

Git/GitHub

An Introduction

Version control

Version control (also known as source control or revision control) is the management of changes to documents, code, and other collections of information.

Changes are generally identified by a number or letter code, usually termed the "revision number".

For example, an initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on. Each revision is associated with a timestamp and the user making the change. Revisions can be compared, restored, and with some types of files, merged.

Version control

Some other version control software you might be familiar with:

- Microsoft Visual SourceSafe (uses file locking to prevent concurrent access between users)
- CVS (Concurrent Versions System)
- SVN (Apache Subversion)
 - Both CVS and SVN use a centralized model - code lives in a central repository and operations against it take place on a shared server, making it possible for developers to overwrite one another's work.
- hg (Mercurial) is another distributed system, very similar to Git

Git is:

- open source
- written in C, making it stable, high-performing and very fast

Git is:

- distributed:

You, the developer, clone the entire repository — the whole history of your project's source code — instead of just a snapshot of the latest version. This means that if there is a crash on the main server, there are many copies that survive.

Unlike many other VCSs, Git allows you to work locally - your ability to work is not dependent on network access or a central server. You only need to communicate with your team's development server when you are ready to add your contribution to the source code or clone the source code to use locally (ie, “push”, “pull”, “fetch”, or “clone”).

Typical workflow

1. Clone (copy) a repository to your local machine.
2. Make some changes to the code.
3. Commit those changes to your local copy of the repository.
4. Push those changes back up to the development server so they become part of the main project.

Getting started

Install git locally: <http://bit.ly/J6fdpy>

Mac:

```
$ sudo port  
install git-core +svn +doc  
+bash_completion +gitweb
```

Windows:

```
http://  
code.google.com/p/  
msysgit
```

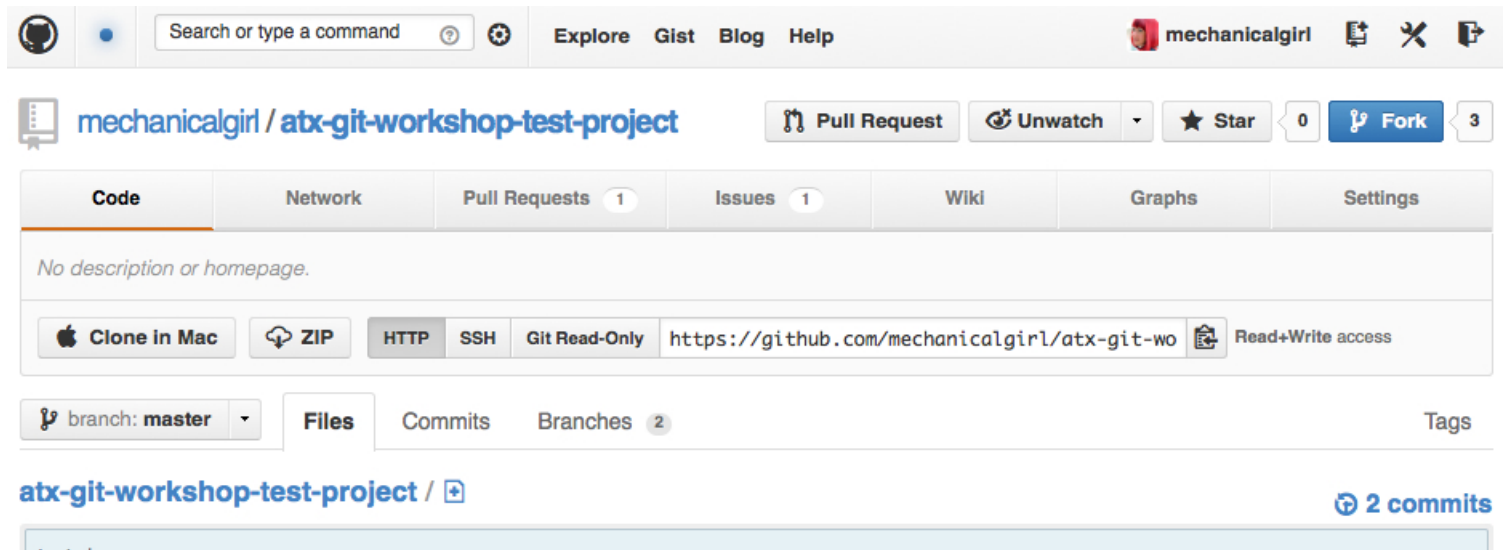
Linux:

```
$ yum install git-core  
$ apt-get install git
```

Getting started

- Create a GitHub account and log in
- Fork this repository:

`github.com/mechanicalgirl/atx-git-workshop-test-project`



Getting started

Open a terminal window,
create a folder where your Git projects will live,
then navigate to that folder:

```
$ cd ~                # change to your home directory  
$ mkdir GitHub        # make a directory named 'GitHub'  
$ cd GitHub           # change to that GitHub directory
```

Getting help

If you ever get stuck, git has a handy cheat sheet available any time.

Just type 'git help' in your terminal window, and you'll get something like this:

```
[barbara@bshaurette atx-git-workshop-test-project]$ git help
usage: git [--version] [--exec-path[=<path>]] [--html-path]
        [-p|--paginate|--no-pager] [--no-replace-objects]
        [--bare] [--git-dir=<path>] [--work-tree=<path>]
        [-c name=value] [--help]
        <command> [<args>]

The most commonly used git commands are:
add      Add file contents to the index
bisect   Find by binary search the change that introduced a bug
branch   List, create, or delete branches
checkout Checkout a branch or paths to the working tree
clone    Clone a repository into a new directory
commit   Record changes to the repository
diff     Show changes between commits, commit and working tree, etc
fetch    Download objects and refs from another repository
grep     Print lines matching a pattern
init     Create an empty git repository or reinitialize an existing one
log      Show commit logs
merge    Join two or more development histories together
mv       Move or rename a file, a directory, or a symlink
pull     Fetch from and merge with another repository or a local branch
push     Update remote refs along with associated objects
rebase   Forward-port local commits to the updated upstream head
reset    Reset current HEAD to the specified state
rm       Remove files from the working tree and from the index
show     Show various types of objects
status   Show the working tree status
tag      Create, list, delete or verify a tag object signed with GPG

See 'git help <command>' for more information on a specific command.
```

Cloning a repository

In your terminal window, type this command:

```
$ git clone https://github.com/{your username}/atx-git-workshop-test-project
```

This brings a copy of the entire repository down to your computer so that you can work with it locally.

Edit some code

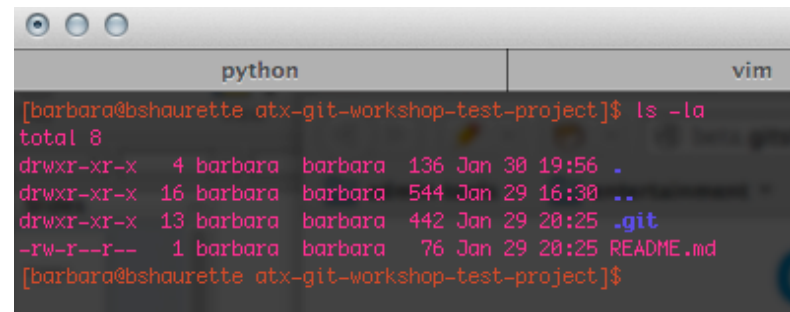
Navigate into the working directory:

```
$ cd atx-git-workshop-test-project
```

Open the README file and add your name to the list:

```
$ vi README.md
```

(you can also do this with the text editor of your choice)

A terminal window titled 'python' and 'vim' showing the output of the command 'ls -la' in the directory 'atx-git-workshop-test-project'. The output lists the current directory, parent directory, and a '.git' directory, along with a 'README.md' file.

```
[barbara@bshaurette atx-git-workshop-test-project]$ ls -la
total 8
drwxr-xr-x  4 barbara  barbara  136 Jan 30 19:56 .
drwxr-xr-x 16 barbara  barbara  544 Jan 29 16:30 ..
drwxr-xr-x 13 barbara  barbara  442 Jan 29 20:25 .git
-rw-r--r--  1 barbara  barbara   76 Jan 29 20:25 README.md
[barbara@bshaurette atx-git-workshop-test-project]$
```

Edit some code

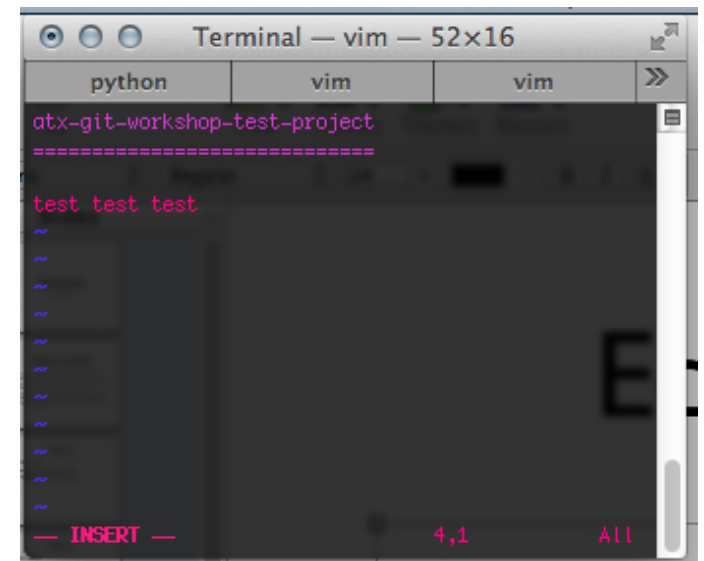
A quick vim lesson:

Type the letter 'i' to enter 'INSERT' mode on the file - this allows you to type, delete, and edit text.

Type your name anywhere in the file. (You should be able to use arrow keys to move up and down.)

Hit the ESC key to get out of INSERT mode.

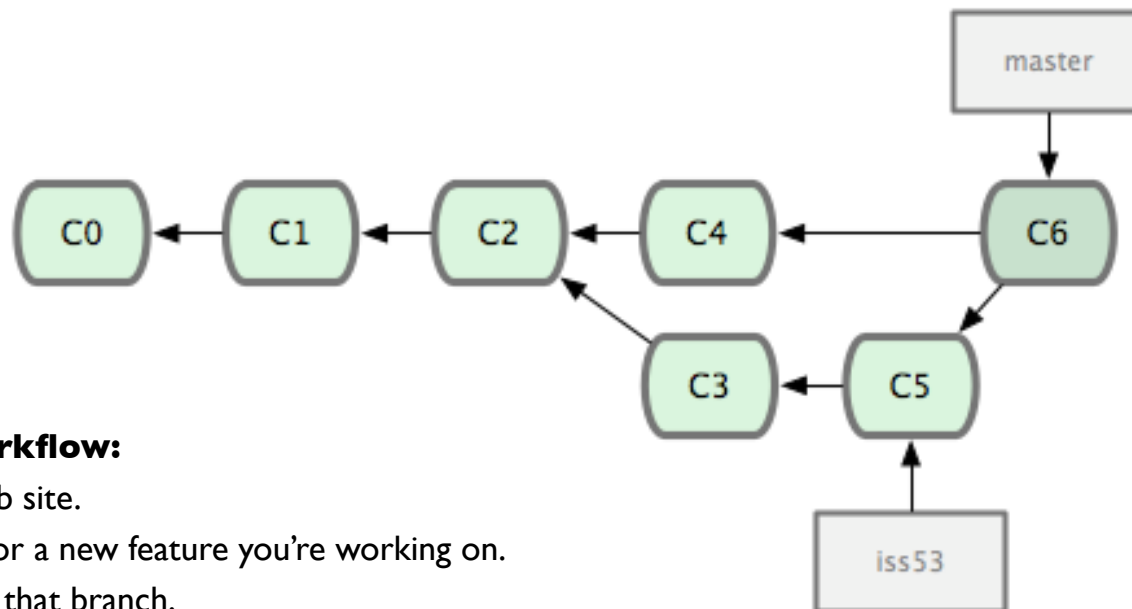
Type SHIFT-Z-Z to save and exit the file.



The screenshot shows a terminal window titled "Terminal — vim — 52x16". The window has three tabs: "python", "vim", and "vim", with the second "vim" tab selected. The editor content shows a file named "atx-git-workshop-test-project" with a header line of equals signs. Below this, the text "test test test" is on the first line, followed by several lines of tilde (~) characters. The status bar at the bottom indicates "— INSERT —" on the left, "4,1" in the center, and "ALL" on the right.

Branches

Before we actually commit that change, let's take a look at a few other things you can do.



An example workflow:

1. Do work on a web site.
2. Create a branch for a new feature you're working on.
3. Do some work in that branch.
4. Receive a call that another issue is critical and you need a bug fix.
5. Revert back to your master branch.
6. Create a branch to add the bug fix.
7. After it's tested, merge the bug fix branch back into mater and push to production.
8. Switch back to your original feature branch and continue working.

Branches

In your terminal window, type this:

```
$ git branch
```

You should see something like this:

```
* master
```

This indicates that you're currently working out of the branch named "master".

Branches

Create a new branch:

```
$ git checkout -b mynewbranch
```

The "-b" modifier on that command tells git that you're creating a new branch on this repository.

```
$ git branch
```

Now you should see that you've created and switched over to the new branch, but that the original "master" branch is still on your system:

```
master  
* mynewbranch
```


Branches

Go back to the master branch - to do that, you'll just run 'git checkout' again, but this time without that '-b' modifier:

```
$ git checkout master
```

Now that the new branch has been created, you can switch back and forth as much as you like:

```
$ git checkout mynewbranch  
$ git checkout master
```

git checkout

Another `git checkout` trick:

If you edit a file and then realize that something has gone horribly wrong and you want to blow it away completely and revert back to the latest version from the repository, all you have to do is run '`git checkout`' against that file.

```
$ git checkout myfile
```

In this case, `git checkout` will blow away any local changes (provided that you have not committed them yet), and replace the file with the latest committed version.

Extras: diff

'git diff' is a command that is used to display the differences between branches and files, staged or unstaged.

Type this into your terminal:

```
$ git diff
```

You should see references to code that has been changed since the last commit, but that has not yet been staged or committed. Adding '--cached' at the end of the command would show you staged changes if you had any at this point.

Some other ways to use diff:

```
$ git checkout my_new_branch
$ git diff --name-status origin/master
# To preview file changes between branches before you merge

$ git diff master mynewbranch -- index.html
$ git diff mynewbranch master -- index.html
# To preview differences in a specific file between branches
```

File states:

Git recognizes three different file states:

1. **Modified:** you have changed a file, but have not yet added it to your repository
2. **Staged:** you have added a file in its current version to go into your next commit (`git add`)
3. **Committed:** the data is safely stored in your local repository (`git commit`)

File states:

Changes you made to the README.md file still only exist locally and have not become a permanent part of the repository.

Type: `$ git status`

You should see something like this:

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

The important part of that text is the line next to the file name - that indicates that the file "README.md" is in the "modified" state.

Also note the line at the bottom - to stage this file you can use the 'git add' command, or to stage and commit in one step, use 'git commit -a' (not recommended).

File states:

To take our modified "README.md" file and stage it, we're going to add the changed version to the repository:

```
$ git add README.md
```

Now run the status command again and you'll see a different output:

```
$ git status

# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   README.md
#
```

The file still shows as modified, but it should appear in a different color, and the reminder to use 'git add' no longer appears.

Commit your changes

This changed file is now staged - let's go ahead and commit the change to our local repository:

```
$ git commit -m "test change" README.md
```

The commit command takes the '-m' modifier, which must be immediately followed by a commit message - this one is simple, but in general it's a good idea to be as descriptive as you can without writing a novel.

The file name does not necessarily need to be appended to the end of the command - we'll talk about that a little later on. Now run the 'git status' command and you should see something like this:

```
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
# nothing to commit (working directory clean)
```

Adds and commits

A note about adds and commits - you can add/commit individual files if you need to, but you can also use wildcard characters and relative paths to add/commit multiple files at a time:

```
$ git add .
```

```
# adds the current directory (.) and everything under it
```

```
$ git add files/
```

```
# add ./files, ./files/foo.txt, and ./files/foo/bar.txt
```

```
$ git commit -m "my commit" *.html
```

```
# commit all files in the current directory ending in .html
```

```
$ git commit -m "my commit"
```

```
# w/no filename at the end, commits all staged files, recursively
```


Adds and commits

A case where you might want to use these different types of adds and commits:

Suppose you make changes to multiple files, but each change is related to a different part of a project, or a different bug report, etc.

So you might make all of your changes locally over the course of a day, but then commit them at the end of the day with different commit messages, e.g.:

Examples:

```
$ git commit -m "replace all tables with divs" *.html  
$ git commit -m "change sort functionality" *.py
```

Extras: git log

There are several ways to see logs of activity on your repository - the most commonly used is:

```
$ git log
```

You should see something like:

```
commit 52191b83cf7856afd06879eec7dad67c0016f63d
Author: Barbara Shaurette <barbara.shaurette@gmail.com>
Date:   Wed Jan 16 14:59:54 2013 -0600
```

```
test
```

```
commit 2cf3ffa5d5114a6f5b21bdcdb04aa4d64901aa85
Author: Barbara Shaurette <barbara.shaurette@gmail.com>
Date:   Wed Jan 16 10:26:10 2013 -0800
```

```
Initial commit
```

The parts of this log are fairly self-explanatory - a SHA-1 commit number, the name of the user and date on the commit, and the commit message.

Git configuration

Inside your working folder, you should see a directory named ".git" - this folder and its contents were created when you checked out (cloned) the repository.

Let's take a look at the config file inside this folder:

```
cat .git/config
```

You'll see something that looks like this:

```
[core]
  repositoryformatversion = 0
  filemode = true
  bare = false
  logallrefupdates = true
  ignorecase = true
[remote "origin"]
  fetch = +refs/heads/*:refs/remotes/origin/*
  url = https://github.com/.../atx-git-workshop-test-project
[branch "master"]
  remote = origin
  merge = refs/heads/master
```

Git configuration

The [core] section is unrelated to git remotes, so we'll ignore that for now.

The [remote "origin"] and [branch "master"] sections are related to how git will interact with remote repositories.

The [remote "origin"] section refers to the address of the remote repository. The name "origin" is basically shorthand for the full repository address.

This is important because "origin" is the name you'll be using to refer to the repository whenever you pull/push (check changes in or out).

For example, if you changed that entry to '[remote "austin"]', you would use the name "austin" for your pull/push commands.

```
[core]
  repositoryformatversion = 0
  filemode = true
  bare = false
  logallrefupdates = true
  ignorecase = true
[remote "origin"]
  fetch = +refs/heads/*:refs/remotes/origin/*
  url = https://github.com/.../atx-git-workshop-test-project
[branch "master"]
  remote = origin
  merge = refs/heads/master
```

For more information about what the sections of the config file mean:

<http://www.gitguys.com/topics/the-configuration-file-remote-section/>

<http://www.gitguys.com/topics/the-configuration-file-branch-section/>

Push your changes

Now let's push those committed files to the central repository.:

```
$ git push origin master
```

As we mentioned earlier,

1. "origin" is the shorthand name for the repository address.
2. "master" refers to the branch you're pushing (in other words, if you had made changes in a branch other than 'master', that branch name is what you'd be using here).

Git will ask for your GitHub username and password.

Once your file change is successfully pushed, you'll see a response like this:

```
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 302 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/.../atx-git-workshop-test-project
2cf3ffa..311ab22  master -> master
```

When you return to the repository on GitHub, you'll see your change reflected there:

```
https://github.com/{your username}/atx-git-workshop-test-project
```

Extras: git merge

Merging branches:

When you are finished working on a branch and are ready to merge it back into its parent (say the develop branch or the master branch), you must merge from within the parent branch:

```
$ git checkout master  
# Makes "master" the active branch
```

```
$ git merge branchName  
# Merges commits from branchName to master
```

If you need to delete a branch when you're done with it:

```
$ git branch -d branchName  
# deletes branchName branch
```

More on basic branching and merging:

<http://git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging>

Extras: pull vs. fetch

There are two ways to get commits from a remote repo or branch — fetch and pull.

fetch

This command is useful if you need to keep your repo up to date but are working on something that might break if you update your files.

Fetch retrieves any changes that you do not have from the target remote repository, but does not merge them with your current branch.

To integrate the commits into your local branch, you use "git merge," which combines the specified branches and prompts you if there are any conflicts.

```
$ git fetch origin master
# pulls in any new changes
# not present in your local repo
# without modifying your files:
# You would have to merge the files
# manually.
```

pull

In "git pull", git tries to do your work for you.

It is context sensitive, so Git will merge any pulled commits into the branch you are currently working in.

Because "git pull" automatically merges the commits without letting you review them first, if you don't closely manage your branches you may run into conflicts.

```
$ git pull origin master
# pulls commits from 'master' and
# merges them into your active branch
```

Extras: reflog

Another logging command, commonly used for resetting a repository and recovering lost commits, is:

```
$ git reflog
```

It returns something that looks like this:

```
52191b8 HEAD@{0}: commit: test
2cf3ffa HEAD@{1}: checkout: moving from master to mynewbranch
311ab22 HEAD@{2}: checkout: moving from mynewbranch to master
2cf3ffa HEAD@{3}: checkout: moving from 2cf3ffa5d5114a6f5b21bdcdb04aa4d64901aa85 to mynewbranch
2cf3ffa HEAD@{4}: clone: from https://github.com/.../atx-git-workshop-test-project
```

The first column is an unencrypted commit number.

The second column is the state of HEAD (a pointer to the top of the current branch).

The third column contains a description of the repo activity at that point.

To learn more about how git reflog can be used to recover lost commits, take a look at these blog posts:

<http://effectif.com/git/recovering-lost-git-commits>

<http://alblue.bandlem.com/2011/05/git-tip-of-week-reflogs.html>

<http://gitready.com/intermediate/2009/02/09/reflog-your-safety-net.html>

Further reading:

Branching Workflows

<http://git-scm.com/book/en/Git-Branching-Branching-Workflows>

Distributed Workflows

<http://git-scm.com/book/en/Distributed-Git-Distributed-Workflows>

Custom git configuration:

<http://git-scm.com/book/en/Customizing-Git-Git-Configuration>

Other resources:

<http://git-scm.com/book>

Using GitHub

GitHub is a web-based hosting service for projects that use the Git revision control system.

At it's most simple, it's a collection of repositories, but they've thrown a nice visual interface on top of it to give you features like:

- shortcuts into basic git commands, like cloning and forking
- the ability to join teams, find projects, and follow users and projects easily
- a simple way to watch for changes on projects you follow

Using GitHub

- Fork a project - We've done that already
- Make a change locally - Done
- Commit that change to your own version of the project - Done
- Make a 'pull request' to get your changes incorporated back into the original project.

Pull requests

Note that a “pull request” on GitHub is not precisely the same as the `git pull` command you might use to update your own repository.

On GitHub, a pull request lets the owner of a repository know that you have code that you would like to add to his/her project.

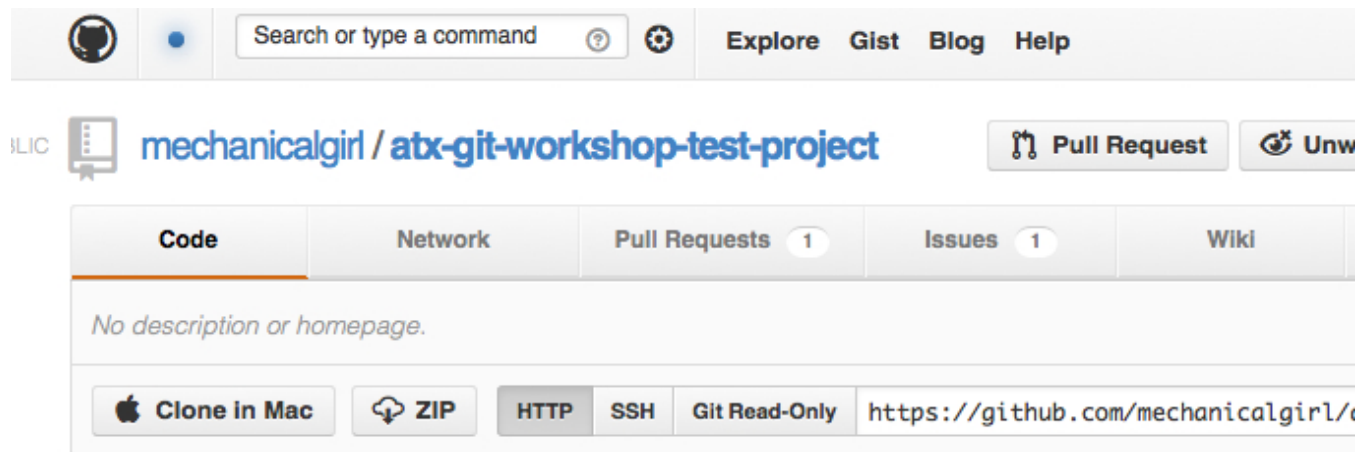
Pull requests

Make a request to the owner of a repository to merge your changes in:

1. Navigate to the forked version of the repository on your account:

`https://github.com/{your username}/atx-git-workshop-test-project`

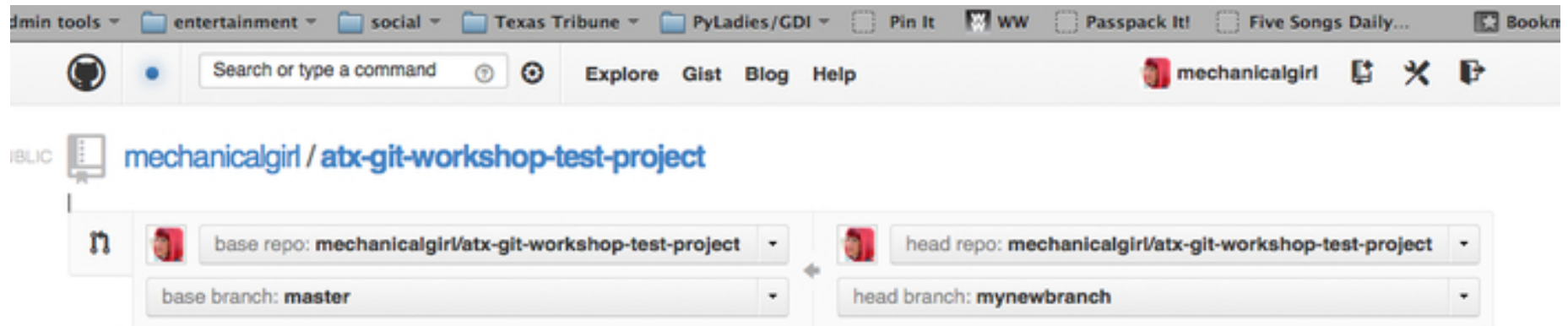
2. Click on the "Pull Request" button on the top nav bar



Pull requests

3. The request should show:

- target repository on the lefthand side
- source repository on the righthand side



Pull requests

4. Enter a brief description and submit the request - this creates a page that allows the manager of the repository to review the changes, accept/reject the code, and leave notes:

Open

mechanicalgirl wants to merge 1 commit into master from mynewbranch

4 ■■■■ #1

Discussion

Commits 1

Files Changed 1

Showing 1 changed file with 3 additions and 1 deletion.

Show Diff Stats

4 ■■■■ README.md

View file @ 52191b8

```

...  ...  @@ -1,2 +1,4 @@
1    1    atx-git-workshop-test-project
2    2    -----
      \ No newline at end of file
2    2    +-----
3    3    +
4    4    +test test test

```

+

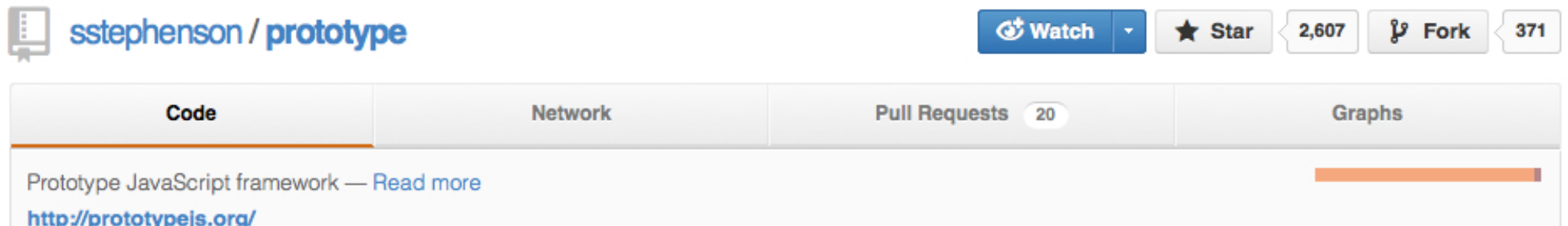
38 39 40

58 59 60

Tip: You can add notes to lines in a file.
 Hover to the left of a line to make a note

Watch and contribute

If there's a project you use regularly (such as the Django framework) and you want to keep an eye on changes as they are committed, you can use the 'watch' feature on GitHub.



The screenshot shows the GitHub repository page for `sstephenson / prototype`. The repository is a JavaScript framework. The page features a navigation bar with tabs for **Code**, **Network**, **Pull Requests** (with a count of 20), and **Graphs**. Below the navigation bar, the repository description is "Prototype JavaScript framework" with a [Read more](#) link. The URL <http://prototypejs.org/> is displayed below the description. On the right side of the repository header, there are buttons for **Watch**, **Star** (with a count of 2,607), and **Fork** (with a count of 371).


Watch and contribute




Once you've 'watched' a few projects, you'll see a feed like this on your GitHub home page:


When you're ready to contribute to one of these projects, you'll probably go through the fork/pull request process we just learned about.




News Feed




Pull Requests



 [mlavin](#) starred [nvie/pip-tools](#) an hour ago

 2 hours ago
[twig](#) opened pull request [django/django#681](#)
 Users with unsalted MD5 passwords unable to log in with Django 1.4
 1 commit with 1 addition and 1 deletion

 [coderanger](#) starred [nvie/pip-tools](#) 2 hours ago

 2 hours ago
[erikrose](#) pushed to testing at [mozilla/dxr](#)
  [c989397](#) Fix indentation to conform to Mozilla's (and the greater communit...

 3 hours ago
[abbeyj](#) opened pull request [mozilla/dxr#81](#)
 Fix member: queries for classes/structs that only contain member variables or member functions.
 2 commits with 37 additions and 12 deletions

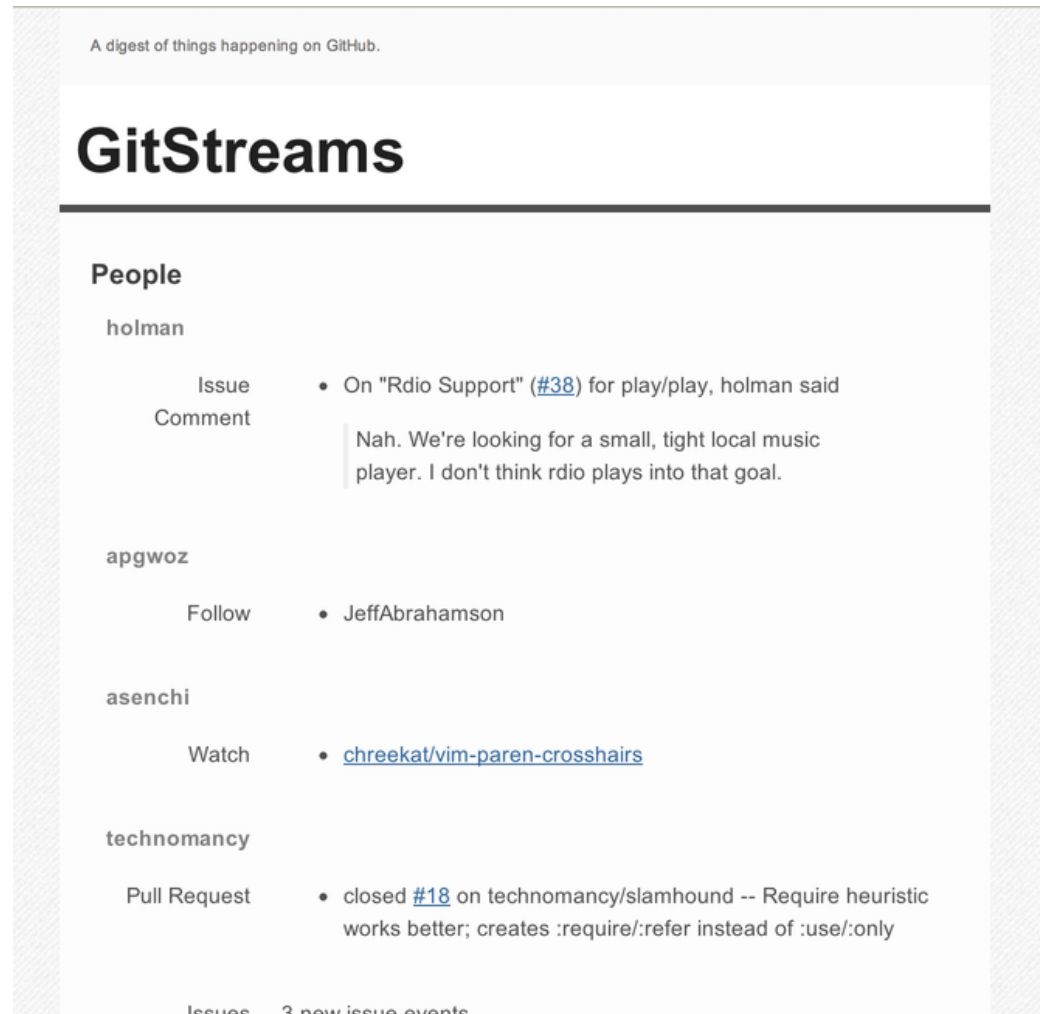
 3 hours ago
[charettes](#) commented on pull request [django/django#679](#)
 [@jonashaag](#) `min` and `max` values are only suitable for an input of type `"number"`. However, we can't use this input type when dealing with localized fi...

Watch and contribute

The GitHub activity stream can be a little overwhelming to keep up with on a daily basis.

One new tool hopes to change that, making it easier to track the important changes in projects you follow by providing a simple daily digest sent to you by email.

To sign up for the service, just go to <http://beta.gitstreams.com/>.



Gists

A 'gist' is a simple way to share code snippets with others.

All gists are git repositories, so they are automatically versioned and forkable.


The screenshot displays a GitHub Gist interface. At the top, there's a search bar and a 'Discover Gists' link. The user 'mechanicalgirl' is logged in, with their profile picture and name visible. The gist is titled 'gist:320b48b142ba2060c286' and is marked as 'SECRET'. It was created 6 hours ago. On the left sidebar, there are options to 'Download Gist', 'Clone this gist' (with a URL), 'Embed this gist' (with a script tag), and 'Link to this gist' (with a URL). The main content area shows a file named 'gistfile1.txt' containing Python code for a Django form. The code defines a class 'ImageAdminForm' that inherits from 'UniqueSEOSlugForm'. It includes a 'Meta' class with 'model = Image'. The 'def __init__(self, *args, **kwargs):' method checks for an 'initial' key in 'kwargs' and sets default values for 'pub_date', 'publication_status', 'authors', and 'tags'. It then updates the defaults with 'kwargs['initial']' and calls 'super(ImageAdminForm, self).__init__(*args, **kwargs)'. A comment at the bottom explains that the 'multiple' attribute was added to the 'image' field's widget to allow multiple file uploads.

```
1 # forms.py
2
3 class ImageAdminForm(UniqueSEOSlugForm):
4     class Meta:
5         model = Image
6
7     def __init__(self, *args, **kwargs):
8         if 'initial' in kwargs and kwargs['initial'] is not None:
9             defaults = {
10                 'pub_date': datetime.datetime.now(),
11                 'publication_status': 'P',
12                 'authors': (1,), # Admin
13                 'tags': (2414,), # The Texas Tribune | Company
14             }
15             defaults.update(kwargs['initial'])
16             kwargs['initial'] = defaults
17             super(ImageAdminForm, self).__init__(*args, **kwargs)
18
19 # This is the only thing I've added to the form - the 'multiple' attribute on the input,
20 # so that I can get multiple files into the request to test with
21 self.fields['image'].widget = forms.FileInput(attrs={'size' : '30', 'multiple': ''})
```

Gists





A gist can be:


- public: Anyone can see it.
- secret: Only someone with the link can get to it.


 **gist**


Search...


Discover Gists

 **mechanicalgirl**   


 [gist:320b48b142ba2...](#)
No description.

 [gist:2ee7c0add0498...](#)
PyLadies represent at Py...

 [gist:ae9612eccb2745...](#)
PyLadies represents at P...

 [gist:315b92fa7596aa...](#)
PyLadies represents at P...

Your Gists →



Gist description...

name this file...

language: **Text**

indent mode: **Spaces**

indent size: **2**

1 |

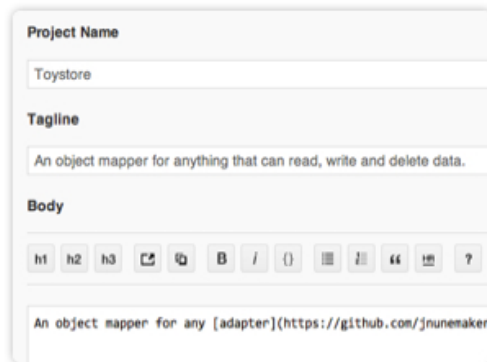
Add Another File

Create Secret Gist

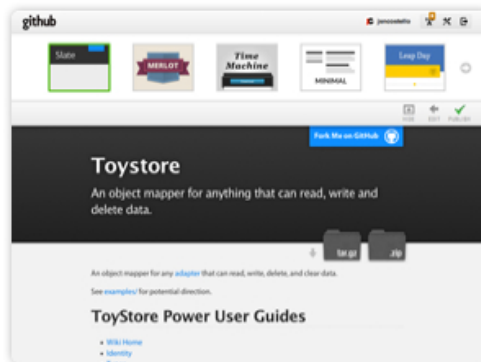
Create Public Gist

Pages

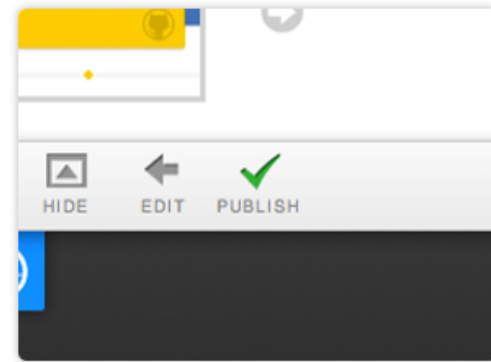
Create Project Pages In 3 Steps



Author



Theme



Publish



Free Hosting

GitHub Pages are hosted free and easily published through our site, the GitHub for Mac app, or from the command line. Manage your site's content from GitHub using the tools and workflow that you're familiar with, so you won't skip a beat. More about publishing with [GitHub Pages](#).

Page Generation with Themes

If you're creating a page for your project, check out our automatic Project Page generator. It lets you to author your page content in Markdown and toggle through our selection of designer themes. Many of our themes are responsive and include layouts optimized for mobile, so your page looks great on any device.

Manual Pages and Jekyll

There are a few ways to create GitHub Pages, including manually pushing html or a Jekyll site. You can easily redirect your page to a custom url. To read more about creating Pages manually with Jekyll, read the [documentation here](#).

<http://pages.github.com>