -m "Add a line in our home copy"
$ git push origin master
```

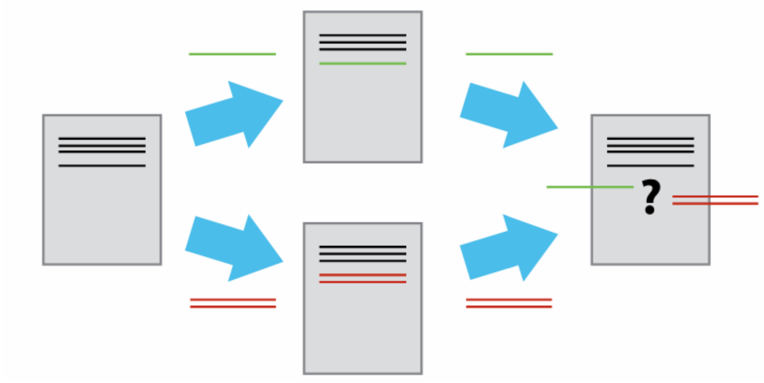
Now let's have the other partner make a different change to their copy without updating from GitHub:

```
$ nano title.txt
$ git add title.txt
$ git commit -m "Add a line in my copy"
$ git push origin master
```

Conflict error message



```
To https://github.com/am1987/Thesis.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/am1987/Thesis.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```



Solving the Conflict

Git rejects the push because it detects that the remote repository has new updates that have not been incorporated into the local branch. What we have to do is pull the changes from GitHub, merge them into the copy we're currently working in, and then push that. Let's start by pullin

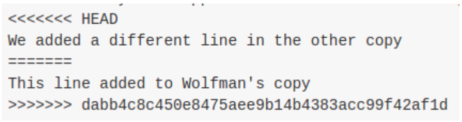
```
$ git pull origin master
```

The **git pull command updates the local repository** to include those changes already included in the remote repository. After the **changes from remote branch have been fetched**, Git detects that changes made to the local copy overlap with those made to the remote repository, and therefore **refuses to merge** the two versions to stop us from trampling on our previous work.

Conflict at Local repository

The conflict is marked in in the affected file:

```
$cat title.txt
```



```
<<<<<<< HEAD
We added a different line in the other copy
=======
This line added to Wolfman's copy
>>>>>>> dabb4c8c450e8475aee9b14b4383acc99f42af1d
```

Our change is preceded by <<<<<<< HEAD. Git has then inserted ======= as a separator between the conflicting changes and marked the end of the content downloaded from GitHub with >>>>>>>. (The string of letters and digits after that marker identifies the commit we've just downloaded.)

It is now up to us to edit this file to remove these markers and reconcile the changes. We can do anything we want: keep the change made in the local repository, keep the change made in the remote repository, write something new to replace both, or get rid of the change entirely.

Replace the Conflict

To finish merging, we add title.txt to the changes being made by the merge and then commit:

```
$ nano title.txt
$ git add title.txt
$ git commit -m "Merge changes from GitHub"
$ git push origin master
```

Good Practice

- Pull from upstream **more frequently**, especially before starting new work
- **Use topic branches** to segregate work, merging to master when complete
- Make **smaller** more atomic **commits**
- **Break large files** into smaller ones so reduce the possibility that two authors will alter the same file simultaneously

Conflicts can also be minimized with:

- Clarify **who is responsible for what** areas with your collaborators
- Discuss what order tasks should be carried out in with your collaborators so that **tasks expected to change the same lines won't be worked on simultaneously**
- If the **conflicts are stylistic** churn (e.g. tabs vs. spaces), es**tablish a project convention that is governing and use code style tools** (e.g. htmltidy, perltidy, rubocop, etc.) to enforce, if necessary

A Typical Work Session

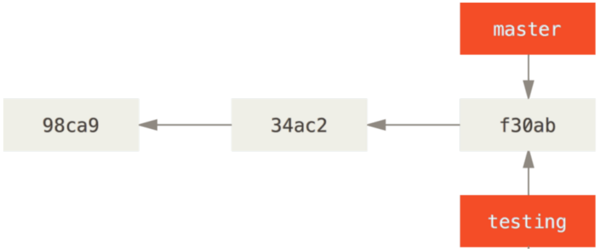
```
Update local     git pull origin master
Make changes     echo 100 >> numbers.txt
Stage changes    git add numbers.txt
Commit changes   git commit -m "Add 100 to numbers.txt"
Update remote    git push origin master
```

Creating a New Branch

Let's say you want to create a new branch called testing. You do this with the git branch command:

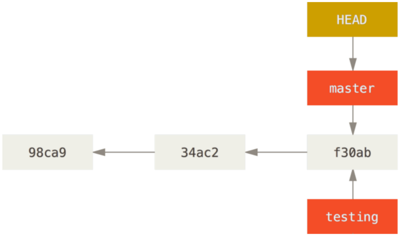
```
$ git branch testing

$ git branch
```

This creates a new pointer to the same commit you're currently on.



Current Branch (HEAD)

How does Git know what branch you're currently on? It keeps a special pointer called **HEAD**. In Git, this is a pointer to the local branch you're currently on.



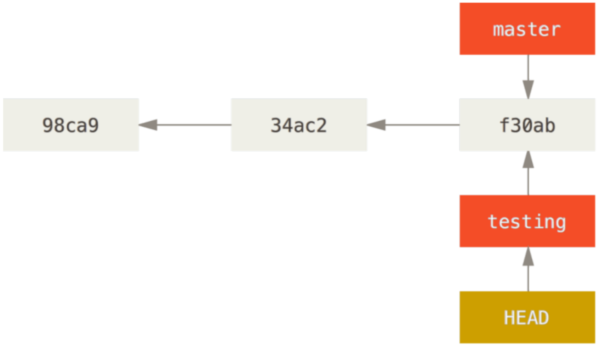
You can easily see this by running a simple git log command that shows you where the branch pointers are pointing.

```
$ git log --oneline --decorate
```

Switching Branches

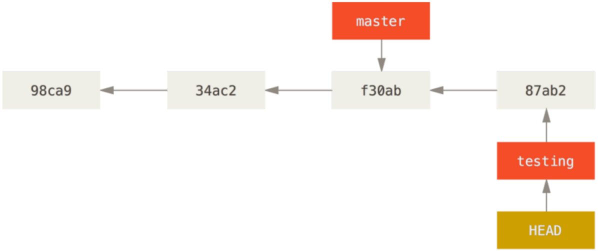
To switch to an existing branch, you run the git checkout command. Let's switch to the new testing branch:

```
$ git checkout testing
```



Let's do Another commit

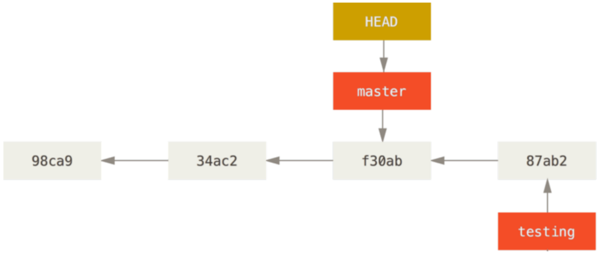
```
$ nano title.txt
$ git add title.txt
$ git commit -m 'made a change'
```



This is interesting, because now your testing branch has moved forward, but your master branch still points to the commit you were on when you ran git checkout to switch branches.

Switch to the Master

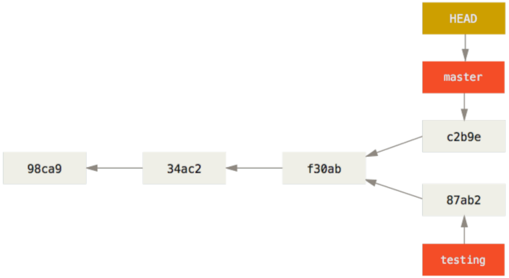
```
$ git checkout master
```



That command did two things. It moved the HEAD pointer back to point to the master branch, and it reverted the files in your working directory back to the snapshot that master points to.

Let's do Another Commit on the Master

```
$ nano title.txt
$ git add title.txt
$ git commit -m 'made a change'
```



Now the project is diverging. Both of those changes are isolated in separate branches: you can switch back and forth between the branches and merge them together when you're ready.

Git Log

If you run git log –oneline –decorate –graph –all it will print out the history of your commits, showing where your branch pointers are and how your history has diverged.

```
$ git log --oneline --decorate --graph --all
```

Git Merge

In order to merge two different branches you should use:

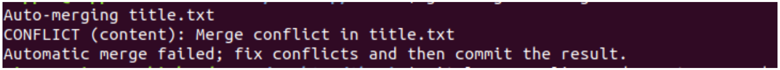
```
$ git checkout master
$ git merge testing
```

Now that your work is merged in, you have no further need for the testing branch. You can close the ticket in your ticket-tracking system, and delete the branch:

```
$ git branch -d testing
```

Basic Merge Conflicts

Occasionally, this process doesn't go smoothly. If you changed the same part of the same file differently in the two branches you're merging together, Git won't be able to merge them cleanly.



```
Auto-merging title.txt
CONFLICT (content): Merge conflict in title.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Git hasn't automatically created a new merge commit. It has paused the process while you resolve the conflict. If you want to see which files are unmerged at any point after a merge conflict, you can run git status:

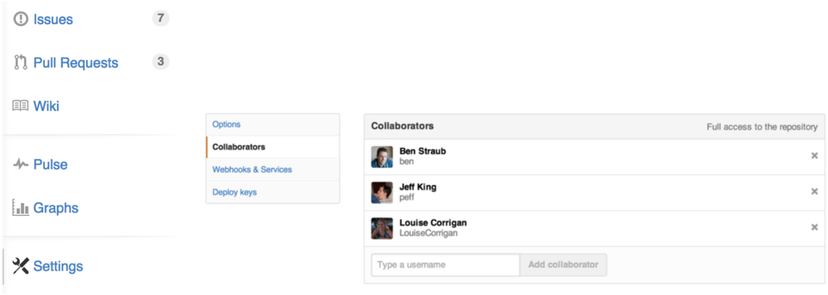
```
$ git status

$ nano title.txt
```

If you was able to solve the conflict, and you verify that everything that had conflicts has been staged, you can type **git commit** to finalize the merge commit.

Adding Collaborators in a GitHub Repository

If you're working with other people who you want to give commit access to, you need to add them as "collaborators".

Click the "Settings" link at the bottom of the right-hand sidebar.



Managing Pull Request

Pull requests let you tell others about changes you've pushed to a branch in a repository on GitHub.

Many open source projects on Github use pull requests to **manage changes from contributors** as they are useful in providing a way to notify project maintainers about changes one has made and in initiating code review and general discussion about a set of changes before being merged into the main branch.

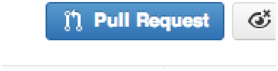
Creating a Pull Request

There are 2 main work flows when dealing with pull requests:

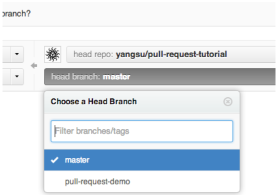
- Pull Request from a forked repository
- Pull Request from a branch within a repository

Creating a Pull Request

To create a pull request, you must have changes committed to the your new branch.
Go to the repository page on github. And click on "Pull Request" button in the repo header.

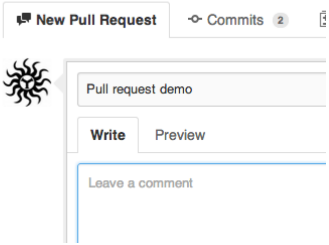


Pick the branch you wish to have merged using the "Head branch" dropdown. You should leave the rest of the fields as is, unless you are working from a remote branch.

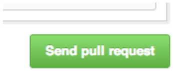


Creating a Pull Request

Enter a title and description for your pull request. Remember you can use Github Flavored Markdown in the description and comments

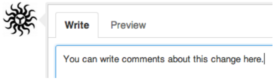


Finally, click on the green "Send pull request" button to finish creating the pull request.



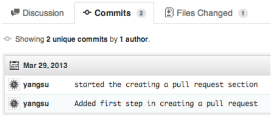
Using Pull Request

You can write comments related to a pull request,



You can write comments about this change here.

View all the commits contained by a pull request under the commits tab,

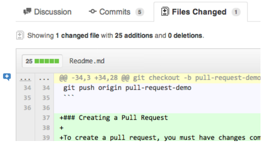


Showing 2 unique commits by 1 author.

Mar 29, 2013

yangsu	started the creating a pull request section
yangsu	Added first step in creating a pull request

See all the file changes from the pull request across all the commits under the "Files Changed" tab.



Showing 1 changed file with 25 additions and 0 deletions.

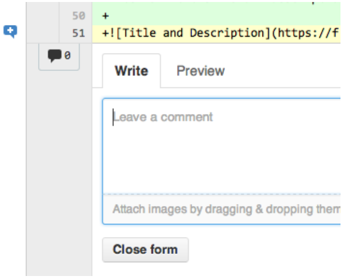
25 Readme.md

```
@@ -34,3 +34,28 @@ git checkout -b pull-request-demo
 git push origin pull-request-demo
 ```

+### Creating a Pull Request
+
+To create a pull request, you must have changes com
```
```

Using Pull Request

You can event leave a comment on particular lines in the code change simply by hovering to the left of a line and clicking on the blue note icon.

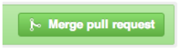


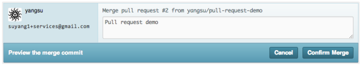
```
+
+![Title and Description](https://f
```

Merging a Pull Request

Once you and your collaborators are happy with the changes, you start to merge the changes back to master.

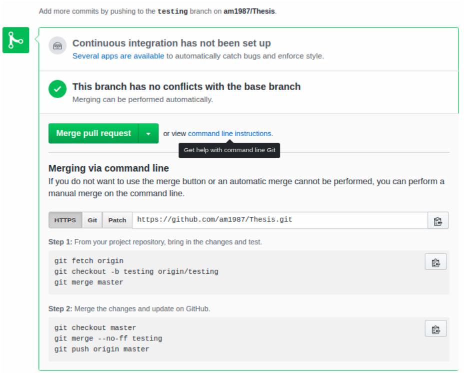
First, you can use github's "Merge pull request" button at the bottom of your pull request to merge your changes. This is only available when github can detect that there will be no merge conflicts with the base branch. If all goes well, you just have to add a commit message and click on "Confirm Merge" to merge the changes.




yangsu
suyang1+services@gmail.com
Merge pull request #2 from yangsu/pull-request-demo
Pull request demo

Merging Locally

If the pull request cannot be merged online due to merge conflicts, or you wish to test things locally before sending the merge to the repo on Github, you can perform the merge locally instead.
You can find the instruction to do so by clicking the (i) icon on the merge bar.



Add more commits by pushing to the `testing` branch on **am1987/Thesis**.

Continuous integration has not been set up
Several apps are available to automatically catch bugs and enforce style.

This branch has no conflicts with the base branch
Merging can be performed automatically.

or view command line instructions.

Merging via command line
If you do not want to use the merge button or an automatic merge cannot be performed, you can perform a manual merge on the command line.

```
https://github.com/am1987/Thesis.git
```

Step 1: From your project repository, bring in the changes and test.

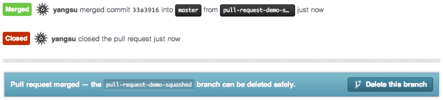
```
git fetch origin
git checkout -b testing origin/testing
git merge master
```

Step 2: Merge the changes and update on GitHub.

```
git checkout master
git merge --no-ff testing
git push origin master
```

Closing a Pull Request

You can simply click on the "Close" button on the pull request to close it. Optionally, you can delete the branch directly using the "Delete this branch" button.



Merged yangsu merged commit 33a3916 into master from pull-request-demo-s… just now

Closed yangsu closed the pull request just now

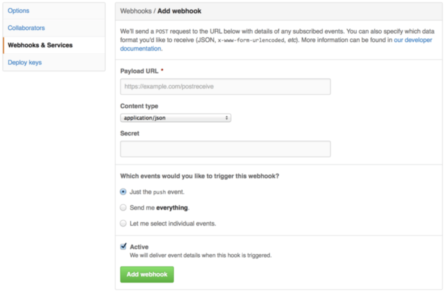
Pull request merged — the pull-request-demo-squashed branch can be deleted safely.

Hooks

The Hooks and Services section of GitHub repository administration is the easiest way to have **GitHub interact with external systems.**

You specify a URL and GitHub will post an HTTP payload to that URL on any event you want.

Generally the way this works is you can setup a small web service to listen for a GitHub hook payload and then do something with the data when it is received.



Webhooks / **Add webhook**

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, *etc*). More information can be found in our developer documentation.

Payload URL *

https://example.com/postreceive

Content type

application/json

Secret

Which events would you like to trigger this webhook?

● Just the push event.

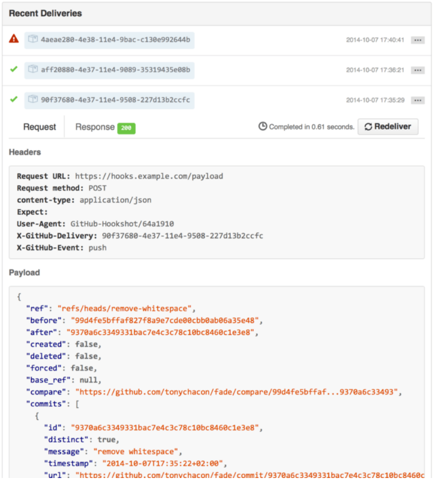
○ Send me **everything**.

○ Let me select individual events.

✓ Active

We will deliver event details when this hook is triggered.

Hook Example



Recent Deliveries

```
4aeae280-4e38-11e4-9bac-c130e992644b    2014-10-07 17:40:41
aff20880-4e37-11e4-9089-35319435e08b    2014-10-07 17:36:21
90f37680-4e37-11e4-9508-227d13b2ccfc    2014-10-07 17:35:29
```

Request | Response 200 — Completed in 0.61 seconds. Redeliver

Headers

```
Request URL: https://hooks.example.com/payload
Request method: POST
content-type: application/json
Expect:
User-Agent: GitHub-Hookshot/64a1910
X-GitHub-Delivery: 90f37680-4e37-11e4-9508-227d13b2ccfc
X-GitHub-Event: push
```

Payload

```
{
  "ref": "refs/heads/remove-whitespace",
  "before": "99d4fe5bffaf827f8a9e7cde00cbb0ab06a35e48",
  "after": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
  "created": false,
  "deleted": false,
  "forced": false,
  "base_ref": null,
  "compare": "https://github.com/tonychacon/fade/compare/99d4fe5bffaf...9370a6c33493",
  "commits": [
    {
      "id": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
      "distinct": true,
      "message": "remove whitespace",
      "timestamp": "2014-10-07T17:35:22+02:00",
      "url": "https://github.com/tonychacon/fade/commit/9370a6c3349331bac7e4c3c78c10bc8460c
```

Fork a Project

`https://github.com/octocat/Spoon-Knife`

Questions?