Git Basics

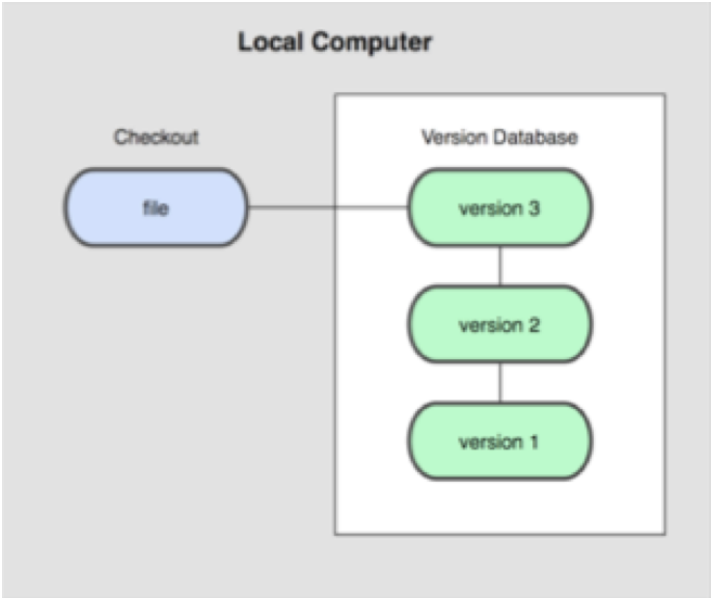
Christopher Simpkins
chris.simpkins@gatech.edu

Version Control Systems

- Records changes to files over time
- Allows you to recover older versions of a file (effectively “undoing” recent chagnes)
- Many people already do this
  - manually by saving copies of files with version information embedded in the file name, or
  - automatically using a commercial solutions such as Apple’s Time Machine.

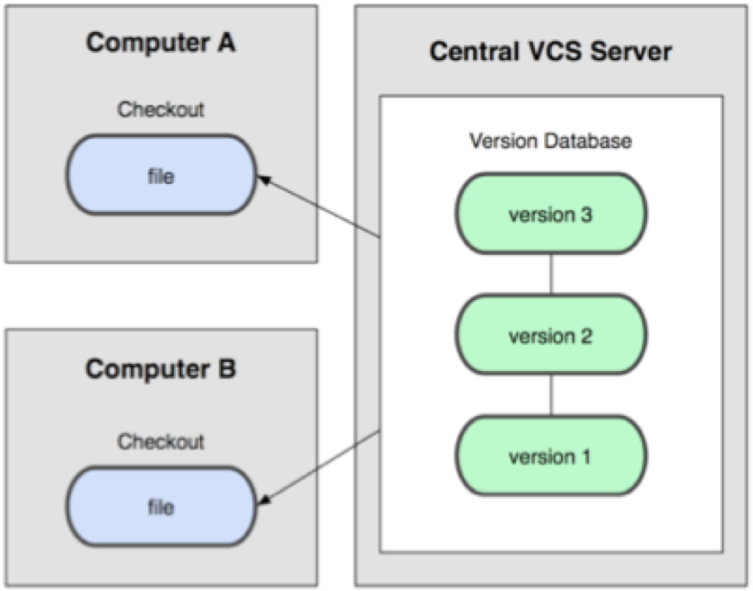
Neither of these approaches is sufficiently powerful for professional software development.

Local Version Control Systems



- RCS - Revision Control System
- Stores file versions as diffs
  - Any version of the file is constructed by applying each of the diffs in order from the original file.
- Can be used by a team if team has access to a central machine (over ssh, for example), but very clumsy.

Centralized Version Control Systems

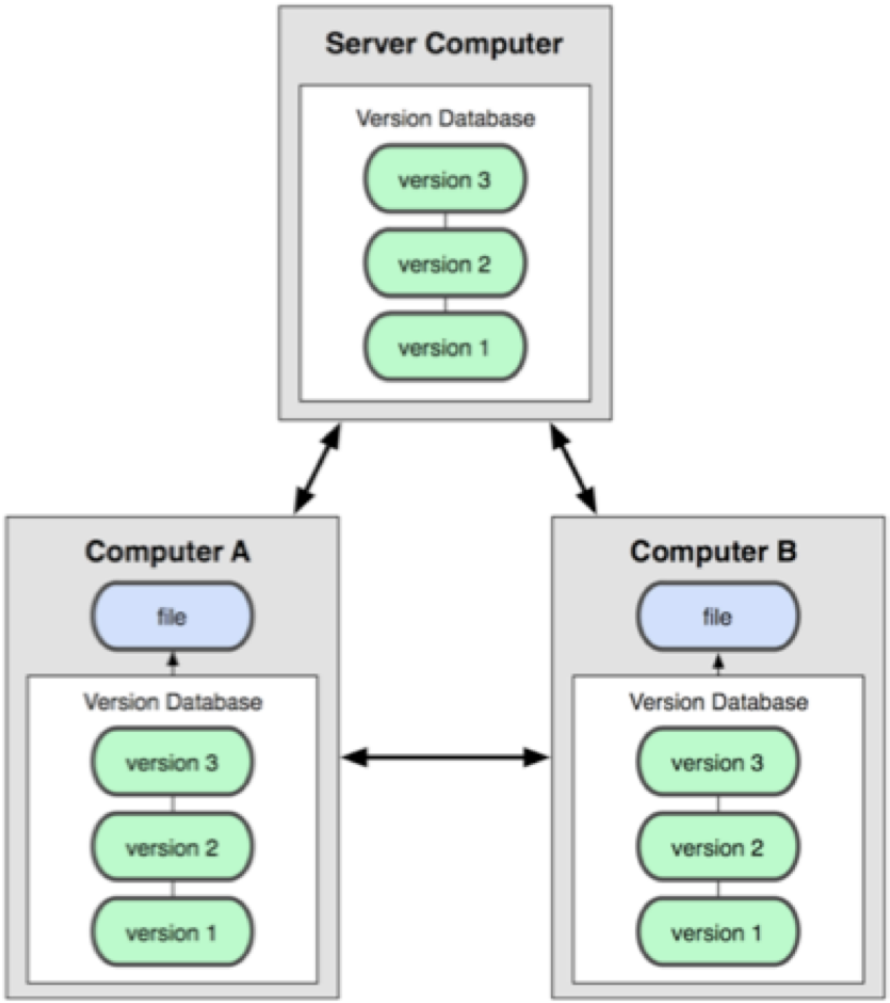


- Central server contains the version database
- Programmers “check out” the project on their local computers
- Each programmer is dependent on central server
- Most popular VCS architecture (CVS, SVN, MKS, Perforce, StarTeam, etc.)

Locking Models

- Pessimistic locking (most commercial VCSes)
  - When one programmer has a file “checked out” no one else can edit it.
- Optimistic locking (Most open-source VCSes - CVS, SVN, Git, Hg)
  - Files aren’t locked.
  - If two people edit a file, the first committer wins - the second person must resolve conflicts before committing their changes
- Optimistic locking generally preferred, but ...
  - Centralized VCSes can encourage infrequent commits to delay the hassle of conflict resolution.
  - Distributed VCSes eliminate this problem by decoupling commits from merges (“pushes” in Git parlance).

Distributed Verson Control Systems



- Each programmer has a local version of the repository
- Every repository is “equal” - no central authoritative repsitory (by default, at least)
- Each repository contains the entire project and all of its history
- Commits are made to the local repository
- Repositories are updated with changes from other repositories by merging the repositories

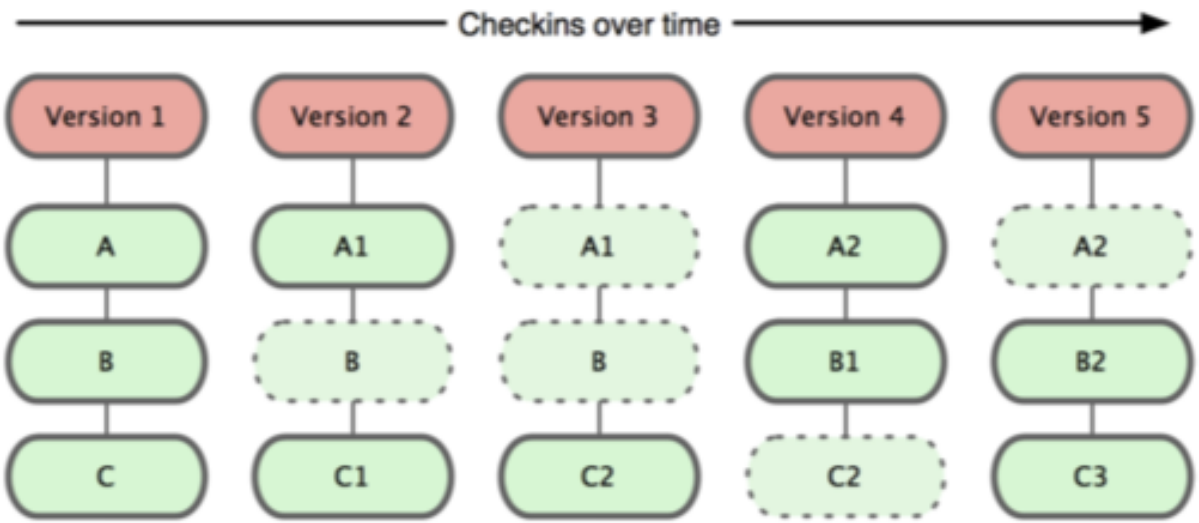
Git

- Created in 2005 by Linus Torvalds to manage the Linux Kernel Super fast and proven in large open source projects
- Probably the leading DVCS in use today
- We'll use Git, and the GitHub hosting service in this class

Many prospective employers ask to see your GitHub projects. After this class, you'll have something to show them.

Git Concepts

- Git stores snapshots of the entire project, not file diffs.
- Every commit contains references to each of the files in the project, some of which may have changed for the particular commit. Pictorially:



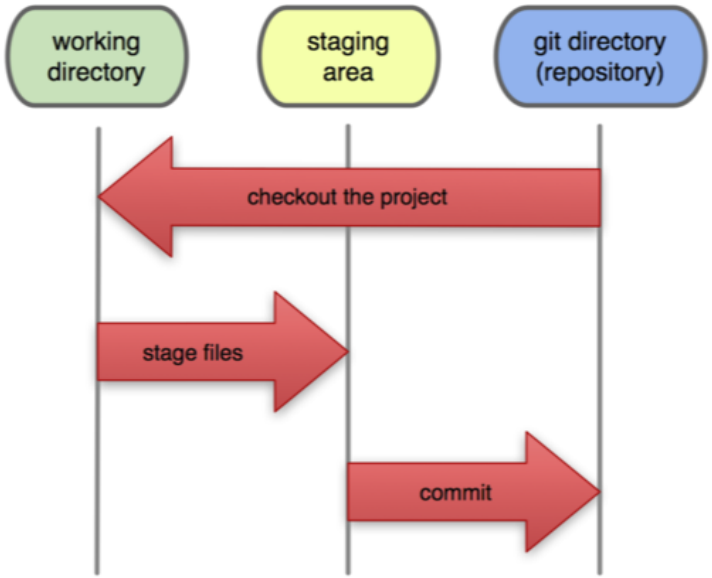
- In Git, every operation you do in normal day-to-day development is local, making it very fast and independent of central repositories or network access.

Local Git Workflow

To Git, a tracked file can be in one of three states:

- modified - working copy is different from repository version
- staged - working copy has been marked for inclusion in the next commit
- committed - working copy matches repository version

Pictorially, the local workflow looks like this:



Setting Up Git

Git configuration stored in three files (on Unix systems):

- `/etc/gitconfig` contains settings for every user on the system
- `~/.gitconfig` contains settings for a specific user, overriding values in `/etc/gitconfig`
- `.git/config` contains settings for a particular reporsitory, overriding values in `~/.gitconfig` and `/etc/gitconfig`

After you install Git, you'll want to configure your identity and your editor

```
$ git config --global user.name "Chris Simpkins"
$ git config --global user.email chris.simpkins@gmail.com
$ git config --global core.editor emacs
```

You can configure settings for the local repository by leaving off the `-global` option.

Initializing a Local Repository

To create a local Git repository, run `git init`

```
$ git init
Initialized empty Git repository in /Users/chris/git-demo/.git/
```

Local repositories are stored in a `.git` subdirectory of your working directory. Now create a file:

```
$ touch README.rst
$ ls
README.rst
```

Getting the Status of a Working Directory

Run git status to get a picture of your local working directory:

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

Usually Git tells you what you need to do next. In this case, we need to run `git add` to track the `README.rst` file.

Adding Files to be Tracked

`git add` adds a file to the staging area (a.k.a. cache, a.k.a. index) to be included in the next commit:

```
$ git add README.rst
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
# new file:   README.rst
```

Remember, a file can be tracked or untracked, and a tracked file can be in one of three states:

- modified - working copy is different from repository version
- staged - working copy marked for inclusion in next commit
- committed - working file same as in repository

Comitting Files to the Repository

Once a file has been staged, it can be committed to the repository with the `git commit` command:

```
$ git commit -m "Initial commit." README.rst
[master (root-commit) 1d84037] Initial commit.
 0 files changed
 create mode 100644 README.rst
```

The `-m` option is the commit message. If you leave it off, Git invokes your `core.editor` (vi if you didn't configure one) so you can add one. Now all our files are committed:

```
ï¿¼$ git status
# On branch master
nothing to commit (working directory clean)
```

Every time we make changes to the file, we add the changed file to the repository with the same two step process:

- `git add` to stage the file for inclusion in the next commit
- `git commit` to add the changed file to the respository

Unstaging Modified Files

Say we stage a change that we want to undo:

```
$ echo "This is the REDME file for the project." >> README.rst
$ git add README.rst
```

As usual, `git status` tells us what to do:

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

To unstage a file that we staged with `git add`, use `git reset HEAD`:

```
$ git reset HEAD README.rst
Unstaged changes after reset:
M README.rst
```

Unmodifying Files

Say we have changes we want to get rid of by replacing our working file with the version in the respository. `git status` tells us how:

```
$ 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:   README.rst
#
no changes added to commit (use "git add" and/or "git commit -a")
```

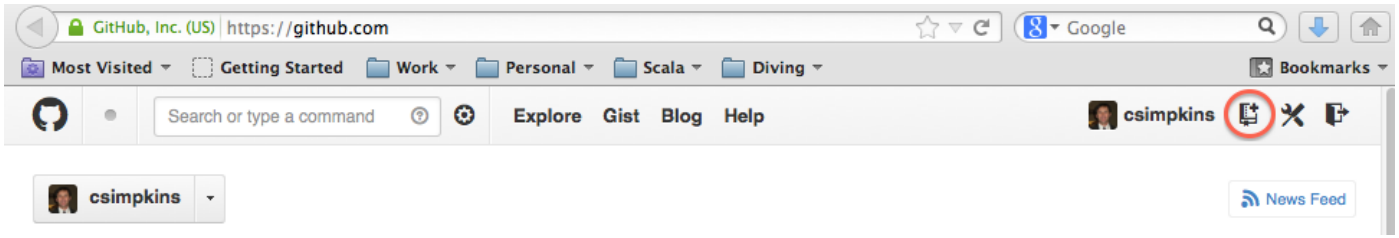
Doing what git status tells us reverts the file.

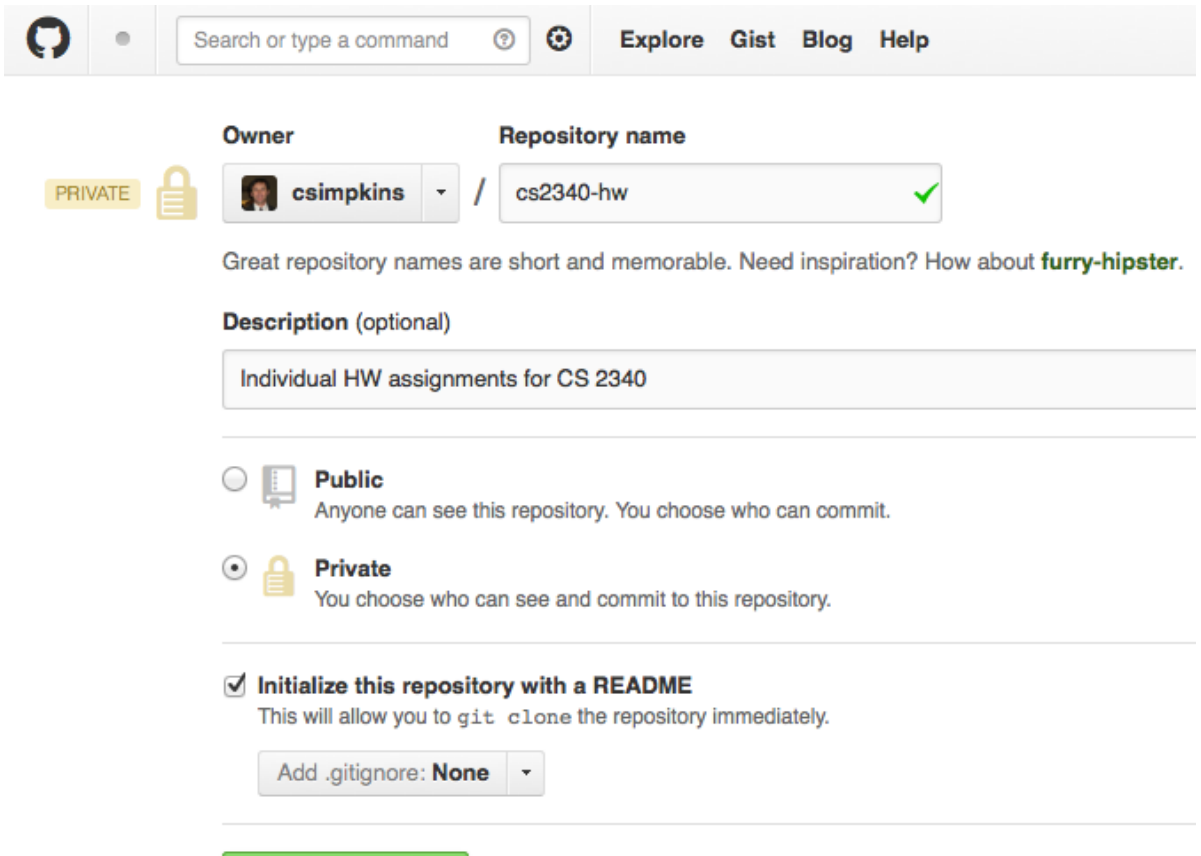
```
$ git checkout -- README.rst
$ git status
# On branch master
nothing to commit (working directory clean)
```

Be careful with this command - it overwrites your working copy.

Creating a GitHub Repository

Log in to github.storm.gatech.edu, Click on Create a New Repo:

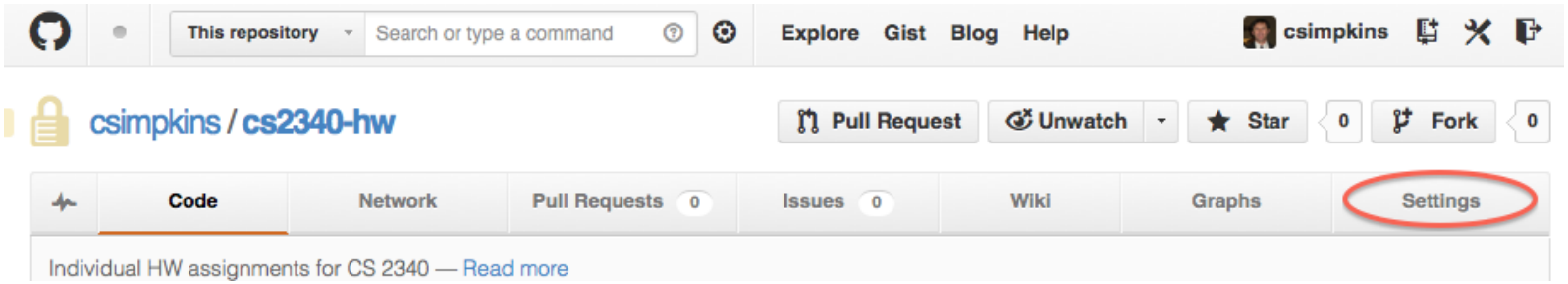




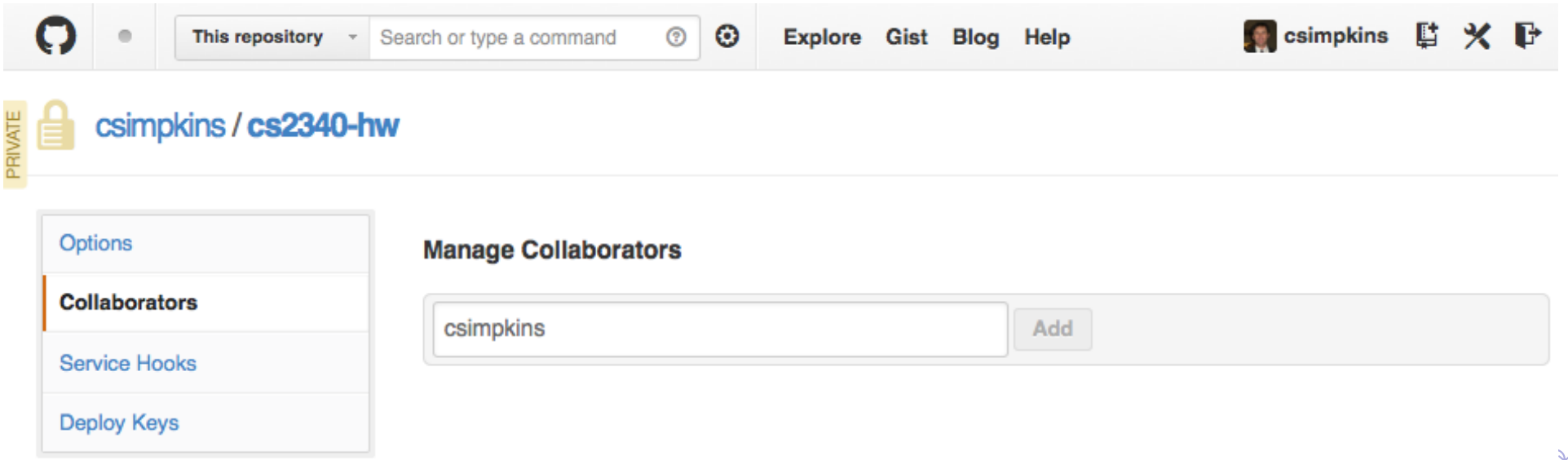
Make it private, and check the option to create a README:

Adding Collaborators

On the repo page, click Settings:

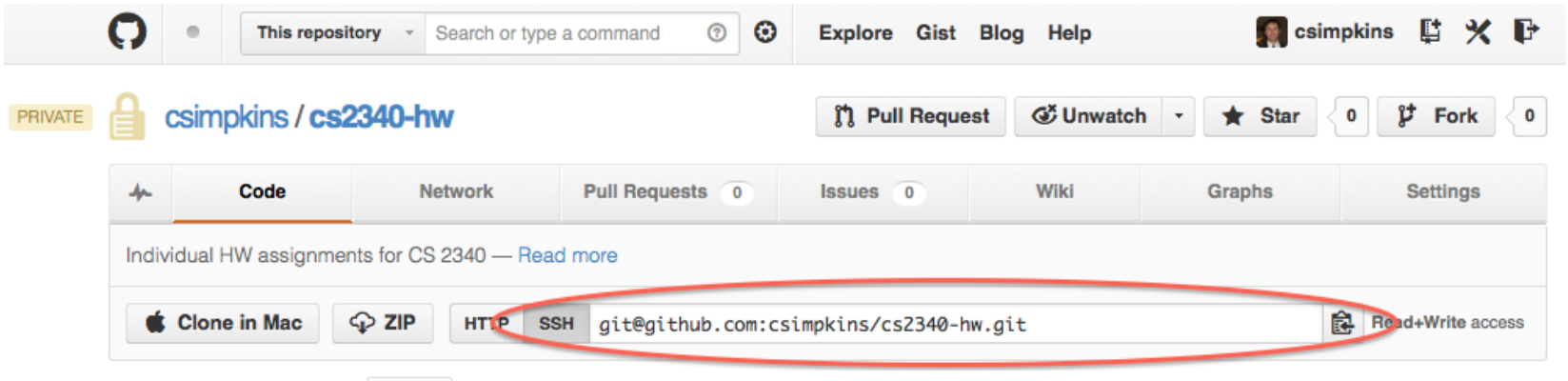


On the settings page, click Collaborators and add me as a collaborator (I'm `csimpkins` on GitHub):



Cloning to Your Local Machine

Back on the repo page, copy the Git URL to your clipboard:



On your local machine, go to the directory where you want your working copy to live and clone the remote GitHub repository like this:

```
$ git clone git@github.com:csimpkins/cs2340-hw.git
Cloning into 'cs2340-hw'...
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
$ cd cs2340-hw/
$ ls
README.md
```

Working With Remotes

A remote repository is a copy of the repository that's on another machine on the network or Internet.

- The GitHub repository we just created is a remote.
- Every remote repository has a name (you can list all your remotes with `git remote -v`). When you clone a remote, that remote automatically get sthe name `origin`.
- Collaborating with others means pushing changes to remotes and pulling changes from remotes.

You can have any number of remote repositories and push, fetch, and pull on any branch. For now we'll focus on our single GitHub remote and the `master` branch.

Pushing Changes To Your Remote

In Git parlance, pushing means uploading to the remote repository. When you've committed your changes to your local repo and want to share them (or simply back them up to your remote) do a `git push`:

```
$ git push origin master
```

- `origin` is the name of your remote, which was set up automatically because you cloned it.
- `master` is the name of the branch you're pushing. For now you'll always be on `master`, which is the default.

To update your local repository with changes from your remote, you use `it fetch` or `git pull`. We'll discuss that in the next Git lesson.

Closing Thoughts

- Version control isn't just for program source code. Learn to store all of your important files in version controllable, diffable, command-line manipulable plain text.
- You now have a general idea how to work with Git. You need much more practice.
- Read the first three chapters of Pro Git.
- Your first homework will be setting up your GitHub account.
- Create practice projects and experiment.