

Princeton University

COS 217: Introduction to Programming Systems

Git and GitHub Primer

Introduction

This document describes how the concepts of version control and file sharing are implemented in Git and GitHub.

For any project involving multiple developers, or even any suitably large one-developer project, it quickly becomes unwieldy and error-prone to maintain multiple stable and developmental versions of the codebase at once. Further, it requires great discipline to consistently document changes to the codebase in order to recall how things were before. To help with managing, and even automating, these necessities of large-scale software development, software engineers use *version control systems*.

Git is one mainstream example of a version control system. Using Git you create source code repositories, which are collections of your project data paired with metadata that tracks changes to that data over time. *GitHub* is an Internet service for hosting Git repositories (one of many!). GitHub is a popular choice for both industry and personal development, and our experience is that even if we were to suggest or encourage another service, most students would end up using GitHub anyway.

Starter code, data, and tools for COS 217 assignments are stored in repositories hosted on GitHub as part of the COS 217 GitHub organization. While it is *possible* for you to access these entirely via the web without actually using Git, doing so would defeat the educational purpose of maintaining the course's assignment resources this way and would subject yourself to the challenges of code maintenance noted in the second paragraph of this document. You should use Git to access those resources, and it would be best if you consistently used Git yourself throughout your development on COS 217 assignments.

This document describes a subset of Git and GitHub that may be sufficient for your work in COS 217. The first sections of this document describe setup steps that you should perform near the beginning of the semester. The remaining sections describe common use cases.

Setup Step 1: Installing Git

Perform this step one time only, as you complete Assignment 0.

Install the Git program on the computer that you will use to do software development. The instructions for installing Git depend upon what kind of computer you will use. If your development computer is:

- An armlab computer, then you can skip this step. Git already is installed on armlab.
- Your own Mac computer (running OS X version 10.9 or above), then issue a `git` command in a terminal window. If Git isn't already installed, then OS X will prompt you to install the Xcode command-line tools. Installing the Xcode command-line tools also installs Git.
- If your Mac computer runs macOS Sonoma and git is not installed, you can install Git through Homebrew:
 1. Install homebrew if you don't already have it,

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

then:

```
$ brew install git
```
- Your own computer running Microsoft Windows, you may already have Git installed thanks to the introcs infrastructure from COS 126, which installs Git Bash. If this is not the case, then browse to <http://git-scm.com/download/win> and the download and install will start automatically.

- Your own computer running Linux, use your package manager to install Git if it is not already installed.

Setup Step 2: Configuring Git

Perform this step one time only on each machine where you will be working, once you have installed Git.

Configure Git to indicate your identity and preferences. To do that, issue these commands in a terminal window:

```
$ git config --global user.name "yourname"
$ git config --global user.email youremailaddress
$ git config --global color.ui auto
$ git config --global core.editor yourpreferrededitor
```

For example, these commands:

```
$ git config --global user.name "Christopher Moretti"
$ git config --global user.email cmoretti@cs.princeton.edu
$ git config --global color.ui auto
$ git config --global core.editor emacs
```

For *youremailaddress* we recommend that you specify your Princeton email address, but it's fine to specify any of your email addresses.

Git uses *yourpreferrededitor* if you issue a `git commit` command without the `-m` option, so this last configuration is optional if you will always supply a commit message on the command line. The `git commit` command is described later in this document.

Setup Step 3: Creating a GitHub Account

Perform this step one time only, once you have installed Git.

Create a GitHub free account. Doing so will allow you to create an unlimited number of private GitHub repositories, including ones with collaborators (which will be necessary if you choose to work with a partner on COS 217 assignments for which that is an option). To create your account:

- Browse to GitHub's homepage: <https://github.com/>
- At the top right, click on the *Sign up* button.
- Enter an email address (preferably your "@princeton.edu" address, but any will suffice), choose a unique and secure password (don't forget it), enter an original and professional username (which will be attached to all your future work), and choose whether you would like to receive marketing emails.
- Solve the "verify your account" puzzles, then GitHub will send you a verification email with a code to enter in your web browser.
- Answer the questions that GitHub asks you about your programming experience and goals, and choose "Continue for free" on the pricing page. Eventually, GitHub displays your new dashboard page, with options to create a new repository, set up your profile, etc.
- Set up a personal access token. GitHub requires you to use a personal access token instead of your account password when operating from the command line. The instructions for creating a personal access token are detailed here: <https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens>, but as a summary: from your profile icon in the upper right corner select *Settings*; on the page that loads select *Developer Settings* from the bottom of the menu on the left; on the page that loads select *Personal access tokens* and choose *Tokens (classic)*; then click *Generate new token (classic)*. For the configuration options: you must check the *repo* checkbox to give your token permissions to access your repositories, and we recommend choosing an expiration date beyond the end of this semester.

- It is important to remember your token – you will want to copy it to a file on your computer. You will use this each time you push to or pull from your GitHub repository. (You will still use your password chosen above to log in to the GitHub website.)
- You will **very quickly** tire of copying and pasting your PAT each time you `pull` or `push`. You can use your computer's keychain or password service to save your credentials, or you can use the Git client itself by configuring it using this command:

```
$ git config --global credential.helper store
```

This will create a file in your home directory, with appropriate permissions so only you can access it, that contains your GitHub username and PAT and will be read by `git` when you invoke future `git` commands, so that you do not have to enter your credentials manually.

Optional Setup Step 4: Creating Your First GitHub Repository

Perform this step to create a new, empty repository to play with. This is **not** necessary to complete the COS 217 assignments (this document will cover that in Setup Step 5), but having a sample repository to experiment on will give you the opportunity to practice your Git workflows and experiment with features not covered in this document.

Continued from the previous step:

- Click on the *Create a repository* button.
- In the resulting page, for *Repository name* enter `cos217test`, select the *Private* radio button, check the *Initialize this repository with a README* checkbox, and click on the *Create repository* button.
- Open a terminal window on your computer. Issue `cd` commands to change your working directory to the one where you want your development Git repository to reside. Issue this command:

```
git clone https://github.com/yourgithubusername/cos217test
```

For example, this command:

```
$ git clone https://github.com/cmorette/cos217test
```

When prompted, enter your GitHub username and GitHub personal access token. The command generates output similar to this (the total number of objects and deltas may vary):

```
Cloning into 'cos217test'...
Username for 'https://github.com': yourgithubusername
Password for 'https://yourgithubusername@github.com':
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
```

Your working directory contains a new directory named `cos217test` which contains a `README.md` file and a `.git` directory. The `cos217test` directory is your *development repository* (aliases: *local repo* or *working copy*)

- To create additional repositories: Browse to <https://github.com/new> and sign into your account. (You can also reach this page by going to the GitHub homepage, signing into your account, then clicking on the plus sign at the top right of the page and choosing *New Repository*.) Choose a name for your new repository, select **Private**, and choose whether you want to add a README or other files automatically, as appropriate.

Setup Step 5: Mirroring/Importing a GitHub Repository

Perform this step once for each assignment throughout the semester to create a new repository of your own for that assignment that starts out with the contents of the COS 217 repository for that assignment. There are two ways to do this, one rather service-agnostic and one very GitHub specific.

Recommended procedure: Using the command line `git push --mirror` command:

- First, issue a special `git clone` command to fetch the metadata of the COS 217 assignment repository that you will use as your starting point without actually creating a working copy of that repository's contents. This is called a "bare clone".

For example, this command for Assignment 0:

```
$ git clone --bare https://github.com/COS217/Survey
```

The command generates output similar to this (the number of objects and deltas will increase over time as the repository accumulates more commits, and the exact messages may vary by platform, e.g., "Unpacking objects" instead of "Receiving objects" and "Resolving deltas"):

```
Cloning into bare repository 'Survey.git'...
remote: Enumerating objects: 14, done.
remote: Counting objects: 100% (14/14), done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 14 (delta 3), reused 13 (delta 2), pack-reused 0
Receiving objects: 100% (14/14), done.
Resolving deltas: 100% (3/3), done.
```

- Before you can use the bare clone metadata to populate a repository of your own, you'll need to create that new private repository.

Recommended procedure: use the GitHub website and the instructions from the last point in Setup Step 4. You will not need to create a README or any other files, and once the repository is created, you do **not** need to clone a working copy of it, as you have not yet mirrored the bare clone into it.

Optional alternative procedure: instead of using the website, creating your own new private repository could be done on the command line using the GitHub API if you configured your personal access token above. But the details of this complicated invocation to do so are beyond the scope of this tutorial:

For example, this command (all on one line):

```
$ curl -u 'yourgithubusername' https://api.github.com/user/repos
-d '{"name":"My_COS217_A0", "private":"true"}
```

- `cd` into the metadata directory produced by the bare clone (`Survey.git` in the example above).
- Issue a special `git push` command that uses the metadata contained in the bare clone to add all the data and metadata from the existing COS 217 assignment repository into your new repository:

For example, this command (yours will have your own GitHub username and repository name):

```
$ git push --mirror https://github.com/cmorette/My_COS217_A0
```

The command generates output similar to this (again, the number of objects and deltas may be higher):
Counting objects: 11, done.
Delta compression using up to 8 threads.

```
Compressing objects: 100% (11/11), done.
Writing objects: 100% (11/11), 4.27 KiB | 4.27 MiB/s, done.
Total 11 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/cmorette/My_COS217_A0
 * [new branch]      main -> main
```

- The metadata directory produced by the bare clone from the COS 217 assignment repository (Survey.git in this example) can now be deleted. Keeping it isn't a problem, though, if you so choose.

At this point, you have created your own repository with all of the COS 217 assignment repository's contents, but you do not yet have a working copy, so you must now follow the third point from Setup Step 4 to change directories to the desired location for the working copy then issue a clone command to create the working copy:

For example, these commands will create a working copy of your new repository in your home directory (again, yours will have your own GitHub username and repository name):

```
$ cd
$ git clone https://github.com/cmorette/My_COS217_A0
```

The command generates output similar to this (again, the number of objects and deltas may be higher):

```
Cloning into 'My_COS217_A0'...
remote: Enumerating objects: 11, done.
remote: Counting objects: 100% (11/11), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 11 (delta 2), reused 11 (delta 2), pack-reused 0
Unpacking objects: 100% (11/11), done.
```

Optional alternative procedure: use the GitHub-specific Import tool:

- Browse to <https://github.com/new/import> and sign into your account. (You can also reach this page by going to the GitHub homepage, signing into your account, then clicking on the plus sign at the top right of the page and choosing *Import Repository*.)
- Fill in the COS 217 assignment repository URL in the input box labeled *Your old repository's clone URL*. In this example, that would be: <https://github.com/COS217/Survey>
- Choose your new repository's name, select **Private**, and click *Begin import*. GitHub's servers will do the work and email you when the import is complete. (The COS 217 assignment repositories are quite small, so you should be able to wait on the page until the process completes, at which point the page will display a hyperlink to your newly created and populated repository.)
- Now you can follow the steps from Setup Step 4 (also found at the top of this page as the last point of the mirror instructions in the recommended procedure) to obtain a working copy of your new repository.

Setup Step 6: Adding Your Partner to Your Repository

Perform this step once for each assignment throughout the semester for which you are working with a partner.

From Startup Step 5, you have your own private GitHub repository that contains all the contents from one of the COS 217 assignment repositories. If you're working with another COS 217 student on an assignment where it is allowable to work with a partner, then you will browse to <https://github.com/yourgithubusername/yourrepositoryname/settings> (also accessible via the GitHub homepage, clicking on your user icon on

the top right, choosing *Your repositories*, selecting the appropriate repository from the list, and then choosing the *Settings* tab) and choose *Manage access* from the menu on the left. Click *Invite a collaborator*, enter your partner's GitHub account name and select the appropriate match, then click the green confirmation that lists the partner's account name and the repository name.

IMPORTANT: Each of your GitHub **assignment repositories must be private** such that only you (and your partner for that assignment, if applicable) have access.

Common Use Cases

Having completed the setup steps, you now have your own private *GitHub repository* in the GitHub cloud and your own *development repository* on your development computer. Use repositories created in Setup Step 4 as vehicles for learning about Git. In particular, try implementing the use cases from the remainder of this document. Experiment!

Use Case 1: Adding Files

Perform this step repeatedly throughout the semester as required.

Add *file1* and *file2* to your development repository and your GitHub repository. To do that, in a terminal window on your development computer issue these commands:

```
# cd to the local repo directory.
cd reponame

# Pull any updates from the GitHub repo down to the local repo.
git pull
```

Use an editor to create *file1* and *file2*.

```
# Stage file1 and file2.
git add file1 file2

# Commit the staged files to the local repo.
git commit -m "Message describing the commit"
```

Note: If you omit the `-m "Message describing the commit"` option, then Git launches the editor that you specified in Setup Step 2, so you can compose a message using that editor.

```
# Push the local repo to the GitHub repo.
git push origin main

# (Optionally) check status.
git status
```

Use Case 2: Changing Files

Perform this step repeatedly throughout the semester as required.

Change *file1* and *file2* in your development repository and your GitHub repository. To do that, in a terminal window on your development computer issue these commands:

```
# cd to the local repo directory.
cd repodir

# Pull any updates from the GitHub repo down to the local repo.
```

```
git pull
```

If this is a local repository that you don't actively edit (e.g., because you code in an IDE on a different machine, the commands above will suffice to update the changes made elsewhere. For local development repositories that you are editing (e.g., the one on your machine with your IDE), though, you will use additional commands to update the GitHub repository with your local development repository's changes. After you have made changes to *file1* and *file2*, issue these commands:

```
# Stage file1 and file2.
git add file1 file2

# Commit the staged files to the local repo.
git commit -m "Message describing the commit"

# Push the local repo to the GitHub repo.
git push origin main

# (Optionally) check status.
git status
```

Use Case 3: Removing Files

Perform this step repeatedly throughout the semester as required.

Remove *file1* and *file2* from your development repository and your GitHub repository. To do that, in a terminal window on your development computer issue these commands:

```
# cd to the local repo directory.
cd repodir

# Pull the GitHub repo to the local repo.
git pull

# Stage file1 and file2.
git rm file1 file2

# Commit the staged files to the local repo.
git commit -m "Message describing the commit"

# Push the local repo to the GitHub repo.
git push origin main

# (Optionally) check status.
git status
```

Use Case 4: Resolving Conflicts

Perform this step repeatedly throughout the semester as required.

Hopefully it is rare, which it should be if you use good Git discipline with yourself and if when working with a partner you each use good Git discipline and you communicate clearly. As a concrete piece of advice, try to avoid conflicts by (1) pulling from your GitHub repository before starting each programming task, and (2) pushing to your GitHub repository as soon as the programming task is finished to your satisfaction.

You experience a *conflict* when (1) your partner pushes a new version of a file to your GitHub repository, (2) you edit an old version of the same file in your development repository, and (3) you attempt to pull from the GitHub

repository to your development repository or push from your development repository to the GitHub repository.

This sequence of commands illustrates a conflict and how to resolve it.

Your partner issues these commands:

```
# cd to the local repo directory.
cd repodir

# Pull the GitHub repo to the local repo.
git pull
```

You issue these commands:

```
# cd to the local repo directory.
cd repodir

# Pull the GitHub repo to the local repo.
git pull
```

Your partner uses an editor to change *file1*. Your partner then issues these commands:

```
# Stage file1.
git add file1

# Commit the staged files to the local repo.
git commit -m "Message describing the commit"
```

You use an editor to change *file1*. You then issue these commands:

```
# Stage file1.
git add file1

# Commit the staged files to the local repo.
git commit -m "Message describing the commit"
```

Your partner issues this command:

```
# Push the local repo to the GitHub repo.
git push origin main
```

You issue this command:

```
# Push the local repo to the GitHub repo; fails!
git push origin main
```

Your `git push` command fails because of a conflict. Git recommends that you pull from the GitHub repository. So you issue this command:

```
# Pull the GitHub repo to your local repo.
git pull
```

Git notes that *file1* is in conflict. Git annotates *file1* to note the points of conflict. You edit *file1* to resolve the conflicts and eliminate the annotations. Then you issue these commands:


```
# Stage file1
git add file1

# Commit the staged files to the local repo.
git commit -m "Message describing the commit"

# Push the local repo to the GitHub repo; succeeds!
git push origin main
```

Use Case 5: Branching

Perform this step repeatedly throughout the semester as required.

For COS 217, it may be possible to avoid using branches, since assignments are self-contained and there are never more than two users on a project. That said, it's a very useful feature to know, as it is the backbone of most larger development projects.

You decide to implement a new experimental feature in your application. It may not work out, so you don't want to implement the experiment in the *main* branch. Instead you decide to implement the experiment in a new branch named *exp*.

You issue these commands:

```
# Pull the GitHub repo to your local repo.
git pull

# Create a new branch named exp.
git branch exp

# Make exp the current branch.
git checkout exp
```

You edit *file1* extensively, but not completely. Oh no! Your partner informs you of a bug in *file1* that you must fix in a hurry. It's a good thing that you're implementing your experiment in a non-main branch! You issue these commands:

```
# Stage file1.
git add file1

# Commit the staged files to the local repo.
git commit -m "Message describing the commit"

# Make main the current branch.
git checkout main
```

You edit *file1* to fix the bug. Then you issue these commands:

```
# Stage file1.
git add file1

# Commit the staged files to the local repo.
git commit -m "Message describing the commit"

# Push the main branch to the origin (GitHub) repo.
git push origin main
```

You decide to resume your work on the experiment. Of course you must merge your bug fix into your *exp* branch.

You issue these commands:

```
# Make exp the current branch.
git checkout exp

# Merge the main branch into the current (exp) branch.
git merge main
```

Git notes that *file1* is in conflict. Git annotates *file1* to note the points of conflict. You edit *file1* to resolve the conflicts and eliminate the annotations. Then you issue these commands:

```
# Stage file1.
git add file1

# Commit the staged files to the local repo.
git commit -m "Message describing the commit"
```

You finish editing *file1* with your experimental code. Then you issue these commands:

```
# Stage file1.
git add file1

# Commit the staged files to the local repo.
git commit -m "Message describing the commit"
```

You're happy with the outcome of the experiment, so you decide to merge your *exp* branch into the main branch. You issue these commands:

```
# Make main the current branch.
git checkout main

# Merge the exp branch into the current (main) branch.
git merge exp

# Push the main branch to the origin (GitHub) repo.
git push origin main
```

You decide that you no longer need your *exp* branch, so you issue this command:

```
# Delete the exp branch from your local repo.
git branch -d exp
```

Enrichment

There is much more to Git. To learn more we recommend that you read the Git book at <https://git-scm.com/book/en/v2> and the Git reference manual at <https://git-scm.com/docs>

There also is much more to GitHub. We recommend that you investigate these features after you're comfortable with GitHub basics:

- Projects: for tracking goals and progress of a general "project".

- The official documentation is at:
<https://help.github.com/en/github/managing-your-work-on-github/about-project-boards>
- A reasonable video tutorial is at:
<https://youtu.be/ff5cBkPg-bQ>
- Issues and Pull Requests: for defining and implementing small program features in a collaborative and scalable way.
 - An introductory guide is at:
<https://guides.github.com/activities/hello-world/>
 - A verbose set of documents about GitHub "flow" is at
<https://help.github.com/en/github/collaborating-with-issues-and-pull-requests>
- GitHub Pages: for website hosting directly from source code.
 - GitHub's own Pages introduction is at:
<https://pages.github.com/>
- Actions: for automatically running code like linters, builders, and deployers whenever you push to GitHub.
 - The official documentation is at:
<https://help.github.com/en/actions/automating-your-workflow-with-github-actions>
- Tags and Releases: for creating "versions" of your code both to release and reference.
 - "Tagging" in Git is covered at:
<https://git-scm.com/book/en/v2/Git-Basics-Tagging>
 - Creating GitHub Releases is covered at:
<https://help.github.com/en/github/administering-a-repository/creating-releases>

Copyright © 2024 by Robert M. Dondero, Jr., Joseph Eichenhofer, and Christopher Moretti