Getting Started with Git and GitHub

Summary: We explore the motivation for source code management systems, grounding our understanding in the Git source code management system and the GitHub repository.

A Short Introduction to Source Code Management

Large projects involve lots of files, people, and updates. Source code management (SCM) systems, or version control systems (VCS), were created to tame the chaos that usually ensues. SCM's let you keep track of old versions of files and share them with others, create *branches* where you can do experimental development, keep track of issues to be dealt with, and much much more.

There are many source code management systems. Their capabilities and approaches have varied over the years. For example, RCS (Revision Control System), the first version control system that I used, assumed that only one person would work on a file at a time. CVS (Concurrent Versioning System) added capabilities for multiple people to work simultaneously on the same file. But CVS wasn't great for Web-based collaboration, so Subversion (sometimes abbreviated as SVN) came along. Git (no acronym, at least as far as I know), like Subversion, supports simultaneous edits by people spread around the Web, but also distributes the repository to multiple machines.

A Quick Introduction to Git

For this course, we will be using Git because I use it in my day-to-day software (and course) development. Git is powerful, popular, and provides some interesting perspectives on source code management. Git is also relatively easy to use (although it may take awhile to understand some of its approaches).

We will also rely on GitHub.com, a popular Git hosting site, which provides a place to store our Git repositories. If you plan a career in software development, developing a presence on GitHub can be an important place to start. GitHub gives folks a chance not only to look at your portfolios, but also to observe your work habits.

In Git, there is a main repository where all of the official files for a project are stored. A few people (sometimes as few as one) control this repository. They are the only ones with the permission to make changes to files (or to approve changes that others propose). But anyone can propose changes.

In most cases, if you're not happy with what changes people allow in a repository, you can make your own copy of the repository on GitHub (or other Git hosting site), and you then control that repository, including the abillty to make changes to files that can then be shared with others.

More frequently, though, people simply work with a copy of the main repository, proposing changes for others when appropriate. To start, anyone working in the project can "clone" the latest copy of files from the main repository to work with, creating their own copy on a local machine. On the local machine, a person can open a file, edit it, and then "add" it to the staging area which holds all of the changes that

person makes. Once a person made a set of related changes that they are comfortable with, they "commit" it to their local repository - adding a note about the changes.

At some point, someone working on a project is comfortable enough with their changes that they want to share them with a larger group. (Note that you should generally not share with the larger group unless your code compiles and, ideally, passes at least as many tests as the previous version.) Administrators can "push" these changes back to the main repository. Other contributors send a "pull request" which the administrators may or may not accept.

Once changes have been propagated back to the main repository, collaborators can "pull" those changes back into their local copies.

So, a typical approach to working with a project that is under Git is something like the following. First, you clone the repository onto your local machine. You usually only need to do that once.

At the start of each session, you pull the latest changes from the primary repository. After all, it doesn't make much sense to work with old code - you might end up redoing changes that someone else made. (Sometimes when you pull from the main repository, you'll get conflicts with changes you've made but have not sent back to the main repository. In such cases, you'll need to "resolve" the conflicts and them commit them back to your local copy.)

You spend some time working on the code. At some point, you're satisfied with a bit of the code. At that point, you commit your changes to your repository. Different projects and different people seem to choose different levels of commits. In Git, commits are cheap, so I generally try to commit whenever I've made changes that fit into a single logical unit. (I also do this because, while it's easy to undo one or more commits, it's a pain to undo parts of a single commit.)

Finally, at some point you are ready to share your commits with the rest of your group. (Ideally, you'd have some changes to share at the end of each programming session, but that's rarely the case.) At that point, you do another pull (just in case someone else has made changes since your last pull), resolve any conflicts, and then send the changes back to the main repository, either via a push or a pull request.

Of course, professional programmers have a much deeper process. Most work with branches (alternate threads of development, so that folks can independently try different approaches to problems or otherwise attack issues that may require modifying multiple files in different ways).

You may have noticed that we used a lot of terms here. You may find it helpful to review some of them.

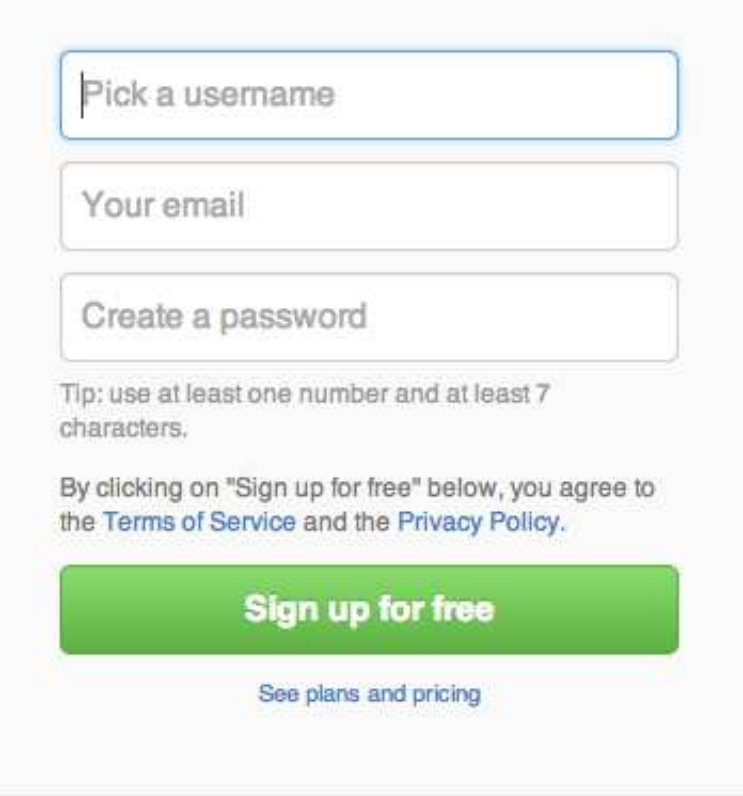
- *clone*: Copy a repository from Git to your computer.
- *add*: Add a file to your local repository. For a file already in your local repository, `add` adds the file to the list of things to commit.
- *commit*: Save and describe changes. (These changes are only saved in your local copy.)
- *pull*: Grab the latest version of files from the repository. (In some SCM's, this is called "update".)
- *merge*: Join together two copies of a file. When you merge things, you may have to resolve the differences by hand. Afterwards, you will usually need to add the updated file back to the list of things to be committed.
- *push*: Send all of your recent commits back to a repository that you administer.
- Send a *pull request*: Request that the administrator accept your recent commits into the main

repository.

Making a GitHub Account

We will be working with Git through GitHub. Hence, you will need your own account on GitHub. You can do these steps while doing the reading, or you can do them in class during our lab time.

Go to `https://GitHub.com/` Sign up for an account on the right hand side. Pick a username, email, and password then click the green “Sign up for free” button. Now you have an account on GitHub! GitHub hosts Git projects and gives you a chance to look at other people’s projects. You can collaborate with them or participate in challenges, or simply just put your own projects up on display.



Configuring Your MathLAN Account

In addition to setting up a GitHub account, it’s useful to configure your MathLAN account so that you can easily access Git We are going to set up your name and email in Git (accessible via terminal). You will only need to do this once. Once again, you will have the the opportunity to do this in the upcoming lab.

1. Open a terminal window.
2. Type the following command. (Don’t type the prompt.)

```
$ git config --global user.name YourName
```

This command will set up the name that accompanies your commits.

3. Next, set up your email (use the same one you used to sign up).

```
$ git config --global user.email username@grinnell.edu
```

This command will set up the name that accompanies your commits.

4. Each time you commit a change, Git will ask you to enter a commit message. By default, Git uses the editor Vim. While I like Vim, I admit that I'm in the minority. If you want to change the editor you use (e.g., to Emacs), type

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

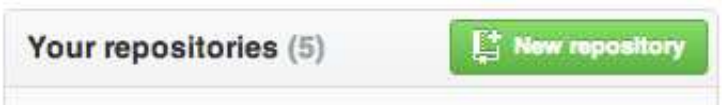
Using Git: A Walkthrough

Making Your Own Repository

Now that you have an account and you have set up your name and username in Git, let's consider how to make a repository on GitHub. There are two things that can only be done through GitHub: putting your project up on GitHub (making a repository on the website to house the project) and making a copy of someone else's project (forking someone else's repository).

After you log into GitHub, you should see a green button labelled New Repository toward the lower right-hand side of the page. Click that button. Give your repository a name and a brief description. I suggest that you also click Initialize this repository with a README, that you add a Java `.gitignore` file, and choose a license for your code. (I generally use LGPL v3.)

Once you've made your configuration decisions, click "create repository" You have now made a repository that others can see on GitHub (if you made the repository public)! This repository is the main repository.



You may be wondering what the `.gitignore` file is. By default, Git assumes that every file you have in your local repository should eventually go back to the main repository. The `.gitignore` file lets Git know that some files aren't needed. For example, you wouldn't want `.o` files or executables in a C repository. Similarly, you don't want `.class` files in a Java repository.

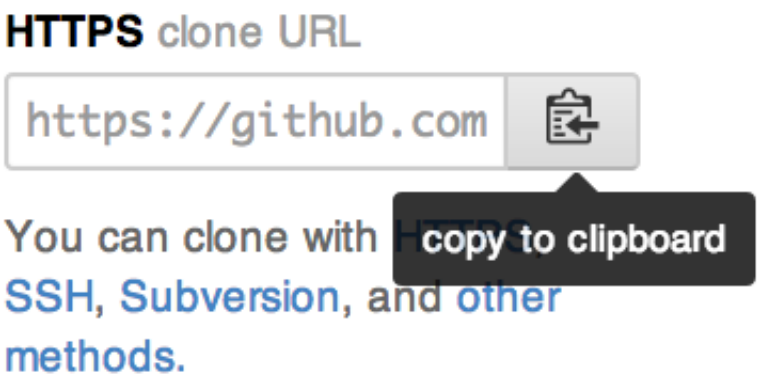
Cloning A Repository

Quick notes! Typing

```
$ git help
```

into the terminal will bring up a list of the most common Git commands and a brief description of what they do. You can also get help on a particular command with `git help` *`command`*. Also, remember that you need to precede every Git command with `git`. So, for example, to add something to a repository, you use `git add`, not just `add`.

1. First, get the URI of the repository you just created on GitHub. You can use the click the Copy to Clipboard button along the right column.



   You may find it equally useful to just grab the URL from your Web browser's bar.

2. Type in

   ```
   $ git clone uri
   ```

   For example, to clone the repository with this textbook, you would type

   ```
   $ git clone https://GitHub.com/Grinnell-CSC207/tao-of-java
   ```

You will now have a directory on your computer whose name matches that of the repository. Inside the directory will be all of the files in the original repository that you cloned so that you can edit, add, and/or remove files the way you normally do.

Note that you do not have to have your directory name match the name of the repository. You can rename the directory. Alternately, when you clone the repository, you can add a directory name.

```
$ git clone uri directory
```

When you clone a repository, you can only contribute to it if you are the creator or have been declared as one of the contributors. Otherwise, the code is just there for you to play around with and not upload.

Forking a Repository

If you want to be able to share changes to a repository that is not your own, and you don't think the owners will want your changes, you should "fork" the repository. When we fork a repository, we make a copy of the repository that we have control over. We can ask the original creator to accept changes in our forked version via a "pull request", and it's up to their discretion whether or not to accept it. Therefore, the fork is a copy of the repository you want to work with/on.

1. Find a project you would like to contribute to on GitHub. (Alternately, find a project we've told you to fork.)
2. Go to the repository that holds the project.
3. Click the button in the upper right-hand corner that says Fork.

4. GitHub will fork the repository for you.
5. Now, you have a copy of the repository that will show up under your repositories on GitHub. You can follow the steps in the section above and clone the forked repository into your machine so that you can work on it.

If you intend to interact with the main repository, you will need to create a "remote" connected to the main repository. The remote will let you pull updates from that repository and will also allow you to send pull requests back to that repository.

```
$ git remote add name uri
```

You can name the remote whatever makes sense to you, but it is convention to call it "upstream". Use the HTTP uri of the repository that you forked.

```
$ git remote add upstream https://GitHub.com/Grinnell-CSC207/tao-of-java
```

Now the main repository will be referred to as upstream, and you can pull updates from it to your local repository.

Adding Files and Making Changes

Now that you have either created your own project or joined one, lets begin editing.

Are you done yet? Good. Now we're ready to look at getting your changes into both the local and Web repositories.

If you're like most programmers, you may have forgotten what files you've created or changed and what changes you've made. Fortunately, Git provides tools to help you figure this out.

To check if there are any changes/additions to your local repository, type in:

```
$ git status
```

Git will then tell you a variety of things. It will tell you how many commits are in your local repository that haven't been pushed back to GitHub. It will tell you what files are staged for commit. It will tell you which files that are already in the repository have been modified but not committed. And it will tell you which new files have been created but not yet added to the repository.

Once in a while, it will tell you that no changes whatsoever have been made.

So far, so good. You have a list of general changes. But what if you want to know precisely what happened to an individual file? Git provides a way of finding that out, too.

```
$ git diff filename
```

The `git diff` command will show you the portions of a file that are changed. Usually added lines are prefixed with a plus sign, deleted lines a prefixed with a minus sign, and a few surrounding lines appear for context.

Adding Files

To add files into the local repository they must first exist in your working directory (the folder you made and initialized). Once they exist, you simply use the `git add` command.

```
$ git add filename
```

Editing Files

Editing files under Git is like editing any file. Open the file using some type of editor (emacs, vim, gedit, etc) make your changes, and then save.

Adding/Staging Changes

You may recall that `git status` tells you not just about modified files and new files, but also files that are "staged" for the next commit. You know how to stage a new file: Just use `git add`. How do you stage a modified file? Exactly the same way, with `git add filename`.

Staging puts the files into the staging area, which holds all of the changes made in the working directory that will be committed to the local repository. This distinction is important because only the files that you stage will be the ones that go into the local directory when you commit changes, and only the changes in the local repository will be added to the main repository when you push. It is good practice to only "add" what is necessary, nothing more (no temporary files, no executable files that can be generated, etc). If you do, the repository will only get cluttered, and if you were planning on sending a pull request, adding more than necessary will not be in your favor. (The student who wrote this paragraph discovered this after Sam

criticized zir repeatedly for making his life more difficult.)

Deleting Files

Deleting files from a repository is a bit like deleting files in any Unix system. The one difference is that you have to tell Git about it. So, instead of

```
$ rm file
```

you write

```
$ git rm file
```

Deletion is automatically staged. (Yeah, it just seems a bit to contradictory to use `git add` to stage a deletion.) But you still have to use `git commit` to confirm the deletion.

Committing

At this point, you are ready to incorporate some or all of your changes into the repository. First, make your that you add the changed or new files. Next, use Git's commit command.

```
$ git commit
```

Git will open the editor you specified above and you can enter information about the change. Custom is to use present tense in a somewhat imperative mode: "Fix bug 23a" or "Add capability to print". Enter something that will be useful to you and to others who may see your code, then save and exit the editor. Congratulations, you have successfully committed changes to your local repository!

Note: To avoid having any text editor pop up, use the -m flag to enter a message directly in the terminal. For example,

```
$ git commit -m 'Add readme file'
```

Notice the quotation marks that surround the message.

Pushing

At this point, you have changes in your local repository and you want to upload them to the primary repository. We'll start with a repository you own. The most basic command is

```
$ git push
```

Git will then prompt you for your username and password and upload the files.

As you become more advanced, you may want to specify the repository and branch you are using. In that case, you write

```
$ git push repository branch
```

By default, the repository is `origin` and the branch is `master`.

Warning! If you are working in a repository that you share with others, they may have pushed to the repository too. So make sure to "pull" from the repository before pushing. (Git will probably warn you if your push will create problems, but it's better to be safe.)

Pull Requests

Pushing works fine if you own the repository. But what if you forked a repository and want to send a request to have your changes be a part of the main code? Recall that you are dealing with (at least) three copies of the code: the original repository (out of your control), your fork of the repository (which you control), and your clone of your fork (where you do your editing).

To inform the owners of the primary repository that you have made changes that you think they would like to include, you send what is called a "pull request". It is difficult (and perhaps impossible) to send a pull request from the terminal; traditionally one does it through GitHub (at least for code stored on GitHub.

Since pull requests will be rare in this class, we'll leave it to you to figure out how to send one.

Once you've sent a pull request, it will be up to the owner of the main repository whether or not to accept the request.

Note: Just as you pulled from your repository before pushing, so should you pull from the main repository before sending a pull request. In this case, you need to name the main repository as described above (traditionally, you name it "upstream") and then issue the following command:

```
$ git pull upstream master
```

Keeping Your Local Repository up to Date

We've described this step a few times, but it's useful to hear it again.

You are going to want the latest version of the main repository in your local repository, so to do this you will need to pull. If your repository was not empty to begin with, you would have had to pull from it before pushing. Type in the following to pull your the repository you cloned.

```
$ git pull
```

If you forked your repository before cloning and want to grab updates from the original repository, you need to name that original repository and then pull from that repository.

```
$ git remote add upstream url
$ git pull upstream master
```

Wrapping Up

Important Terms

- Push/pull
- Commit
- Clone
- Fork
- Version Control
- GitHub
- Repository
- Stage

Review Questions

- Why is Git useful?
- What is forking and when would you use it?
- What is cloning and when would you use it?
- What is the normal sequence of working operations a programmer should use when dealing with a Git project?

Exploratory Questions

- Git allows you to undo commits (rolling your code back to a previous version). Figure out how.
- In addition to forks and clones, Git allows you to make variants of a repository using “branches”. Learn what a branch is, how to make one, and how to use branches.

Selected Sources

- `http://www.cs.grinnell.edu/~rebelsky/Courses/CSC195/2013S/Outlines/outline.02.html`
- `http://git-scm.com/book/en/Getting-Started-Git-Basics`
- `http://hoth.entp.com/output/git_for_designers.html`