THE UNIVERSITY OF WESTERN ONTARIO

Computer Science 2212b - Winter 2014
Introduction to Software Engineering

Lab 1 - Version Control with Git

Part I
Introduction to Git and GitHub

Estimated Time: 30 minutes

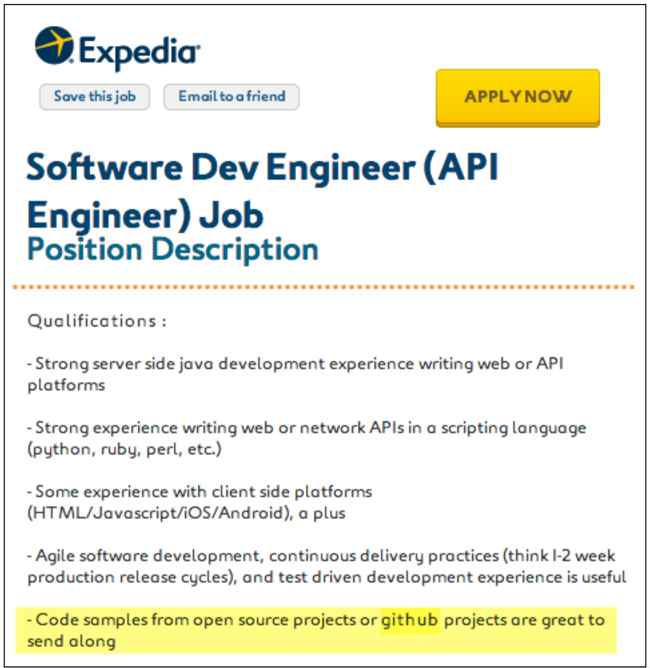
In a *version control* system, an individual or team of programmers store code in a *repository.* The version control system allows one to keep track of changes to files, examine the history of changes made to them, and potentially *roll back* changes made. It also allows multiple developers to work with the same code concurrently. When a developer is finished making changes to the code, she can *push* or *check in* her changes, and other developers can then *pull* or *check out* those changes into their local copies of the repository.

There are many version control systems out there – Git, Mercurial, Subversion, CVS, and Bazaar, to name a few – so why Git? Well, everyone is using Git these days. Git was initially created by Linus Torvalds (of Linux fame) to host the development of the Linux kernel. Since then, it has skyrocketed in popularity, thanks to its relative ease of use, powerful features, and speed. It has also spawned the wildly popular GitHub.com – a hosting service for Git repositories that boasts over 3.5 million users and 6 million repositories as of April 2013.

More and more these days, job descriptions are requesting that candidates include their GitHub usernames when they apply for jobs. Employers want to take a look at your GitHub account, see the contributions you have made to open source, and browse the code you have written. Don't believe me? Take a look at a few snippets from recent job postings on the next page.

These are just a handful of recent job postings that I found after a couple minutes of searching. I see job postings every single day that ask for GitHub usernames. It is important that you start making contributions to open source software (or creating projects of your own) and post them on sites like GitHub so that you'll have a portfolio of quality code to show employers when it's time to get a job. Top employers want to know that you can put your money where your mouth is and they will increasingly ask you to cough up your GitHub username so they can see what you're made of.

Enough said. Let's get setup.



Software Dev Engineer (API Engineer) Job

Position Description

Qualifications :

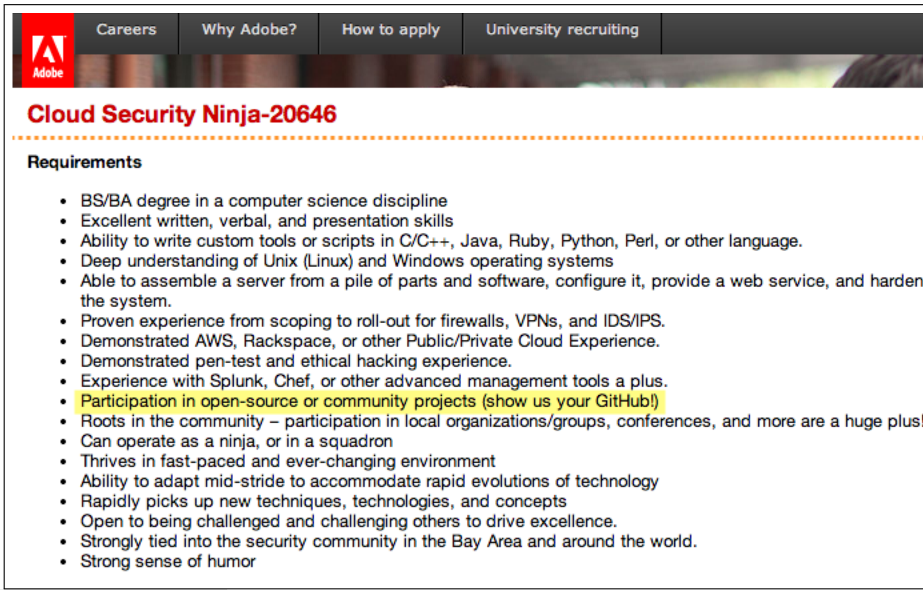
- Strong server side java development experience writing web or API platforms

- Strong experience writing web or network APIs in a scripting language (python, ruby, perl, etc.)

- Some experience with client side platforms (HTML/Javascript/iOS/Android), a plus

- Agile software development, continuous delivery practices (think 1-2 week production release cycles), and test driven development experience is useful

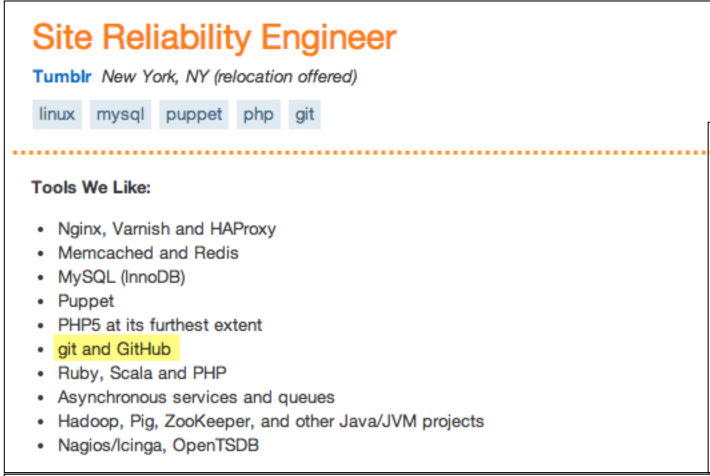
- Code samples from open source projects or github projects are great to send along



Cloud Security Ninja-20646

Requirements

- BS/BA degree in a computer science discipline
- Excellent written, verbal, and presentation skills
- Ability to write custom tools or scripts in C/C++, Java, Ruby, Python, Perl, or other language.
- Deep understanding of Unix (Linux) and Windows operating systems
- Able to assemble a server from a pile of parts and software, configure it, provide a web service, and harden the system.
- Proven experience from scoping to roll-out for firewalls, VPNs, and IDS/IPS.
- Demonstrated AWS, Rackspace, or other Public/Private Cloud Experience.
- Demonstrated pen-test and ethical hacking experience.
- Experience with Splunk, Chef, or other advanced management tools a plus.
- Participation in open-source or community projects (show us your GitHub!)
- Roots in the community – participation in local organizations/groups, conferences, and more are a huge plus!
- Can operate as a ninja, or in a squadron
- Thrives in fast-paced and ever-changing environment
- Ability to adapt mid-stride to accommodate rapid evolutions of technology
- Rapidly picks up new techniques, technologies, and concepts
- Open to being challenged and challenging others to drive excellence.
- Strongly tied into the security community in the Bay Area and around the world.
- Strong sense of humor

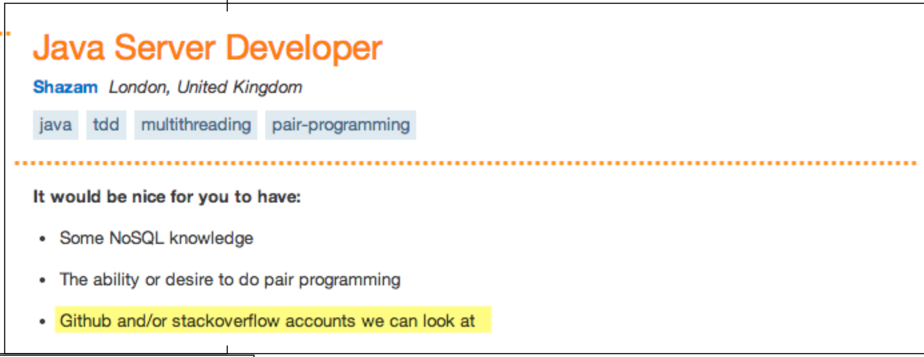


Site Reliability Engineer

Tumblr *New York, NY (relocation offered)*

Tools We Like:

- Nginx, Varnish and HAProxy
- Memcached and Redis
- MySQL (InnoDB)
- Puppet
- PHP5 at its furthest extent
- git and GitHub
- Ruby, Scala and PHP
- Asynchronous services and queues
- Hadoop, Pig, ZooKeeper, and other Java/JVM projects
- Nagios/Icinga, OpenTSDB

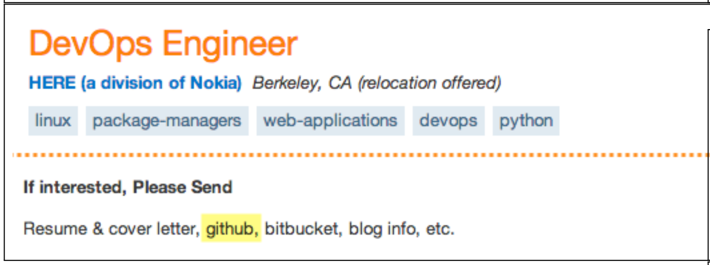


Java Server Developer

Shazam *London, United Kingdom*

It would be nice for you to have:

- Some NoSQL knowledge
- The ability or desire to do pair programming
- Github and/or stackoverflow accounts we can look at

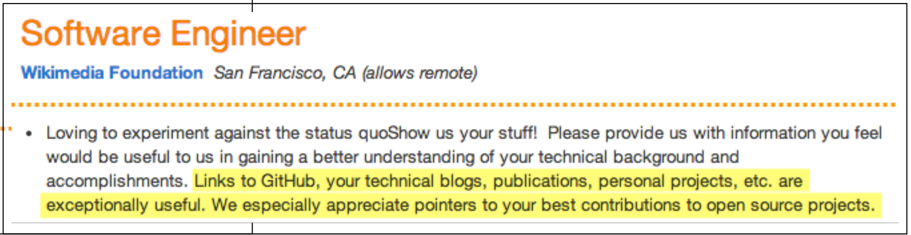


DevOps Engineer

HERE (a division of Nokia) *Berkeley, CA (relocation offered)*

If interested, Please Send

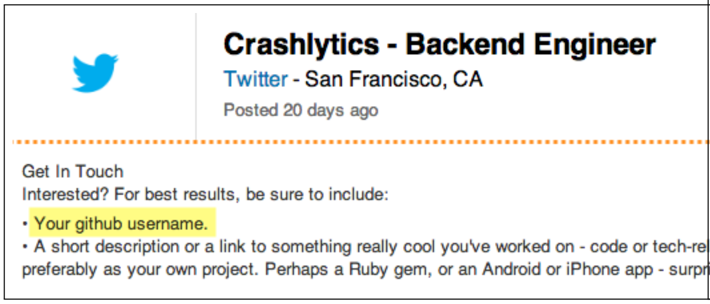
Resume & cover letter, github, bitbucket, blog info, etc.



Software Engineer

Wikimedia Foundation *San Francisco, CA (allows remote)*

- Loving to experiment against the status quoShow us your stuff! Please provide us with information you feel would be useful to us in gaining a better understanding of your technical background and accomplishments. Links to GitHub, your technical blogs, publications, personal projects, etc. are exceptionally useful. We especially appreciate pointers to your best contributions to open source projects.



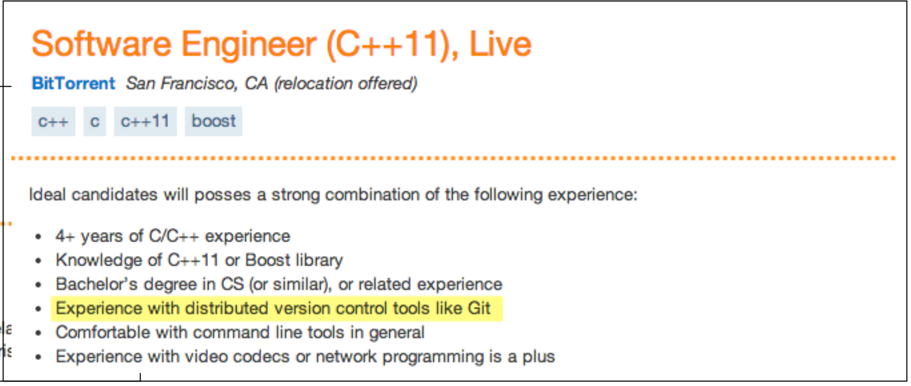
Crashlytics - Backend Engineer

Twitter - San Francisco, CA

Posted 20 days ago

Get In Touch
Interested? For best results, be sure to include:

- Your github username.
- A short description or a link to something really cool you've worked on - code or tech-rela preferably as your own project. Perhaps a Ruby gem, or an Android or iPhone app - surpris



Software Engineer (C++11), Live

BitTorrent *San Francisco, CA (relocation offered)*

Ideal candidates will posses a strong combination of the following experience:

- 4+ years of C/C++ experience
- Knowledge of C++11 or Boost library
- Bachelor's degree in CS (or similar), or related experience
- Experience with distributed version control tools like Git
- Comfortable with command line tools in general
- Experience with video codecs or network programming is a plus

1 Learning Objectives

In this lab, you will learn to:

- setup an account on GitHub.com
- commit and push changes to a remote repository
- pull changes from a remote repository
- clone a remote repository
- resolve merge conflicts, when they occur
- tag releases

2 Installing Git

To begin, you need to install Git on your system. This will differ depending on your operating system, and we will assume that you are capable of installing software on your system and making use of Google to help you solve any problems you may encounter with installation.

Note: There are graphical Git clients available. We will not support these and do not recommend that you use them. Your Git knowledge will be much more portable and you will learn it much more thoroughly if you start by learning the Git command line tools. Additionally, the exam will cover command line Git commands.

- **Windows users**: Download *Git for Windows* from http://msysgit.github.io
- **Linux users**: Consult your package repository. On Ubuntu, you'll want to install the package `git-core`
- **Mac users**: You may already have Git (type `git` at the command line and press Enter). If not, see the following link: https://help.github.com/articles/set-up-git#platform-mac

3 Getting Setup for the Lab

To complete this lab, you will need two separate directories:

Description	**Sample Path**
Main repository directory	`~/courses/cs2212/labs`
Clone repository directory	`~/courses/cs2212/clone`

Create these directories now. **Note:** It is very important that the main repository directory is not created within the clone repository directory, or vice versa. They should be two distinct paths in your file system.

```
$ mkdir -p ~/courses/cs2212/labs
$ mkdir -p ~/courses/cs2212/tmp
```

Recall that the tilde character (`~`) represents your home directory on a Linux, UNIX, or Mac system.

One other important note about these labs: **do not copy and paste commands**. There are specific characters (such as the tilde) that do not copy properly from the PDF to your terminal, and things will not work correctly. Also, you should be striving to learn the commands, as you will both need to use them for your project, and will be tested on them on the final exam. Copying and pasting does nothing to help you build your *muscle memory* as you do when you type in commands manually.

4 Generating an SSH Key Pair

GitHub has generously provided us with numerous free private repositories so that you can each submit your labs in a private GitHub repository (public repositories are always free on GitHub, but since labs are individual efforts, we need to use private repositories, which usually cost money). In addition, you will have access to a private team repository to host your team's project.

Only you, your TA, and your instructor have access to your individual repository. Before you can work with it, you need to setup an *SSH key*, which will allow you to authenticate with GitHub, and then setup an account on GitHub. Complete the following steps:

1. Open a terminal (on Windows, open the *Git for Windows* shell).
2. Enter the command `ssh-keygen -t dsa`. Press **Enter** at all prompts to accept all defaults.

   ```
   $ ssh-keygen -t dsa
   Generating public/private dsa key pair.
   Enter file in which to save the key (/Users/jeff/.ssh/id_dsa):
   Enter passphrase (empty for no passphrase):
   Enter same passphrase again:
   Your identification has been saved in /Users/jeff/.ssh/id_dsa.
   Your public key has been saved in /Users/jeff/.ssh/id_dsa.pub.
   The key fingerprint is:
   07:eb:ed:07:7a:92:dd:4e:f6:b6:6f:11:54:e5:7b:39 jeff@cs2212
   ```

   The command above generated a *public/private key pair*. Your public key is stored in `~/.ssh/id_dsa.pub`, while your private key is stored in `~/.ssh/id_dsa`. You will upload your public key to GitHub, and keep your private key secret. When you interact with GitHub from the command line, your private and public keys will be used to securely authenticate you, and you'll never have to enter a password!
3. Change to the `.ssh` subdirectory of your home directory and list its contents:

   ```
   cd ~/.ssh
   ls
   ```

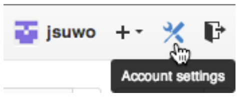
   You should see your public (`id_dsa.pub`) and private (`id_dsa`) keys in this directory.
4. You want to keep your private key secret. If anyone else were to obtain this key, they would be able to access your GitHub repository and steal your lab files. To ensure that it is locked down, enter the following command:

   ```
   chmod 600 id_dsa
   ```

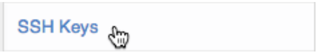
   This will give you read/write access to the file, and disallow anyone else access to it.
5. You now need to upload your public key to GitHub. Display the contents of your *public* key and copy everything in the file (from `ssh-dss` to `your_username@your_hostname`)

   ```
   cat id_dsa.pub
   ```

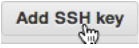
6. Open a web browser and go to GitHub.
7. Create a free account. If you already have one, you can reuse it. If not, please use your UWO username as your username.
8. Once logged in, click the **Account Settings** icon in the top right corner

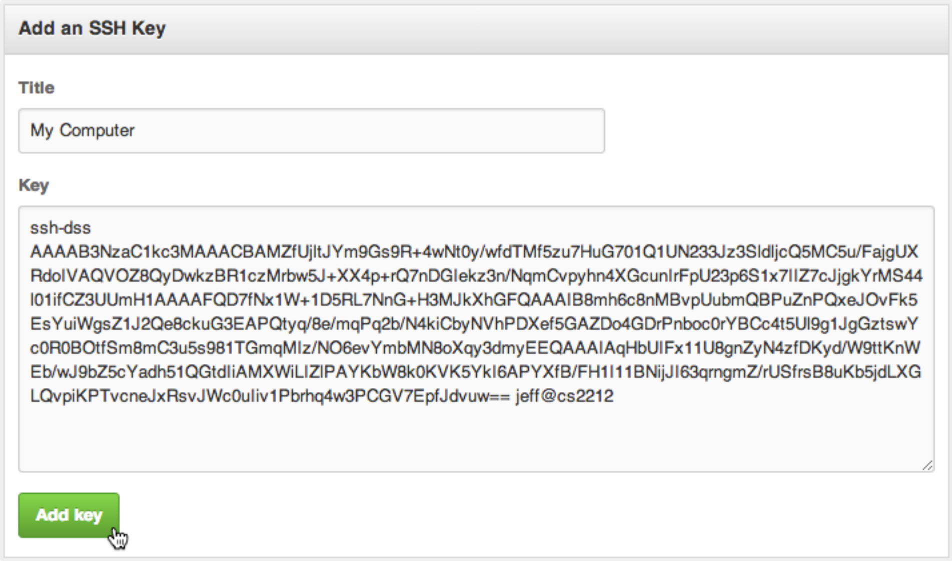
9. Click the **SSH Keys** menu item on the left

10. Click the **Add SSH key** button



11. In the **Title** field, enter a descriptive name like `My Computer`. In the **Key** field, paste your *public* key (the contents of `id_dsa.pub`). Be sure that everything from `ssh-dss` to `your_username@your_hostname` is pasted into this field


Add an SSH Key

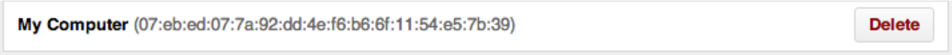
Title

My Computer

Key

```
ssh-dss
AAAAB3NzaC1kc3MAAACBAMZfUjltJYm9Gs9R+4wNt0y/wfdTMf5zu7HuG701Q1UN233Jz3SldljcQ5MC5u/FajgUX
RdolVAQVOZ8QyDwkzBR1czMrbw5J+XX4p+rQ7nDGlekz3n/NqmCvpyhn4XGcunIrFpU23p6S1x7IIZ7cJjgkYrMS44
l01ifCZ3UUmH1AAAAFQD7fNx1W+1D5RL7NnG+H3MJkXhGFQAAAIB8mh6c8nMBvpUubmQBPuZnPQxeJOvFk5
EsYuiWgsZ1J2Qe8ckuG3EAPQtyq/8e/mqPq2b/N4kiCbyNVhPDXef5GAZDo4GDrPnboc0rYBCc4t5Ul9g1JgGztswY
c0R0BOtfSm8mC3u5s981TGmqMIz/NO6evYmbMN8oXqy3dmyEEQAAAIAqHbUIFx11U8gnZyN4zfDKyd/W9ttKnW
Eb/wJ9bZ5cYadh51QGtdliAMXWiLIZlPAYKbW8k0KVK5YkI6APYXfB/FH1I11BNijJI63qrngmZ/rUSfrsB8uKb5jdLXG
LQvpiKPTvcneJxRsvJWc0uliv1Pbrhq4w3PCGV7EpfJdvuw== jeff@cs2212
```

12. Click the **Add key** button. Confirm your password if prompted. You should now see your key in the list of your SSH public keys


My Computer (07:eb:ed:07:7a:92:dd:4e:f6:b6:6f:11:54:e5:7b:39)

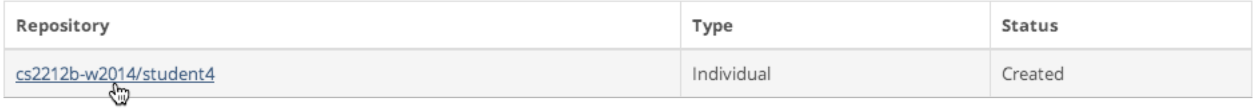
5 Creating Your Repository

To create your private individual repository, you will need to log in to the private course site.

1. Visit http://www.cs2212.ca and sign in with your GitHub account
2. Click the **GitHub Repositories** link in the sidebar

3. Click the **Create Individual Repository** button to create your repository. This may take up to 20 minutes. You will receive an email when your repository has been created (check your junk mail folder if you do not see it).

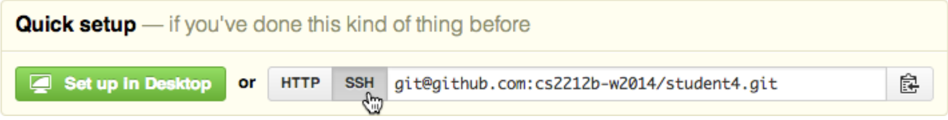
4. Click the link to your GitHub repository in the email you receive. Alternatively, refresh the **GitHub Repositories** page on the private course site and click the link you see there.



Repository	Type	Status
cs2212b-w2014/student4	Individual	Created

Note: Make sure that you click the link for your **individual** repository, and not your team repository.

5. In the **Quick setup** box, click the **SSH** button.



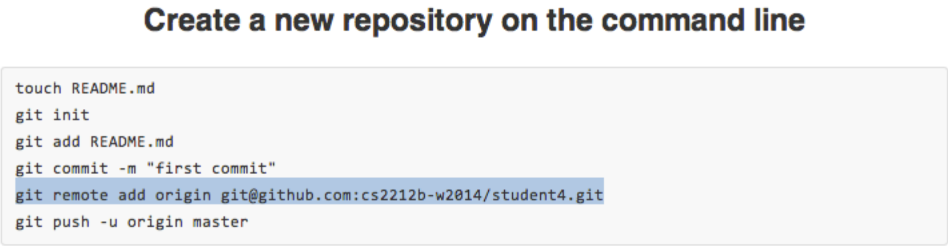
Quick setup — if you've done this kind of thing before

```
git@github.com:cs2212b-w2014/student4.git
```

6. No, seriously, click the **SSH** button. People never do, and then wonder why the instructions don't work for them.

7. Did you click the **SSH** button?

8. Notice that a few instructions are given on your repository page. Do not worry about these, we will take care of them in a few moments. For now, find the line under the **Create a new repository on the command line** heading that reads `git remote add ...`. Copy this line. We'll need it in a few minutes.



Create a new repository on the command line

```
touch README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin git@github.com:cs2212b-w2014/student4.git
git push -u origin master
```

6 Setting Up Your Repository

1. Switch back to your terminal and enter the following commands to configure Git. You should, of course, replace my name with yours, and my email address with your UWO email address. Be sure to use your UWO address and not another from a different provider:

```
git config --global user.name "Jeff Shantz"
git config --global user.email jshantz4@uwo.ca
```

This creates a configuration file `~/.gitconfig` that contains your *global* Git settings (that is, settings that will be used for all your Git repositories on your computer):

```
$ cat ~/.gitconfig
[user]
  name = Jeff Shantz
  email = jshantz4@uwo.ca
```

2. Change to the **main** repository directory that you created in Section 3:

```
cd ~/courses/cs2212/labs
```

3. Lock down your repository by entering the following command:

```
chmod 700 .
```

Note that there is a space and a period (`.`) after `700`. This gives you access to read from, write to, and change to the current directory, but locks everyone else out.

4. Initialize the repository and list the contents of the directory (include the `-a` flag to see all hidden files):

   ```
   git init
   ls -la
   ```

   Observe that a directory `.git` was created. This contains metadata and configuration used by Git, and tells Git that the current directory is a Git repository. Do *not* alter the contents of the `.git` directory or delete it.

5. Create a directory `lab1` within your repository but **do not** change to it:

   ```
   mkdir lab1
   ```

6. Create an empty file `lab1/Lab1.java`:

   ```
   touch lab1/Lab1.java
   ```

7. Type `git status`:

   ```
   # On branch master
   #
   # Initial commit
   #
   # Untracked files:
   #   (use "git add <file>..." to include in what will be committed)
   #
   # lab1/
   nothing added to commit but untracked files present (use "git add" to track)
   ```

   The `git status` command tells us about the current state of the repository. Here, Git is telling us that there is a directory `lab1` that has been created, but is not yet part of the repository – that is, it is *untracked*.

8. Type `git add lab1` and then `git status`:

   ```
   # On branch master
   #
   # Initial commit
   #
   # Changes to be committed:
   #   (use "git rm --cached <file>..." to unstage)
   #
   # new file:   lab1/Lab1.java
   #
   ```

   Here, we have told Git to add everything in the `lab1` directory to the repository. However, notice that our file `Lab1.java` is listed under **Changes to be committed**. This means that the file is *staged* in preparation to be committed to the repository, but it has not yet been committed to the repository. Git allows us to stage multiple changes (i.e. group together multiple changes in a batch) before we finally commit that set of changes to the repository. If we decided we didn't want to add `Lab1.java` to the repository after all, we could *unstage* it by typing `git rm --cached lab1/Lab1.java`. Don't do this, though.

9. Change to the `lab1` directory

10. Edit the file `Lab1.java` and add the necessary class and main method to print the following:

    ```
    Hello Git World
    My username is jshantz4
    ```

    Your output should appear **exactly** as above, except that you should use your UWO username instead of mine.

11. Save the file and exit your editor

12. Compile your code and run it to ensure it works:

    ```
    $ javac Lab1.java
    $ java Lab1
    Hello Git World
    My username is jshantz4
    ```

13. Type `git status`:

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
# new file:   Lab1.java
#
# 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:   Lab1.java
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# Lab1.class
```

Notice a few things here:

- `Lab1.java` is listed under **Changes to be committed**. This is because we used `git add` earlier to *stage* the empty `Lab1.java` file
- `Lab1.java` also appears under **Changes not staged for commit**. This is because we have modified it, but have not staged those changes yet.
- `Lab1.class` appears under **Untracked files** because we have not added this file to our repository with `git add`

14. Type `git add .` and then type `git status`. Specifying `.` as an argument to `git add` stages all changes in the current directory and all subdirectories:

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
# new file:   Lab1.class
# new file:   Lab1.java
#
```

15. Commit your changes by typing `git commit`. Git will open Vim, and you'll need to type a *commit message*. Your commit message is recorded by Git and tells other developers what changes you have made in the code you are committing. It should be brief, but should accurately summarize what you've changed. For now, just type `Initial commit` on the first line, and leave the rest as is:

```
Initial commit

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
# new file:   Lab1.class
# new file:   Lab1.java
#
```

Recall that, to insert text in Vim, you'll need to press `i` to enter *insert mode*. Note that you'll have to type *something* in the commit message. Otherwise, Git will abort the commit.

16. Press **Esc** to return to normal mode and type `:wq` to commit your code. If all went according to plan, you should see a message similar to the following:

```
[master (root-commit) 5d8be8d] Initial commit
 2 files changed, 11 insertions(+)
 create mode 100644 lab1/Lab1.class
 create mode 100644 lab1/Lab1.java
```

17. Your code has now been committed, but has not yet been *pushed* to GitHub. Before we can push your code to GitHub, we first have to connect this local repository with your private repository on GitHub. To do this, you'll need the `git remote add ...` line that you copied in Step 8 of Section 4. Paste that command at the terminal:

    ```
    git remote add origin git@github.com:cs2212b-w2014/student4.git
    ```

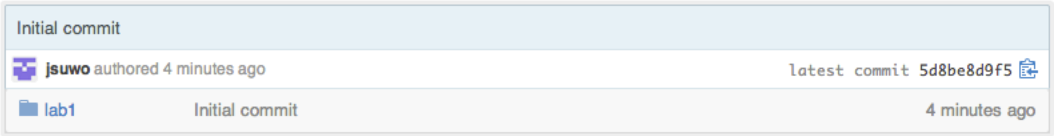
    Note: you need to use the line you copied in Step 8 of Section 4. **Do not** use the line shown here – it is merely shown as an example of what you would type.

18. We're finally ready to push our changes to GitHub. Enter the following command:

    ```
    $ git push -u origin master
    Counting objects: 5, done.
    Delta compression using up to 4 threads.
    Compressing objects: 100% (4/4), done.
    Writing objects: 100% (5/5), 774 bytes | 0 bytes/s, done.
    Total 5 (delta 0), reused 0 (delta 0)
    To git@gh-jsuwo:cs2212b-w2014/student4.git
     * [new branch]      master -> master
    Branch master set up to track remote branch master from origin.
    ```

    This told Git to push your *master* branch (the current branch that we're on in the repository) to your private GitHub repository. Branches are beyond the scope of this tutorial, but, when we create a new repository, we always start out on the master branch.

19. Open your repository page again on GitHub. Instead of a list of instructions, you will now see the file that you committed, along with your commit log message:

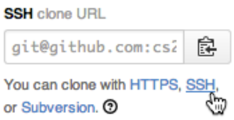


Initial commit

jsuwo authored 4 minutes ago latest commit 5d8be8d9f5

lab1 Initial commit 4 minutes ago

20. On your repository page, in the right sidebar, click the **SSH** link and copy the **SSH clone URL** for your repository. You'll need it in the next section.



SSH clone URL

git@github.com:cs2

You can clone with HTTPS, SSH, or Subversion.

Part II
Cloning, Pushing, and Pulling

Estimated Time: 5 minutes

1. Change to the **clone** repository directory that you created in Section 3:

   ```
   $ cd ~/courses/cs2212/clone
   ```

2. *Clone* your repository using `git clone`, along with the address of your repository that you copied in Step 20 of the last section:

   ```
   $ git clone git@github.com:cs2212b-w2014/student4.git
   Cloning into ’student4’...
   remote: Counting objects: 5, done.
   remote: Compressing objects: 100% (4/4), done.
   remote: Total 5 (delta 0), reused 5 (delta 0)
   Receiving objects: 100% (5/5), done.
   Checking connectivity... done
   ```

   Cloning a repository makes a new local copy of the repository on your system. For instance, you could create a repository on one computer, and clone it on another so that you could work on your code on both systems.

   Additionally, if multiple people had access to the repository, each could clone it from GitHub and work on the code within it on their local systems. When they were finished making changes, they would *push* their changes back to the GitHub repository, and others could then *pull* those changes into their local repositories.

3. List the contents of the current directory. You should see a new directory with the same name as your UWO username. This is the repository that you have just cloned. Change to this directory.

4. List the contents of the current directory and observe that the directory `lab1` is present, since it belongs to the repository. Change to the `lab1` directory and change `Lab1.java` to print the following output:

   ```
   Hello Git World
   My username is jshantz4
   My name is Jeff Shantz
   ```

   Of course, it should print your username and name, instead of mine.

5. Type `git status`:

   ```
   $ git status
   # On branch master
   # Changes not staged for commit:
   #   (use "git add <file>..." to update what will be committed)
   #   (use "git checkout -- <file>..." to discard changes in working directory)
   #
   # modified:   Lab1.java
   #
   no changes added to commit (use "git add" and/or "git commit -a")
   ```

   Observe that Git tells us that `Lab1.java` has been modified, but is not staged for commit. In other words, the changes we have made to `Lab1.java` are *unstaged*.

6. Type `git add Lab1.java` to stage your changes, and then type `git status`:

   ```
   $ git add Lab1.java
   $ git status
   # On branch master
   # Changes to be committed:
   #   (use "git reset HEAD <file>..." to unstage)
   #
   # modified:   Lab1.java
   #
   ```

Observe that Git now tells us that there are changes to be committed. That is, we have now *staged* our changes, but have not yet committed them.

7. Type `git commit` and enter the commit message `Modified Lab1.java`. Type `:wq` to save your commit log and exit Vim. If all went swimmingly, you should see:

   ```
   [master 733d224] Modified Lab1.java
    1 file changed, 1 insertion(+)
   ```

8. Our changes have now been *committed*, but have not yet been *pushed* to GitHub. Enter `git push` to do this:

   ```
   $ git push
   Counting objects: 7, done.
   Delta compression using up to 4 threads.
   Compressing objects: 100% (3/3), done.
   Writing objects: 100% (4/4), 384 bytes | 0 bytes/s, done.
   Total 4 (delta 1), reused 0 (delta 0)
   To git@gh-jsuwo:cs2212b-w2014/student4.git
      5d8be8d..733d224  master -> master
   ```

9. Change back to the `lab1` directory in your **main** repository directory and display the contents of the file `Lab1.java`:

   ```
   $ cd ~/courses/cs2212/labs/lab1
   $ cat Lab1.java
   ```

   Observe that it has not yet been updated. This is because we need to *pull* the changes into this copy of the repository.

10. Type `git pull` to pull all new changes into the repository:

    ```
    $ git pull
    remote: Counting objects: 7, done.
    remote: Compressing objects: 100% (2/2), done.
    remote: Total 4 (delta 1), reused 4 (delta 1)
    Unpacking objects: 100% (4/4), done.
    From gh-jsuwo:cs2212b-w2014/student4
       5d8be8d..733d224  master     -> origin/master
    Updating 5d8be8d..733d224
    Fast-forward
     lab1/Lab1.java | 1 +
     1 file changed, 1 insertion(+)
    ```

11. Display the contents of `Lab1.java`. Observe that it is now updated:

    ```
    $ cat Lab1.java
    ```

    Look at that! You just learned how to *clone* a Git repository, *push* changes to it, and *pull* the most recent changes down from GitHub into your local copy.

    One thing you might be wondering about is that when you initially pushed your changes to GitHub, you had to type `git push -u origin master`. The second time, you only had to type `git push`.

    The reason is that, when we first pushed our changes, we had to tell Git to push the master branch up to GitHub. This uploaded the master branch to GitHub, establishing a link between the master branch in your local copy of the repository, and the master branch in the GitHub repository.

    In all subsequent pushes, since the master branch has already been linked to GitHub, we simply have to type `git push` to push all changes on the master branch in our local repository to the master branch on GitHub.

Part III

Resolving Merge Conflicts

Estimated Time: 10 minutes

Suppose two developers make a different change to the same line in the same file, each in their own local copies of a project repository. They both attempt to push their changes to GitHub, but whose change should be accepted, and whose change should be discarded? After all, they both edited the same line in the same file. This scenario is known as a *merge conflict*, and you will inevitably encounter such conflicts when you are working with Git in a team environment. Knowing how to deal with merge conflicts is, therefore, an essential skill.

7 Setting Up the Conflict

1. Change to the `lab1` directory in your **main** repository directory:

   ```
   $ cd ~/courses/cs2212/labs/lab1
   ```

2. Edit `Lab1.java` and change the line that prints your username so that it prints your email address instead. The output should appear as follows:

   ```
   Hello Git World
   My email address is jshantz4@uwo.ca
   My name is Jeff Shantz
   ```

   Your output should appear **exactly** as above, except that you should use your UWO email address and name instead of mine.

3. Stage, commit, and push your changes to GitHub:

   ```
   $ git add .
   $ git commit -m "Prints email address"
   $ git push
   ```

   Notice that we use the `-m` argument with `git commit` to specify our commit log message on the command line.

4. Now change to the `lab1` directory in the **clone** repository directory you cloned earlier:

   ```
   $ cd ~/courses/cs2212/clone/student4/lab1
   ```

5. **<u>Without</u> pulling changes from the remote repository**, edit `Lab1.java` and change the line that prints your username so that it prints your GAUL username instead. The output should appear as follows:

   ```
   Hello Git World
   My GAUL username is jshantz4
   My name is Jeff Shantz
   ```

   Your output should appear **exactly** as above, except that you should use your GAUL username and name instead of mine.

6. Stage and commit your changes:

   ```
   $ git add .
   $ git commit -m "Prints GAUL username"
   ```

7. Try pushing your changes to GitHub:

   ```
   $ git push
   To git@gh-jsuwo:cs2212b-w2014/student4.git
    ! [rejected]        master -> master (fetch first)
   error: failed to push some refs to ’git@gh-jsuwo:cs2212b-w2014/student4.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 merge the remote changes (e.g.,
   hint: ’git pull’) before pushing again.
   hint: See the ’Note about fast-forwards’ in ’git push --help’ for details.
   ```

The push fails because Git recognizes that there are changes on the server that we have not yet pulled into our local copy of the repository.

8. Pull the changes from GitHub:

```
$ git pull
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 1), reused 4 (delta 1)
Unpacking objects: 100% (4/4), done.
From gh-jsuwo:cs2212b-w2014/student4
   733d224..991c106  master     -> origin/master
Auto-merging lab1/Lab1.java
CONFLICT (content): Merge conflict in lab1/Lab1.java
Automatic merge failed; fix conflicts and then commit the result.
```

   Observe that Git warns us of a merge conflict. It notifies us that we must fix the conflicts and commit the result.

9. Edit the file `Lab1.java`. You should see something similar to the following:

```
<<<<<<< HEAD
    System.out.println("My GAUL username is jshantz4");
=======
    System.out.println("My email address is jshantz4@uwo.ca");
>>>>>>> 991c106af830751c105147e4946286280640281
```

   This is a *merge conflict*. The area between `<<<<<<< HEAD` and `=======` is the change you made in the current copy of the repository. The area following that (up to `>>>>>>>`) is the current line in the remote repository on GitHub.

   We now need to tell Git which line we want to keep. We do this by simply deleting the lines we do not want.

10. Edit the file so that it prints your GAUL username. Hence, you'll need to remove the `<<<<<<< HEAD` line, the `=======` line, the line that prints your email address, and the `>>>>>>>`) line. In the end, your output should appear as follows:

```
Hello Git World
My GAUL username is jshantz4
My name is Jeff Shantz
```

11. Type `git status`:

```
# On branch master
# Your branch and 'origin/master' have diverged,
# and have 1 and 1 different commit each, respectively.
#   (use "git pull" to merge the remote branch into yours)
#
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
# both modified:      Lab1.java
#
no changes added to commit (use "git add" and/or "git commit -a")
```

   Notice that Git is telling us we have *unmerged paths* – that is, we have not yet committed our resolution to the merge conflict.

12. Stage your changes and check `git status`:

```
$ git add .
$ git status
# On branch master
# Your branch and 'origin/master' have diverged,
# and have 1 and 1 different commit each, respectively.
#   (use "git pull" to merge the remote branch into yours)
#
# All conflicts fixed but you are still merging.
#   (use "git commit" to conclude merge)
#
nothing to commit, working directory clean
```

Observe that Git now recognizes that we have fixed all conflicts, and it tells us to use `git commit` to finish off the merging process.

13. Commit your changes:

```
$ git commit
Merge branch ’master’ of gh-jsuwo:cs2212b-w2014/student4

Conflicts:
  lab1/Lab1.java
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
# .git/MERGE_HEAD
# and try again.
```

    Observe that Git has filled in the commit log message for us, indicating which merge conflicts are resolved by this commit. Save and exit to commit the changes (use `:wq` in Vim).

14. Push your changes to GitHub:

```
$ git push
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 598 bytes | 0 bytes/s, done.
Total 5 (delta 1), reused 0 (delta 0)
To git@gh-jsuwo:cs2212b-w2014/student4.git
   991c106..b72fb00  master -> master
```

    Notice that the push now works correctly. We’ve resolved the conflict!

Part IV

Ignoring Files

Estimated Time: 5 minutes

Earlier, we committed both **Lab1.java** and **Lab1.class** to our repository. In general, we typically avoid committing binary files (such as a **.class** file) or any other files that can be easily regenerated (for instance, through compilation). This is to make it faster for people to pull our changes. The larger the files in our repository, the slower it will be to transfer them to our systems.

Git allows us to create a file `.gitignore` in our repository to specify file patterns that we do not wish to include when committing files to the repository.

1. Change to the root of the **main** repository directory (this directory should contain the `lab1` directory):

   ```
   $ cd ~/courses/cs2212/labs
   ```

2. Create an empty `.class` file `Hello.class`:

   ```
   $ touch Hello.class
   ```

3. Type `git status`. Notice that `Hello.class` is listed as an untracked file:

   ```
   $ git status
   # On branch master
   # Untracked files:
   #   (use "git add <file>..." to include in what will be committed)
   #
   # Hello.class
   nothing added to commit but untracked files present (use "git add" to track)
   ```

   If we were to type `git add .`, `Hello.class` would be staged, which we do not want.

4. Create a new file `.gitignore` with the following contents:

   ```
   *.class
   ```

5. Type `git status`:

   ```
   $ git status
   # On branch master
   # Untracked files:
   #   (use "git add <file>..." to include in what will be committed)
   #
   # .gitignore
   nothing added to commit but untracked files present (use "git add" to track)
   ```

   Notice that `Hello.class` is no longer listed as an untracked file – it is being ignored. However, we still have to commit our `.gitignore` file to the repository.

6. Stage, commit, and push your changes:

   ```
   $ git add .
   $ git commit -m "Adds .gitignore"
   $ git push
   ```

   Now, we've prevented `.class` files from being committed to our repository in the future, but what about the existing `Lab1.class` that we committed? We need to remove this.

7. Remove the file with the `git rm` command:

   ```
   $ git rm lab1/Lab1.class
   ```

8. Type `git status`. Notice that the file is staged to be deleted from the repository:

   ```
   $ git status
   # On branch master
   # Changes to be committed:
   #   (use "git reset HEAD <file>..." to unstage)
   #
   # deleted:    lab1/Lab1.class
   #
   ```

9. Commit and push your changes to remove the file from the repository on GitHub:

   ```
   $ git commit -m "Removes Lab1.class"
   $ git push
   ```

You are expected to keep your repositories tidy. Binary files and other useless files (like `Desktop.ini` on Windows, or `.DS_Store` on Mac) should not be present in your repositories. Instead, `.gitignore` should be used to keep them out. A project repository with all sorts of extraneous files will lose your team marks on your final submission.

Part V

Introducing Commits and Tags

Estimated Time: 10 minutes

You can (and should) commit and push to GitHub as often as you like. If you want to see a history of your commits, use the `git log` command. For example, issuing this command after making four commits produces output similar to the following:

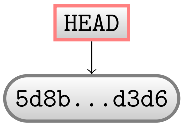
```
$ git log --pretty=oneline
e27a1e859d17becfaabba5447a30d5a11c18b0b6 Prints GAUL username
991c106af830751c105147e49462862806402881 Prints email address
733d2245bea446fd2e37f95507308d97fc5f5090 Modified Lab1.java
5d8be8d9f5d819eccd9bedad071a43bce822d3d6 Initial commit
```

Observe that each commit message is displayed, along with a 40-byte hexadecimal hash which represents that commit. You can use this hash to reference a commit. For instance, if you wanted to see what changes were made in the commit with the log message `Prints GAUL username`, you would use `git show`, specifying the hash representing that commit[1]:

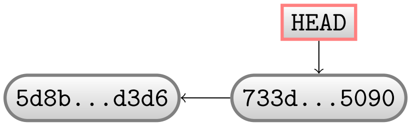
```
$ git show e27a
commit e27a1e859d17becfaabba5447a30d5a11c18b0b6
Author: Jeff Shantz <jeff@csd.uwo.ca>
Date:   Sun Jan 26 18:44:53 2014 -0500

    Prints GAUL username

diff --git a/lab1/Lab1.java b/lab1/Lab1.java
index 2903c53..97dd363 100644
--- a/lab1/Lab1.java
+++ b/lab1/Lab1.java
@@ -3,7 +3,7 @@ public class Lab1 {
   public static void main(String[] args) throws Exception {

     System.out.println("Hello Git World");
-    System.out.println("My username is jshantz4");
+    System.out.println("My GAUL username is jshantz4");
     System.out.println("My name is Jeff Shantz");

   }
```

Here, Git is telling us that some guy named Jeff Shantz edited the file `Lab1.java`. He removed the `My username is jshantz4` line and added the `My GAUL username is jshantz4` line. A `-` at the start of a line indicates that the line was removed as of that commit, while a `+` indicates that the line is new as of that commit.

Each time you commit to a Git repository, you can think of Git as adding a new node to a linked list of commit nodes. For example, when we performed our initial commit (5d8b...d3d6), a new node was created to represent that commit:

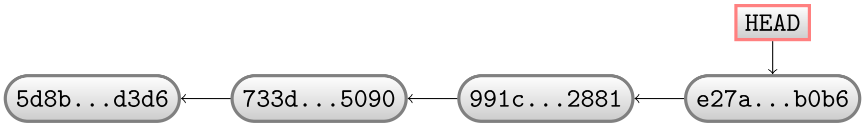


After the next commit (8288...d1e2), an additional node was added:



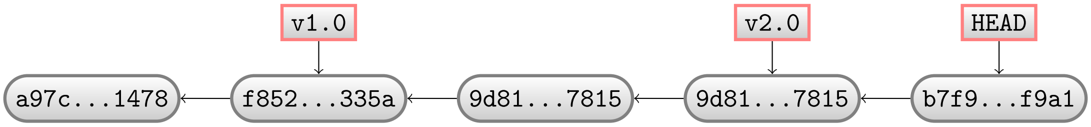
After our fourth commit, the commit chain would look as follows:

[1] Actually, you don't have to type the entire 40 byte hash. You'll have to at least specify the first 4 characters of the hash. If there are other commits whose hashes begin with the same 4 characters, you'll have to specify a few more characters until the specified hash is unambiguous. In the case above, we could simply type `git show e27a`.



Observe that each time we commit our changes in Git, not only does a new node get added to the chain, but the *tag* `HEAD` is moved to point to our latest commit. We use tags to refer to specific commits. Instead of typing `git show e27a`, we could simply type `git show HEAD`. Git automatically creates the tag `HEAD` for us, but we can also create our own tags.

Tags are generally used to represent specific milestones. For example, when we commit our final changes before releasing a new version of our software, we might tag the commit with the version number of the software being released. In the figure below, you can see that two tags have been created: one to represent version 1 of the program, and another to represent version 2:



As we'll see in the next section, when you're ready to commit your changes and submit your lab, you'll commit and then create a tag that specifies that the commit is your lab submission. For example, if you were submitting this lab (lab 1), you would commit your changes, and then tag the commit with the tag `lab1` to indicate to the automarker that this commit represents your lab 1 submission. Without tags, we would have no way of knowing which commit represents your final submission, since you can commit as often as you like!

8 Submitting Your Labs

Phew. All that just to submit a lab. Well, it's actually a pretty simple process, but it required a bit of background knowledge on Git before we could get to this point.

Recall that to commit changes to your repository, you must first *stage* the changes using `git add`, commit them using `git commit`, and push them to your repository on GitHub using `git push`:

```
cd ~/courses/cs2212/labs/lab1
git status
git add .
git commit
git push
```

Once you have pushed your final commit to GitHub and are ready to submit your lab, you will *tag* it with the name of the lab. As with your lab directory, the tag **must be named** `labX`, where `X` is the lab number. For instance, to submit lab 1, you would type:

```
$ git tag -a lab1
```

You will be asked to enter a commit message. Enter something like `Lab 1 submission`, and save and exit Vim. This creates a tag `lab1` that points to your most recent commit. You can see a list of tags that you've created by typing `git tag`:

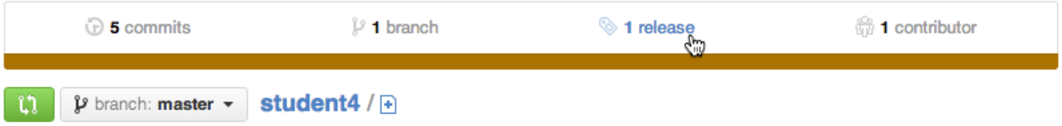
```
$ git tag
lab1
```

As you can see, your `lab1` tag has been created, but has not yet been pushed to GitHub, so the automarker will not yet know about it. Therefore, you must tell Git to push the tag to the remote repository on GitHub by passing the `--tags` argument to `git push`:

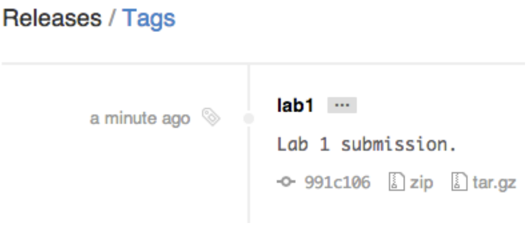
```
$ git push --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 165 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:cs2212b-w2014/student4.git
 * [new tag]         lab1 -> lab1
```

Observe that Git tells us it has pushed a new tag `lab1` to the remote repository.

Next, open your repository page on GitHub and notice that a *release* has been created:



If you click this link, you'll see your tag listed. If you click either the `zip` or `tar.gz` links below the tag, you will receive a compressed copy of the repository in the exact state it was in at the time the tag was created.



Lab 1 submission.

Congratulations! You've submitted your lab!

Part VI

Resubmitting a Lab

Estimated Time: 2 minutes

With the exception of the `HEAD` tag that Git automatically maintains for us, when we create a tag, it is generally fixed to a specific commit and never moves. This allows us to “go back in time" and return our repository to the state it was in at a particular commit. For instance, we could type:

```
git checkout lab1
```

to travel back in time and work with our repository in the state it was in at the time we created the tag `lab1`. To travel back to the present, we would simply type:

```
git checkout master
```

Now, tags are neat, but what if we need to resubmit a lab? Suppose we’ve tagged a commit as `lab1`, but later realize we need to make some changes and resubmit? How do we move a tag to a new commit?

The answer is that you cannot. Tags are static and point to a specific commit. In order to “move" a tag to point to a more recent commit, you have to delete the tag, retag the latest commit, and push the tag to GitHub again. If you were doing this for lab 1, you would do the following:

```
git add .                          # Stage the changes representing your
                                   # modified submission
git commit                         # Commit your changes
git push                           # Push them to the server
git tag -d lab1                    # Delete the ’lab1’ tag
git push origin :refs/tags/lab1    # Tell GitHub to delete it, too
git tag -a lab1                    # Create the ’lab1’ tag again, pointing to ur
                                   # most recent commit
git push --tags                    # Push it to GitHub
```

Part VII
What's Next?

Git is extremely powerful and actually pretty easy to use. We've only scratched the surface here, so if you want to learn more about it (and you should, as discussed earlier), then I highly recommend picking up the excellent book from O'Reilly Media, *Version Control with Git*, 2nd edition. Check it out at http://www.amazon.ca/Version-Control-Git-collaborative-development/dp/1449316387.