ECE 4750 Computer Architecture, Fall 2016
Tutorial 2: Git Distributed Version Control System

School of Electrical and Computer Engineering
Cornell University

revision: 2016-10-02-21-51

1. Introduction

In this course, we will be using Git as our revision control and source code management system. We will be using GitHub for centralized online repository hosting, and TravisCI for online continuous integration testing. These tools will enable us to adopt an agile hardware development methodology so your group can rapidly collaborate and iterate on the design, verification, and evaluation of the assignments. This tutorial covers how to: setup your GitHub account, use Git for basic incremental development, use GitHub to collaborate with your group, manage Git branches and GitHub pull requests, use TravisCI for continuous integration, and use the ECE 4750 lab admin script. This tutorial assumes that you have completed the Linux tutorial.

To follow along with the tutorial, access the ECE computing resources and type the commands without the `%` character. In addition to working through the commands in the tutorial, you should also try the more open-ended tasks marked with the ★ symbol.

Before you begin, make sure that you have **sourced the ece4750-setup script** or that you have added it to your `.bashrc` script, which will then source the script every time you login. Sourcing the setup script sets up Git configuration details required for this tutorial.

2. Setting up Your GitHub Account

GitHub is an online service that hosts centralized Git repositories for a growing number of open-source projects. It has many useful features including a web-based source browser, history browser, branch management, merge requests, code review, issue tracking, and even a built-in wiki attached to every repository. We have created a dedicated GitHub organization for the course located here:

- `https://github.com/cornell-ece4750`

You will learn in Section 6 how to join this organization. For most of this tutorial you will be using a repository in your own personal GitHub account. If you do not yet have an GitHub account, you can create one here:

- `https://github.com/join`

Your NetID makes a great GitHub ID. Be sure to use your Cornell University email address. Once your account is setup, please make sure you set your full name so we can know who you are on GitHub. Please also consider uploading a profile photo to GitHub; it makes it more fun to interact on GitHub if we all know what each other look like. Go to the following page and enter your first and last name in the *Name* field, and then consider uploading a profile photo.

- `https://github.com/settings/profile`

Before you can begin using GitHub, you need to create an SSH key pair on an `ecelinux` machine and upload the corresponding SSH public key to GitHub. GitHub uses these keys for authentication. The course setup script takes care of creating an SSH key pair which you can use. Login to an `ecelinux` machine, source the ece4750-setup script, and then view the contents of your public key using the following commands:

```
% source setup-ece4750.sh
% cat ~/.ssh/ece4750-github.pub
```

Use the following page to upload the public key to GitHub:

- `https://github.com/settings/ssh`

Click on *Add SSH Key*, and then cut-and-paste the public key you displayed using `cat` into the *key* textbox. Give the key the title "ece4750-github". Then click *Add key*. To test things out try the following on an `ecelinux` machine.

```
 % ssh -T git@github.com
```

You may see a warning about the authenticity of the host. Don't worry, this is supposed to happen the first time you access GitHub using your new key. Just enter "yes". The GitHub server should output some text including your GitHub ID. Verify that the GitHub ID is correct, and then you should be all set. There are two good GitHub Guides you might want to take a look at:

- `https://guides.github.com/activities/hello-world`
- `https://guides.github.com/introduction/flow`

GitHub has two integrated tools that students might find useful: an issue tracker and a wiki. Consider using the GitHub issue tracker to track bugs you find in your code or to manage tasks required to complete the lab assignment. You can label issues, comment on issues, and attach them to commits. See the following links for more information about GitHub issues:

- `https://guides.github.com/features/issues`
- `https://help.github.com/articles/about-issues`

Consider using the GitHub per-repository wiki to create task lists, brainstorm design ideas, rapidly collaborate on text for the lab report, or keep useful command/code snippets. See the following links for more information about GitHub wikis:

- `https://guides.github.com/features/wikis`
- `https://help.github.com/articles/about-github-wikis`

3. Git and GitHub

In this section, we begin with a basic single-user workflow before demonstrating how Git and Github can be used for effective collaboration among multiple users. We discuss how to resolve conflicts and how to manage branches and pull requests.

3.1. Single-User Workflow

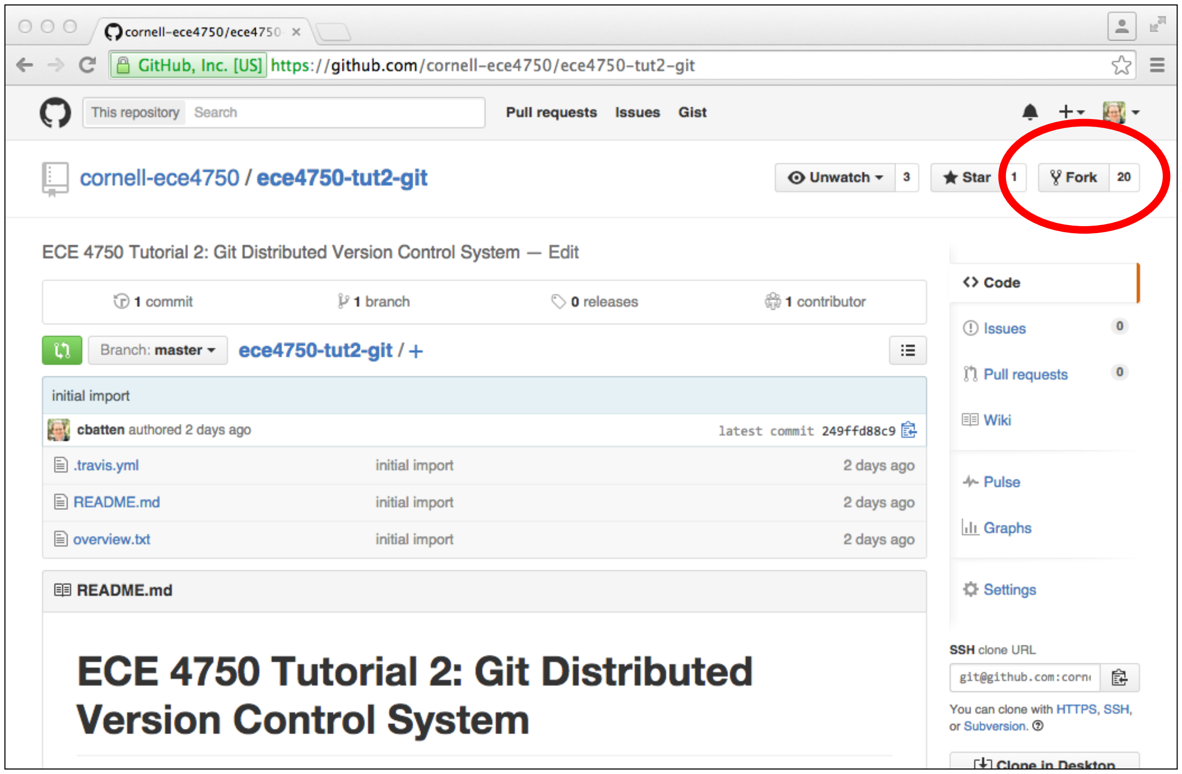
In this section, we cover some basic Git commands and illustrate a simple Git workflow. We have created a Git repository that we will be using as an initial template, so the first step is to *fork* this tutorial repository. Forking is the process of making a personal copy of someone else's repository on GitHub. Start by going to the GitHub page for the tutorial repository located here:

- `https://github.com/cornell-ece4750/ece4750-tut2-git`

Figure 1 shows the corresponding GitHub page. Click on *Fork* in the upper right-hand corner. If asked where to fork this repository, choose your personal GitHub account. After a few seconds, you should have a brand new repository in your account:

- `https://github.com/<githubid>/ece4750-tut2-git`

Where `<githubid>` is your GitHub ID, not your NetID. Now that you have your own copy of the tutorial repository, the next step is to *clone* this repository to an `ecelinux` machine so you can manipulate the content within the repository. We call the repository on GitHub the *remote repository* and



ECE 4750 Tutorial 2: Git Distributed Version Control System — Edit

initial import

cbatten authored 2 days ago latest commit 249ffd88c9

.travis.yml	initial import	2 days ago
README.md	initial import	2 days ago
overview.txt	initial import	2 days ago

ECE 4750 Tutorial 2: Git Distributed Version Control System

You can clone with HTTPS, SSH, or Subversion.

Figure 1: Forking the `ece4750-tut2-git` **Repository**

we call the repository on the `ecelinux` machine the *local repository*. A local repository is a first-class mirror of the remote repository with the entire history of the repository, and thus almost all operations are essentially local requiring no communication with GitHub. The following commands write an environment variable with your GitHub ID, create a subdirectory for this tutorial in your home directory before using the `git clone` command to clone the remote repository and thus create a local repository.

```
% source setup-ece4750.sh
% GITHUBID="<githubid>"
% mkdir -p ${HOME}/ece4750
% cd ${HOME}/ece4750
% git clone git@github.com:${GITHUBID}/ece4750-tut2-git tut2
% cd tut2
% TUTROOT=${PWD}
```

Where again `<githubid>` is your GitHub ID, not your NetID. The `git clone` command takes two command line arguments. The first argument specifies the remote repository on GitHub you would like to clone, and the second argument specifies the name to give to the new local repository. Note that we created an environment variable with the directory path to the local repository to simplify navigating the file system in the rest of this tutorial.

The repository currently contains two files: a `README` file, and `overview.txt` which contains an overview of the course. These files are contained within what we call the *working directory*. The repository also includes a special directory named `.git` which contains all of the extra repository metadata. You should never directly manipulate anything within the `.git` directory.

```
% cd ${TUTROOT}
% ls -la
```

Let's assume we want to create a new file that contains a list of fruits, and that we want to manage this file using Git version control. First, we create the new file.

```
% cd ${TUTROOT}
% echo "apple" > fruit.txt
```

To manage a file using Git, we need to first use the `git add` command to tell Git that it should track this file from now on. We can then use `git commit` to commit our changes to this file into the repository, and `git log` to confirm the result.

```
% cd ${TUTROOT}
% git add fruit.txt
% git commit -m "initial fruit list"
% git log
```

The `-m` command line option with the `git commit` command enables you to specify a *commit message* that describes this commit. All commit messages should include a "subject line" which is a single *short* line briefly describing the commit. Many commits will just include a subject line (e.g., the above commit). If you want to include more information in your commit message then skip the `-m` command line option and Git will launch your default editor. You still want to include a subject line at the top of your commit message, but now you can include more information separated from the subject line by a blank line.

Note, you can learn about any Git command and its usage by typing `git help <command>`, where `<command>` should be substituted by the actual name of the command. This would display the output similar to the manual pages for a Linux command, as seen in Tutorial 1. Students are encouraged to learn more about each Git command beyond the details covered in this tutorial.

The `git log` command displays information about the commit history. The beginning of the output from `git log` should look something like this:

```
commit 0e5b2b2c05b5837839554fa047e52e121c8206b1
Author: cb535 <cb535@cornell.edu>
Date:   Sat Aug 18 18:01:17 2015 -0400

    initial import of fruit
```

Conceptually, we should think of each commit as a copy of all of the tracked files in the project at the time of the commit. This commit just included changes to one file, but as we add more files each commit will include more and more information. The history of a git repository is just a long sequence of commits that track how the files in the repository have evolved over time. Notice that Git has recorded the name of who made the commit, the date and time of the commit, and the log message. The first line is the commit id which uniquely identifies this commit. Git does not use monotonically increasing revision numbers like other version control systems, but instead uses a 40-digit SHA1 hash as the commit id. This is a hash of *all* the files included as part of this commit (not just the changes). We can refer to a commit by the full hash or by just the first few digits as long as we provide enough digits to unambiguously reference the commit. Now let's add a fruit to our list and commit the change.

```
% cd ${TUTROOT}
% echo "mango" >> fruit.txt
% git commit -m "added mango to fruit list"
```

Unfortunately, this doesn't work. The output from `git commit` indicates that there have been no changes since the last commit so there is no need to create a new commit. Git has a concept of an *index* which is different compared to other version control systems. We must "stage" files (really we stage content not files) into the index, and then `git commit` will commit that content into the repository. We can see this with the `git status` command.

```
% cd ${TUTROOT}
% git status
```

which should show that `fruit.txt` is modified but not added to the index. We stage files in the index with `git add` like this:

```
% cd ${TUTROOT}
% git add fruit.txt
% git status
```

Now `git status` should show that the file is modified and also added to the index. Our commit should now complete correctly.

```
% cd ${TUTROOT}
% git commit -m "added mango to fruit list"
% git status
```

So even though Git is tracking `fruit.txt` and knows it has changed, we still must explicitly add the files we want to commit. There is a short cut which uses the `-a` command line option with the `git commit` command. This command line option tells Git to commit any file which has changed and was previously added to the repository.

```
% cd ${TUTROOT}
% echo "orange" >> fruit.txt
% git commit -a -m "added orange to fruit list"
% git status
```

Staging files is a useful way to preview what we will commit before we actually do the commit. This helps when we have many changes in our working directory but we don't want to commit them all at once. Instead we might want to break them into smaller, more meaningful commits or we might want to keep working on some of the modified files while committing others.

Figure 2 illustrates how the commands we have used so far create a single-user development workflow. The `git clone` command copies the remote repository to create a local repository which includes both the working directory and the special `.git` directory. The `git add` command adds files to the index from the working directory. The `git commit` command moves files from the index into the special `.git` directory. The `-a` command line option with the `git commit` command can commit files directly from the working directory to the special `.git` directory.

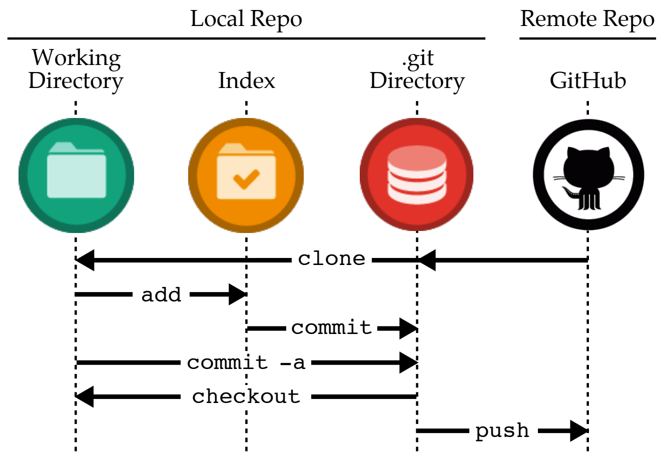


Figure 2: Git and GitHub for Single-User Development

Now that we have made some changes, we can use `git log` to view the history of last few commits and then add another line to the `fruit.txt` file.

```
% cd ${TUTROOT}
% git log
% echo "plum" >> fruit.txt
% cat fruit.txt
```

Imagine you didn't like your changes and want to revert the changes, you would use the `git checkout` command as below.

```
% cd ${TUTROOT}
% git checkout fruit.txt
% cat fruit.txt
```

As illustrated in Figure 2, the `git checkout` command resets any a file or directory to the state it was in at the time of the last commit. The output from the `git status` command should look something like this:

```
% cd ${TUTROOT}
% git status
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)
nothing to commit, working directory clean
```

The `git status` command is telling us that the local clone of the repository now has more commits than the remote repository on GitHub. If you visit the GitHub page for this repository you will not see any changes. This is a critical difference from other centralized version control systems. In Git, when we use the `git commit` command it only commits these changes to your *local repository*.

If we have done some local work that we are happy with, we can push these changes to the *remote repository* on GitHub using the `git push` command.

```
% cd ${TUTROOT}
% git push
% git status
```

Notice that the output of the `git status` command indicates that our local repository is up-to-date with the remote repository on GitHub. Figure 2 shows visually the idea that the `git push` command moves commits from your local repository to the remote repository on GitHub. Visit the GitHub page to verify that our new commits have been pushed to the remote repository:

- `https://github.com/<githubid>/ece4750-tut2-git`

Click on *commits* at the top of the GitHub page to view the log of commits. You can browse who made each commit, what changed in each commit, and the state of the repository at each commit. Return to the main GitHub page for the repository and click on the `fruit.txt` file to view it.

★ *To-Do On Your Own:* Create a new file called `shapes.txt` that includes a list of different shapes. Commit the new file, make some edits, and commit these edits. Use `git status` and `git log` to keep track of your changes. Push your changes to GitHub and browse the updated files on GitHub.

3.2. Multi-User Workflow

Since your tutorial repository is public on GitHub, any other user can also clone this repository. If you would like to collaborate with another GitHub user, you would need to give that user read/write permission. Section 6 describes a lab admin script we have created to simplify setting up the permissions so that a lab group can collaborate on the lab assignments. To emulate how collaboration with GitHub works, we will "pretend" to be different users by cloning extra copies of the tutorial repository.

```
% cd ${HOME}/ece4750
% git clone git@github.com:${GITHUBID}/ece4750-tut2-git tut2-alice
% cd tut2-alice
% ALICE=${PWD}
% cd ${HOME}/ece4750
% git clone git@github.com:${GITHUBID}/ece4750-tut2-git tut2-bob
% cd tut2-bob
% BOB=${PWD}
```

We can now emulate different users by simply working in these different local repositories: when we work in `ALICE` we will be acting as the user Alice, and when we work in `BOB` we will be acting as the user Bob. Figure 3 illustrates a multi-user development environment: both Alice and Bob have their own separate local repositories (including their own working directories, index, and special `.git` directories), yet they will both communicate with the same centralized remote repository on GitHub.

Let's have Alice add another entry to the `fruit.txt` file, commit her changes to her local repository, and then push those commits to the remote repository on GitHub:

```
% cd ${ALICE}
% echo "banana" >> fruit.txt
```

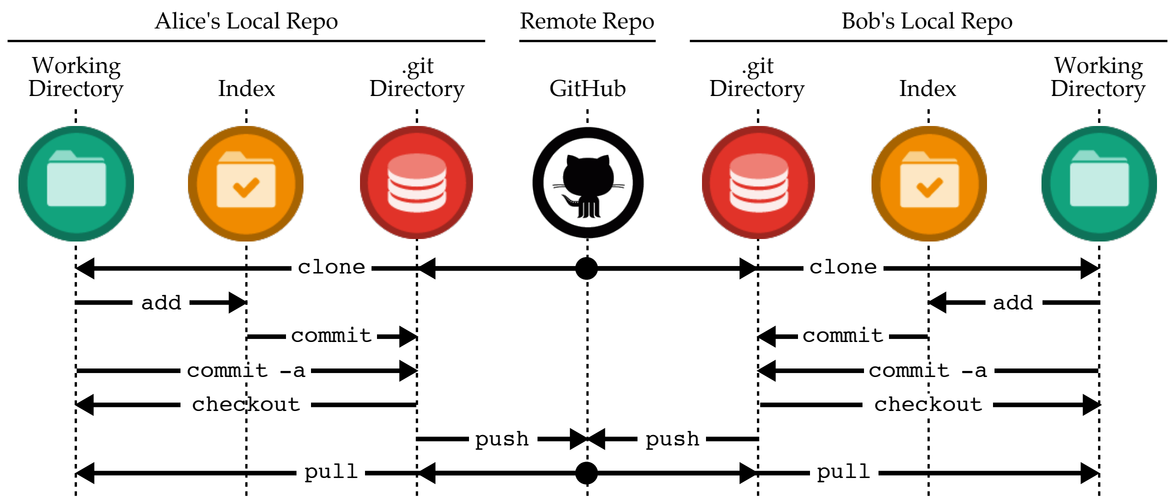


Figure 3: Git and GitHub for Multi-User Development

```
% git commit -a -m "ALICE: added banana to fruit list"
% git log --oneline
% git push
% cat fruit.txt
```

If you view the GitHub page for this repository it will appear that you are the one making the commit (remember we are just pretending to be Alice), which is why we used `ALICE:` as a prefix in the commit message.

Now let's assume Bob wants to retrieve the changes that Alice just made to the repository. Bob can use the `git pull` command to pull all new commits from the remote repository into his local repository. The `git pull` command performs two actions, it first fetches all the updates and then merges or applies them to the local project. If there are no conflicts in the file contents, the command executes successfully. If there are conflicts, the command does not merge all the changes and reports the conflicting content. We will learn how to resolve conflicts in Section 3.3.

```
% cd ${BOB}
% git pull
% git log --oneline
% cat fruit.txt
```

Figure 3 shows visually the idea that the `git pull` command moves commits from the remote repository on GitHub to your local repository. Bob's copy of tutorial repository should contain Alice's most recent commit and his copy of the `fruits.txt` file should include "banana". Now let's assume Bob also wants to make some changes and push those changes to the remote repository on GitHub:

```
% cd ${BOB}
% echo "peach" >> fruit.txt
% git commit -a -m "BOB: added peach to fruit list"
% git log --oneline
% git push
% cat fruit.txt
```

Similar to before, Alice can now retrieve the changes that Bob just made to the repository using the `git pull` command.

```
% cd ${ALICE}
% git pull
% git log --oneline
% cat fruit.txt
```

This process is at the key to collaborating via GitHub. Each student works locally on his or her part of the lab assignment and periodically pushes/pulls commits to synchronize with the remote repository on GitHub.

★ *To-Do On Your Own:* Create a new file called `letters.txt` in Bob's local repository that includes a list of letters from A to M, one per line. Commit the new file, make some edits to add say more letters from say M to Z, and commit these edits. Use `git push` to push these commits to the centralized repository. Switch to Alice's local repository and use `git pull` to pull in the new commits. Verify that all of your files and commits are in both Bob's and Alice's local repositories.

3.3. Resolving Conflicts

Of course the real challenge occurs when both Alice and Bob modify content at the same time. There are two possible scenarios: Alice and Bob modify different content such that it is possible to combine their commits without issue, or Alice and Bob have modified the exact same content resulting in a conflict. We will address how to resolve both scenarios.

Let us assume that Alice wants to add `lemon` to the list and Bob would like to create a new file named `vegetables.txt`. Alice would go ahead and first pull from the central repository to grab any new commits from the remote repository on GitHub. On seeing that there are no new commits, she edits the file, commits, and pushes this new commit.

```
% cd ${ALICE}
% git pull
% echo "lemon" >> fruit.txt
% git commit -a -m "ALICE: added lemon to fruit list"
% git push
```

Since Bob recently pulled from the remote repository on GitHub, let's say he assumes that there have been no new commits. He would then go ahead and create his new file, commit, and attempt to push this new commit.

```
% cd ${BOB}
% echo "spinach"  >  vegetables.txt
% echo "broccoli" >> vegetables.txt
% echo "turnip"   >> vegetables.txt
% git add vegetables.txt
% git commit -m "BOB: initial vegetable list"
% git push
To git@github.com:<githubid>/ece4750-tut2-git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'git@github.com:<githubid>/ece4750-tut2-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 integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

On executing the sequence of commands above, you should notice that Git does not allow Bob to push his changes to the central repository as the version of the central repository has been updated by Alice. You should see a message similar to the one above. Git suggests us to merge the remote commits before pushing the local commits. We can do so by first using the `git pull` command to merge the local commits.

```
% cd ${BOB}
% git pull
```

Git will launch your default text editor because we need to *merge* your local commits and the remote commits. You will need to enter a commit message, although usually the default message provided by Git is fine. After saving the commit message, we can take a look at the Git history using `git log` to see what happened.

```
% cd ${BOB}
% git log --oneline --graph
*   656965a Merge branch 'master' of github.com:<githubid>/ece4750-tut2-git
|\
| * 9d943f3 ALICE: added lemon to fruit list
* | 4788dbe BOB: initial vegetable list
|/
* c7cc31e BOB: added peach to fruit list
* dc28bc9 ALICE: added banana to fruit list
```

The `--graph` command line option with the `git log` command will display a visual graph of the commit history. You can see that Bob and Alice worked on two different commits at the same time. Alice worked on commit `9d943f3` while Bob was working on commit `4788dbe`. Bob then merged these two sets of commits using a new commit `656965a`. Bob can now push his changes to the remote repository in GitHub.

```
% cd ${BOB}
% git push
```

GitHub has a nice commit history viewer which shows a similar commit graph as we saw above:

- `https://github.com/<githubid>/ece4750-tut2-git/network`

While this approach is perfectly reasonable, it can lead to a very non-linear and complex commit history. We strongly recommend that you use an alternative called *rebasing*. While merging "merges" sets of commits together, rebasing will apply one set of commits first and then apply a second set of commits on top of the first set of commits. This results in a more traditional linear commit history. Let's reconstruct a similar situation as before where Alice adds another fruit and Bob adds another vegetable at the same time.

```
% cd ${ALICE}
% git pull
% echo "plum" >> fruit.txt
% git commit -a -m "ALICE: added plum to fruit list"
% git push
% cd ${BOB}
% echo "potato" >> vegetables.txt
% git commit -a -m "BOB: added potato to vegetable list"
% git push
To git@github.com:<githubid>/ece4750-tut2-git
 ! [rejected]        master -> master (fetch first)
```

To rebase, we will use the `--rebase` command line option with the `git pull` command.

```
% cd ${BOB}
% git pull --rebase
% git push
% git log --oneline --graph
* 0d5fba5 BOB: added potato to vegetable list
* e56ad1b ALICE: added plum to fruit list
*   656965a Merge branch 'master' of github.com:<githubid>/ece4750-tut2-git
|\
| * 9d943f3 ALICE: added lemon to fruit list
* | 4788dbe BOB: initial vegetable list
|/
* c7cc31e BOB: added peach to fruit list
* dc28bc9 ALICE: added banana to fruit list
```

Study the output from the `git log` command carefully. Notice how instead of creating a new merge commit as before, rebasing has applied Alice's commit `e56ad1b` first and then applied Bob's new commit `0d5fba5` on top of Alice's commit. Rebasing keeps the git history clean and linear.

Sometimes Alice and Bob are editing the exact same lines in the exact same file. In this case, Git does not really know how to resolve this *conflict*. It does not know how to merge or rebase the two sets of commits to create a consistent view of the repository. The user will have to manually resolve the conflict. Let's explore what happens when Alice and Bob want to add a new fruit to the `fruits.txt` file at the exact same time. First, Alice adds `kiwi` and pushes her updates to the remote repository on GitHub.

```
% cd ${ALICE}
% git pull
% echo "kiwi" >> fruit.txt
% git commit -a -m "ALICE: added kiwi to fruit list"
% git push
```

Now Bob adds `date` and tries to push his update to the remote repository on GitHub.

```
% cd ${BOB}
% echo "date" >> fruit.txt
% git commit -a -m "BOB: added date to fruit list"
% git push
To git@github.com:<githubid>/ece4750-tut2-git
 ! [rejected]        master -> master (fetch first)
```

As before, Bob uses the `--rebase` command line option with the `git pull` command to pull the commits from the remote repository on GitHub.

```
% cd ${BOB}
% git pull --rebase
First, rewinding head to replay your work on top of it...
Applying: BOB: added date to fruit list
Using index info to reconstruct a base tree...
M       fruit.txt
Falling back to patching base and 3-way merge...
Auto-merging fruit.txt
CONFLICT (content): Merge conflict in fruit.txt
Failed to merge in the changes.
```

Git indicates that it was not able to complete the rebase. There is a conflict in the `fruit.txt` file. We can also use the `git status` command to see which files have conflicts. They will be marked as `both modified`:

```
% cd ${BOB}
% git status
```

Git instructs Bob to first resolve the conflict and then use the `--continue` command line option with the `git rebase` command to finish the rebase. If you take a look at the `fruit.txt` file you will see that it now includes conflict markers showing exactly where the conflict occurred.

```
% cd ${BOB}
% cat fruit.txt
apple
mango
orange
banana
peach
lemon
plum
<<<<<<< 7f8ec0fb9d7c705e3caf9034f3b96b0c8e7cad92
kiwi
=======
date
>>>>>>> BOB: added date to fruit list
```

This shows that the commit from the remote repository on GitHub has `kiwi` as the last line in the file, while the last commit to the local repository has `date` on the last line in the file. To resolve the conflict we can directly edit this file so that it reflects how we want to merge. We can choose one fruit over the other, choose to include neither fruit, or choose to include both fruit. Edit the file using your favorite text editor to remove the lines with markers <<<<, === and >>>> so that the file includes both fruit.

```
% cd ${BOB}
% cat fruit.txt
apple
```

```
mango
orange
banana
peach
lemon
plum
kiwi
date
```

The next step is critical and easy to forget. We need to use the `git add` command to add any files that we have fixed! In this case we need to use the `git add` command for the `fruit.txt` file.

```
% cd ${BOB}
% git status
% git add fruit.txt
% git status
```

Notice how the output from the `git status` command has changed to indicate that we have fixed the conflict in the `fruit.txt` file. Since we have resolved all conflicts, we can now continue the rebase:

```
% cd ${BOB}
% git rebase --continue
% git push
% git log --oneline --graph
% cat fruit.txt
```

Resolving conflicts is tedious, so to avoid conflicts you should communicate with your group members which student is going to be executing which files. Try to avoid having multiple students working on the same file at the same time, or at least avoid having multiple students working on the same lines of the same file at the same time.

★ *To-Do On Your Own:* Experiment with having both Alice and Bob edit the same lines in the `overview.txt` file at the same time. Try to force a conflict, and then carefully resolve the conflict.

3.4. Branches and Pull Requests

In this section, we describe branches and pull requests which are slightly more advanced topics but tremendously useful. Students could probably skim this section initially, and then revisit this information later in the semester. Branches and pull requests enable different students to work on different aspects at the project at the same time while keeping their commits separated in the remote repository on GitHub. So far, all of our work has been on the *master* branch. The master branch is the primary default branch. Creating additional branches can enable one student to work on a new feature while also fixing bugs on the master branch, or branches can enable students to experiment with some more advanced ideas but easily revert back to the "stable" master branch.

Let's say that Alice wants to work on a new list of animals in Alice and Bob's shared repository, but she wants to keep her work separate from the primary work they are focusing on. Alice can create a branch called `alice-animals` and commit her new ideas on that branch. It is usually good practice to prefix branch names with your NetID to ensure that branch names are unique. The following commands will first display the branches in the local repository using the `git branch` command before creating a new branch called `alice-animals`.

```
% cd ${ALICE}
% git branch
% git checkout -b alice-animals
% git branch
% git status
```

The `git branch` command uses an asterisk (`*`) to indicate the current branch. The `git status` command also indicates the current branch. Alice can now create a new file and commit her changes to this new branch.

```
% cd ${ALICE}
% git branch
% echo "cow"  > animals.txt
% echo "pig" >> animals.txt
% echo "dog" >> animals.txt
% git add animals.txt
% git commit -m "ALICE: initial animal list"
% git log --oneline --graph --decorate
```

The `--decorate` command line option with the `git log` command will show which commits are on which branch. It should be clear that the `alice-animals` branch is one commit ahead of the `master` branch. Pushing this branch to the remote repository on GitHub requires a slightly more complicated syntax. We need to specify which branch to push to which remote repository:

```
% cd ${ALICE}
% git push -u origin alice-animals
% cat animals.txt
```

The name `origin` refers to the remote repository that the local repository was originally cloned from (i.e., the remote repository on GitHub). You can now see this new branch on GitHub here:

- `https://github.com/<githubid>/ece4750-tut2-git/branches`

You can browse the commits and source code in the `alice-animals` just like the `master` branch. If Bob wants to checkout Alice's new branch, he needs to use a slightly different syntax.

```
% cd ${BOB}
% git pull --rebase
% git checkout --track origin/alice-animals
% git branch
% cat animals.txt
```

Alice and Bob can switch back to the `master` branch using the `git checkout` command.

```
% cd ${ALICE}
% git checkout master
% git branch
% ls
% cd ${BOB}
% git checkout master
% git branch
% ls
```

The `git branch` command should indicate that both Alice and Bob are now on the `master` branch, and there should no longer be an `animals.txt` file in the working directory. One strength of Git is that it makes it very easy to switch back and forth between branches.

Once Alice has worked on her new branch, she might be ready to merge that branch back into the `master` branch so it becomes part of the primary project. GitHub has a nice feature called *pull requests* that simply this process. To create a pull request, Alice would first go to the branch page on GitHub for this repository.

- `https://github.com/<githubid>/ece4750-tut2-git/branches`

She then just needs to click on *New pull request* next to her branch. *You must carefully select the base fork!* If you simply choose the default you will try to merge your branch into the repository that is part of the `cornell-ece4750` GitHub organization. Click on *base fork* and select *<githubid>/ece4750-tut2-git*. Alice can leave a comment about what this new branch does. Other students can use the pull request page on GitHub to comment on and monitor the new branch.

- `https://github.com/<githubid>/ece4750-tut2-git/pull/1`

Users can continue to develop and work on the branch until it is ready to be merged into `master`. When the pull request is ready to be accepted, a user simply clicks on *Merge pull request* on the GitHub pull request page. When this is finished the Git history for this example would look like this:

```
% cd ${ALICE}
% git pull
% git log --oneline --graph
*   f77c7f2 Merge pull request #1 from cbatten/alice-animals
|\
| * fe471e9 ALICE: initial animal list
* | 80765f3 BOB: added date to fruit list
|/
* 7393cac ALICE: added kiwi to fruit list
* 4c1fff6 ALICE: added kiwi to fruit list
* 1982bea BOB: added potato to vegetable list
* 5fd4d2e ALICE: added plum to fruit list
*   eaab790 Merge branch 'master' of github.com:cbatten/ece4750-tut2-git
|\
| * e44ab18 ALICE: added lemon to fruit list
* | 4a053a6 BOB: initial vegetable list
|/
* 2a431b0 BOB: added peach to fruit list
* 98ad45a ALICE: added banana to fruit list
```

★ *To-Do On Your Own:* Have Bob create his own branch for development, and then create a new file named `states.txt` with the names of states. Have Bob commit his changes to a new branch and push this branch to the remote repository on GitHub. Finally, have Alice pull this new branch into her local repository.

4. TravisCI for Continuous Integration

TravisCI is an online continuous integration service that is tightly coupled to GitHub. TravisCI will automatically run all tests for a students' lab assignment every time the students push their code to GitHub. We will be using the results reported by TravisCI to evaluate the code functionality of the lab assignments. In this section, we do a small experiment to illustrate how TravisCI works.

The first step is to enable TravisCI for the remote repository in GitHub. Login to TravisCI using your GitHub ID and password:

- `https://travis-ci.org/profile`

Once you have signed in, you should go to your TravisCI profile and find the list of your public GitHub repositories. You may need to click *Sync* to ensure that TravisCI has the most recent view of your public repositories on GitHub. Turn on TravisCI with the little "switch" next to the repository we have been using in this tutorial (`<githubid>/ece4750-tut2-git`). Figure 4 shows what the TravisCI settings page should look like and the corresponding "switch". After enabling TravisCI for the `<githubid>/ece4750-tut2-git` repository, you should be able to go to the TravisCI page for this repository:

- `https://travis-ci.org/<githubid>/ece4750-tut2-git`

TravisCI will report that there are no builds for this repository yet. TravisCI looks for a special file named `.travis.yml` in the top of your repository to determine how to build and test your project. We have already created one of those files for you, and you can see it here:

```
% cd ${TUTROOT}
% cat .travis.yml
```

The `.travis.yml` file for this tutorial is very simple. It just uses the `grep` command to check if the fruit `blueberry` is in the `fruit.txt` file. If the `blueberry` is present then the test passes, otherwise the test fails. Let's add `melon` to the list of fruits in our local repository and then push the corresponding commit to the remote repository on GitHub.

```
% cd ${TUTROOT}
% git pull --rebase
% echo "melon" >> fruit.txt
% git commit -a -m "added melon to fruit list"
% git push
```

Notice how we first use the `git pull` command to ensure our local repository has all of the commits from the remote repository on GitHub. Revisit the TravisCI page for this repository. You should be able to see the build log which shows TravisCI running the `grep` command. The test should fail

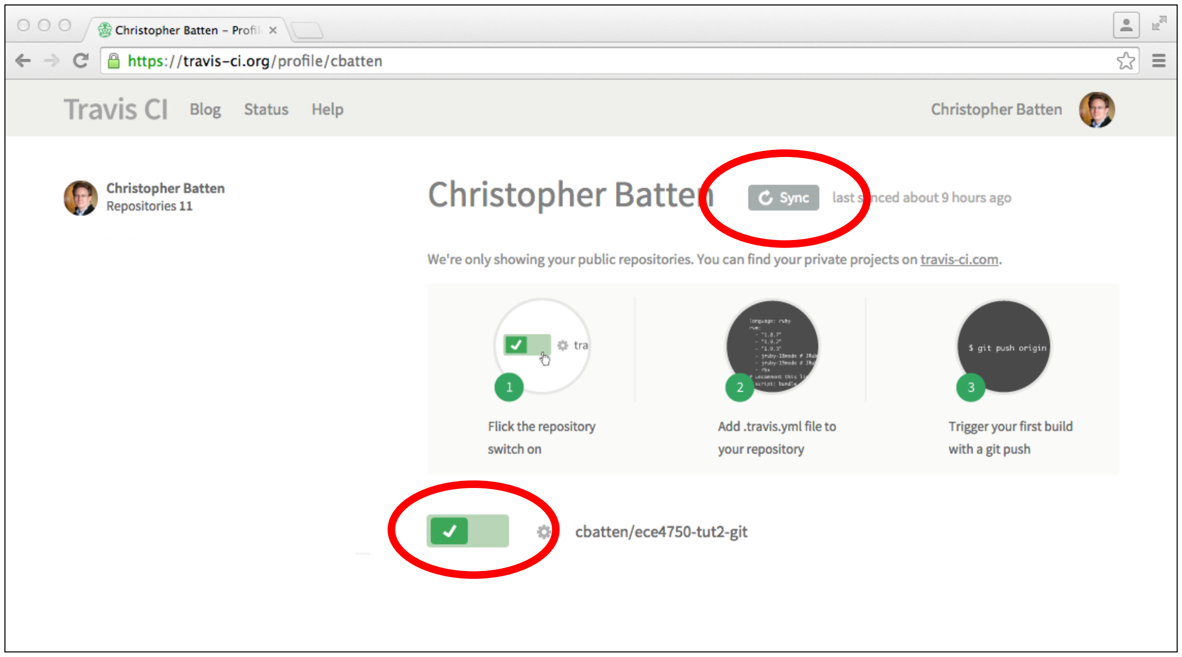
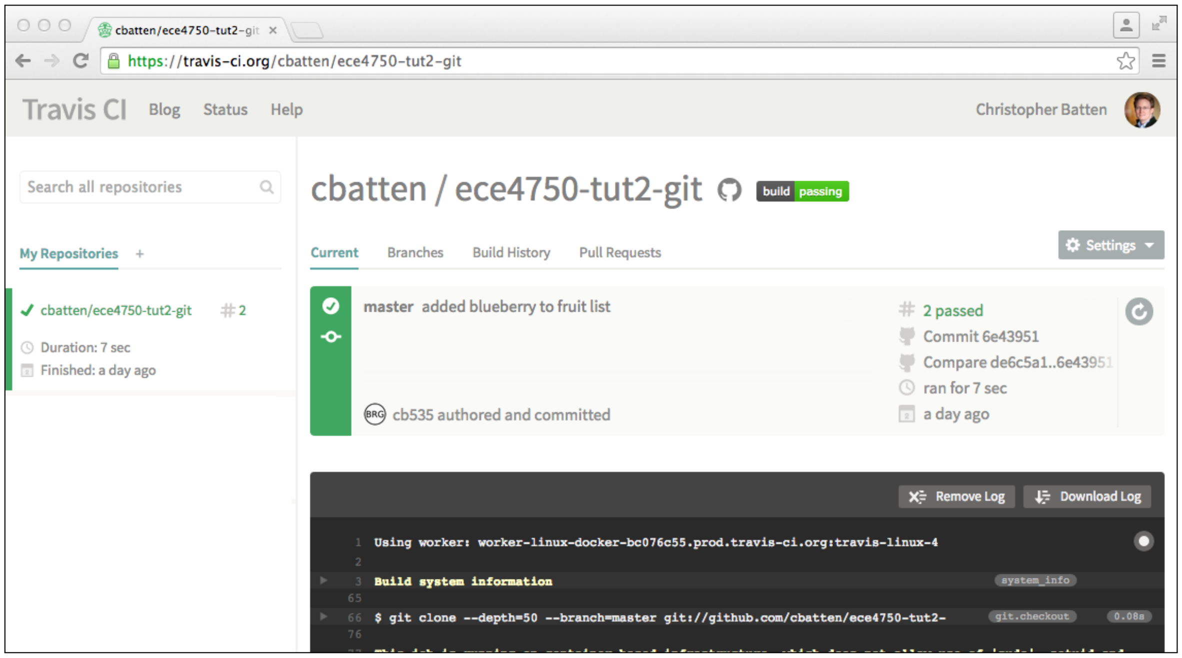

Christopher Batten – Profi
https://travis-ci.org/profile/cbatten
Travis CI
Blog
Status
Help
Christopher Batten
Christopher Batten
Repositories 11
Christopher Batten
Sync
last synced about 9 hours ago
We're only showing your public repositories. You can find your private projects on travis-ci.com.
1
Flick the repository switch on
2
Add .travis.yml file to your repository
3
Trigger your first build with a git push
$ git push origin
cbatten/ece4750-tut2-git

Figure 4: TravisCI Settings Page



```
Using worker: worker-linux-docker-bc076c55.prod.travis-ci.org:travis-linux-4

Build system information                                    system_info

$ git clone --depth=50 --branch=master git://github.com/cbatten/ece4750-tut2-   git.checkout   0.08s
```

Figure 5: TravisCI Build Page for the `ece4750-tut2-git` Repository

because `blueberry` is not currently in the `fruit.txt` file. Now let's add `blueberry` and then push the corresponding to commit to trigger another build on TravisCI.

```
% cd ${TUTROOT}
% echo "blueberry" >> fruit.txt
% git commit -a -m "added blueberry to fruit list"
% git push
```

If you revisit the TravisCI page for this repository, you should see something similar to Figure 5. The TravisCI shows the status of the build as well as all of the output from the build. You should be able to see the test on TravisCI passing. TravisCI display the complete build history so you can see exactly which commits pass or fail the test suite, and you can also see the build status for branches and pull requests:

- `https://travis-ci.org/<githubid>/ece4750-tut2-git/branches`
- `https://travis-ci.org/<githubid>/ece4750-tut2-git/builds`
- `https://travis-ci.org/<githubid>/ece4750-tut2-git/pull_requests`

By ensuring that all of the tests pass for a branch or pull request, students can always be confident that merging those branches or pull requests into the `master` branch will not break anything.

Using TravisCI to perform continuous integration testing is a key component of an agile development methodology. It means the entire group can quickly spot commits which break certain tests, and always be certain that their `master` branch is passing all tests before submitting the lab assignment. The course staff will actually be using TravisCI to grade your lab assignments. The staff will be able to look at the build log in TravisCI to see if your assignment is passing your own test suite, and then the staff can add more tests to see if your assignment passes a more exhaustive test suite.

★ *To-Do On Your Own:* Edit the `.travis.yml` file to search for `apple` instead. Experiment with removing and adding `apple` from the `fruits.txt` file to see the tests on TravisCI pass and fail.

5. ECE 5745 Git Scripts

This section describes some useful scripts we have installed that make it easier to use Git. Each script is invoked by typing the name along with the standard `git` command.

5.1. Using `git xstatus` to Compactly See Status Information

The `git xstatus` command produces a status output somewhat similar to subversion's status command. It first shows the status of all tracked files which are modified, deleted, or added, then shows the status of all files in the index (marked with an asterisk), and finally shows which files and directories are untracked. Here is an example output:

```
% cd ${TUTROOT}
% echo "cyan" >> colors.txt
% echo "rabbit" >> animals.txt
% git add colors.txt animals.txt
% git commit -m "added some colors and animals"
% echo "grape" >> fruit.txt
% git add fruit.txt
% echo "strawberry" >> fruit.txt
% echo "bird" >> animals.txt
% echo "tulip"  >> flowers.txt
% rm colors.txt
% git xstatus
 M animals.txt
 D colors.txt
 M fruit.txt
*M fruit.txt
 ? flowers.txt
```

This shows that the file `colors.txt` has been deleted from the working directory, but this deletion has not been added to the index yet (`colors.txt` is not listed with an asterisk). The file `animals.txt` has been modified buy not added to the index yet. Note that the file named `fruit.txt` has been modified and added to the index, but it has been modified *since* it was added to the index as indicated by its double listing. The file named `flowers.txt` is currently untracked.

The possible status codes are as follows:

```
- A : addition of a file
- C : copy of a file into a new one
- D : deletion of a file
- M : modification of the contents or mode of a file
- R : renaming of a file
- T : change in the type of the file
- U : file is unmerged (you must complete the merge before commit)
- ? : file is untracked (you need to add it if you want to track it)
```

5.2. Using `git xadd` to Add All Changed Files

The `git xadd` command adds all files which are currently tracked and display the new status in a format similar to the `git xstatus` command. You can use this to quickly stage files for commit and see what would be committed before actually executing the commit.

```
% cd ${TUTROOT}
% git xstatus
% git xadd
```

5.3. Using `git xlog` to Compactly See Log Information

The `git xlog` command displays a compact log format with one commit per line and a graph representing the commit history. This script passes along whatever the additional options are included straight onto `git log`. Here is a simple example of the log output.

```
% cd ${TUTROOT}
% git xlog
* aa16ecb  Christopher Batten  added some colors and animals
* 85d90f7  Christopher Batten  added another fruit
* 7ec7919  Christopher Batten  added some colors and animals
*   05ffcfa  Christopher Batten  merged all fruit into a single list
|\
| * 2f80e5a  Christopher Batten  added lemon to fruit list
* | 6ee31c6  Christopher Batten  added plum to fruit list
|/
* 4270ddc  Christopher Batten  added peach to fruit list
* 7717c10  Christopher Batten  added banana to fruit list
* 374b5a0  Christopher Batten  added orange to fruit list
* 8efa69c  Christopher Batten  added mango to fruit list
```

You can see one line per commit along with the commit hash, the committer's name, and the short commit message. The graph shows a merge between commits `2f80e5a` and `6ee31c6`.

5.4. Using `git xpull` to Pull and Rebase

Since we strongly recommend students use rebasing instead of merging, we have provided a `git xpull` command. This command has the exact same effect as using the `--rebase` command line option with the `git pull` command.

```
% cd ${TUTROOT}
% git xpull
```

6. ECE 4750 Lab Admin Script

We have created a dedicated GitHub *organization* for the course located here:

- `https://github.com/cornell-ece4750`

Students will work in groups of three students to complete the lab assignments. Each group will have a corresponding GitHub *team* within the GitHub course organization. Each group has read/write permission to its own lab assignment remote repository on GitHub. Students will be able to access their own lab assignment repository, but will not be able to access the lab assignment repositories of other groups. All lab assignments will be completed within this single repository.

We have created a lab administration script called `ece4750-lab-admin` to simplify creating, joining, and leaving lab groups. Note that the `ece4750-lab-admin` script should be run on an `ecelinux` machine after sourcing the course setup script. You can use the `ece4750-lab-admin` script to submit *lab administration requests*. Once you submit a request it will be pending for a minute or so until the request is accepted by the system. It will take another five minutes or so for any changes to appear on GitHub. Please be patient.

6.1. Using `ece4750-lab-admin` to Join GitHub Course Organization

The first request you will want to submit is to actually join the GitHub course organization. You can do that using the `--join-class` command line option.

```
% ece4750-lab-admin --join-class <githubid>
% ece4750-lab-admin --status
```

where `<githubid>` is your GitHub ID. Use the `--status` command line option to see if you have any pending requests, and also to check the status of your lab group. You can use the `--status` command line option repeatedly to see when your pending request is accepted. After the `join-class` request is accepted by the system, you will receive an email from GitHub in about five minutes asking you to accept an invitation to join the GitHub course organization. Follow the instructions. If you do not receive this email after 10 minutes or so, you might want to try simply visiting the page for the course organization:

- `https://github.com/cornell-ece4750`

If you have been invited, then you should see an invitation at the top of the page that you can click on. Once you are a member of the GitHub course organization you can submit other requests.

6.2. Using `ece4750-lab-admin` to Form a Lab Group

Once you have selected your lab group, you can use the `ece4750-lab-admin` script to create the corresponding GitHub team within the GitHub course organization. One student should start by submitting a `group-create` request.

```
% ece4750-lab-admin --make-request group-create
% ece4750-lab-admin --status
```

Once this request has been accepted by the system, the `--status` command line option should display the group's assigned group number. The other students in the group should use this group number to join the group by submitting `group-join` requests:

```
% ece4750-lab-admin --make-request group-join <groupnum>
% ece4750-lab-admin --status
```

where `<groupnum>` is the group number assigned through the original `group-create` request. Although any student can request to join a group, that student will only be added to the corresponding group if a current member in the group approves. So once the `group-join` requests are submitted, a student already in the group should submit a `group-approve` request. Joining a group only requires one student's approval.

```
% ece4750-lab-admin --make-request group-approve <netids>
% ece4750-lab-admin --status
```

where `<netids>` is the NetIDs of the student trying to join the group. If you want to approve multiple students, you should seperate NetIDs by commas. For example:

```
% ece4750-lab-admin --make-request group-approve ab123,cd456
% ece4750-lab-admin --status
```

All students can use the `--status` command line option to determine when all of these requests have been accepted and the group has been fully formed. The students should then wait about five minutes for the corresponding team to be created on GitHub, at which point the students can visit the GitHub page for their lab assignment repository to verify everything is working:

- `https://github.com/cornell-ece4750/lab-groupXX`

where `XX` is your group number.

6.3. Using `ece4750-lab-admin` to Change Lab Groups

You can also use the `ece4750-lab-admin` script to change groups. A student can only be a member of a single group at a time, so the first step is to leave your current group using the `group-leave` request.

```
% ece4750-lab-admin --make-request group-leave
% ece4750-lab-admin --status
```

Once this request has been accepted by the system, the student can now create a new group or join an existing group using the steps described in the previous section. You should be very careful with this command, once you left your current group, you will need approval from a group member to re-join.

7. Conclusion

This tutorial hopefully helped you become familiar with Git and how to use it for version control. You should also now feel comfortable using the `ece4750-lab-admin` script for managing lab groups. There are many other commands you might want to use for "rolling-back" to previous versions, managing branches, and tagging specific commits. Students are encouraged to read more on Git and GitHub. Keep in mind that learning to use Git productively can pay dividends in many other contexts besides this course.

Acknowledgments

This tutorial was developed for ECE 4750 Computer Architecture course at Cornell University by Shreesha Srinath and Christopher Batten. It has been adapted for ECE 5745.