

Lecture 5: Collaborative Programming and Basic HTML

Collaborative Programming

How do we code collaboratively?

Learning to code in collaboration with others is a very important skill to learn for your life as a developer.

We can code collaboratively by using a combination of version control software, ticketing systems, and workflows to distribute work amongst our team members.

What is version control?

Version control is a system that records sets of changes over time. It allows you to keep track of a history of your files in a very detailed manner.

Version control allows you to see a detailed log of all your changes, as well as roll changes back. It allows you a great deal of control over the state of your files; you can make many, many versions of your code and change between the versions with ease!

- This allows for easy feature development.

What is Git?

Git is an extremely popular version control software that is commonly used both personally and professionally. It is the most widely used version control software at the current point in time.

It was originally created by Linus Torvalds, who happens to also be the creator of the Linux Kernel.

Git is a distributed version control system: everyone will make a copy of a codebase to work off of on their local machine, and changes on one machine will not affect changes on another machine.

How can we use Git?

Git is a command line program, so we will use it from our terminal (much like node)

- <https://git-scm.com/>

Most of our commands are simple, and we will use Git to work collaboratively.

We will, in general, use Git by setting up a centralized repository that we will publish code to and we will synchronize changes in our local repositories.

For most of our work, we will be using Github to store an online copy of our repository.

Distributing Work

The easiest way to work in a team is to distribute work.

Very often, work is distributed in one of the following forms:

1. Feature based; each team member owns a portion of the features and does all the work for that area. This includes: server routes, data code, HTML, CSS, and frontend JavaScript
2. Architectural ownership; each team member takes a region of the code to own. One user will work on authentication, one will work on routes, one will work on CSS, one on JS, etc.
3. Ticket based; a series of tickets are created regarding each issue, bug, feature, etc., and developers claim tickets to work on.

Using a ticket system

Ticketing systems are a popular way to keep track of issues and features. Project managers often use this software to manage the team workload. You may find this useful for your projects.

Some popular ticket systems / project management apps:

- Asana
 - <http://asana.com/>
- Trello
 - <http://trello.com/>
- Github issues
 - <https://developer.github.com/v3/issues/>
- Mingle
 - <https://www.thoughtworks.com/mingle/>
- Waffle
 - <https://waffle.io/>

Git

Terms

Term	Meaning
Repository	A repository is a location that stores the information about the project's file and folder structure, as well as its history
Branch	A branch is a pointer to a certain chain of file change histories; you can have many branches, but will always have at least one. Traditionally, the original branch is called <i>master</i>
Commit	A commit is a snapshot of your repository at a point in time.
Remote	A reference to a repository stored outside of your current local machine; ie, the repository on Github
Push	Pushing is the act of taking your commits and uploading them to a remote repository
Pull	Pulling is the act of taking commits from a remote repository and bringing the changes down to your local repository
Merging	Merging is the act of bringing one set of changes from one branch to another, and creating a new version of the code with both histories.
Pull request	A pull request is a request to bring a series of changes from one branch to another

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner Repository name

Great repository names are short and memorable. Need inspiration? How about **improved-happiness**.

Description (optional)

☒ **Public**
Anyone can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

☐ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **Node** | Add a license: **MIT License**

Create repository

Making a repository on Github

The first step to using Git is making a repository. A repository is a data structure that stores information about the files and folder of a project, as well as the history of the structure.

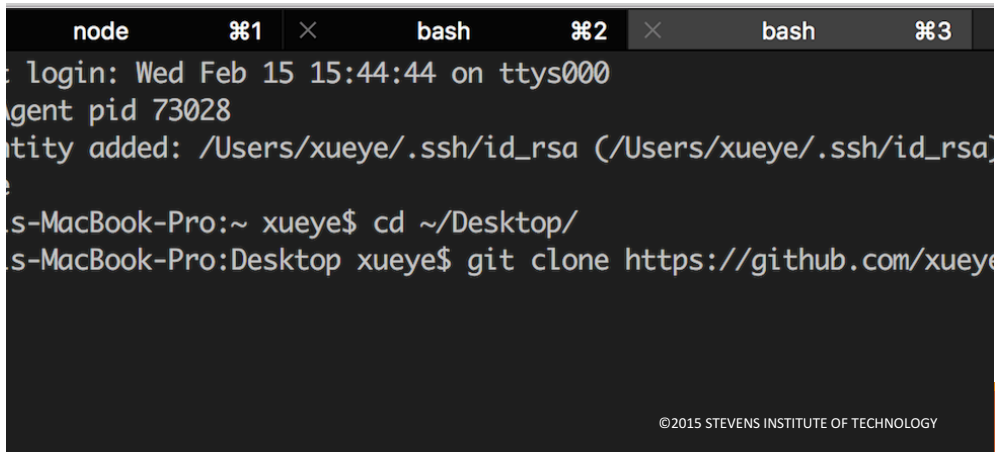
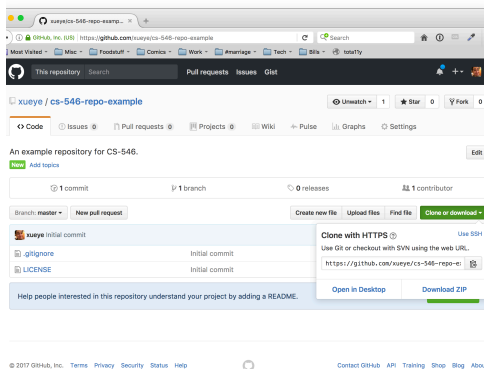
The easiest way to make a repository is to start on Github, with the following settings:

- <https://github.com/new>

Cloning a repository

Once the repository is created on Github, we can clone it to our local machines using the git command line.

On the home page of a repository ([see example](#)) there will be a button to clone or download the repository. Copy the url, and use the *git clone* URL command to clone the repository.

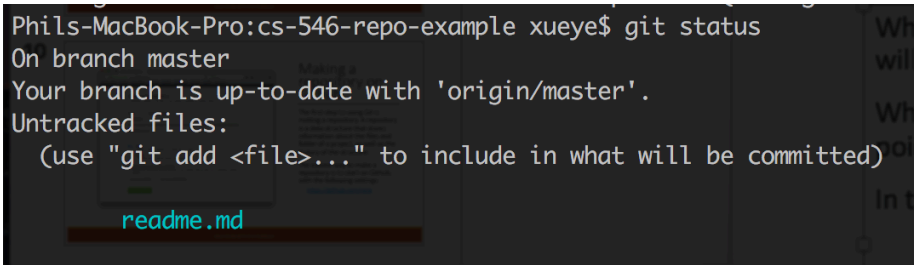


Getting status

While in our project directory, we can make changes as needed to our repository. These changes will not appear online until we commit and push these changes.

When we want to see what changes we have since our last commit (a snapshot of our code at a point in time), we can use the *git status* command.

In the screenshot, we can see that we have an untracked file; this means a file that is brand new to the repository.

A terminal window screenshot showing the output of the 'git status' command. The prompt is 'Phils-MacBook-Pro:cs-546-repo-example xueye\$'. The output indicates the user is on the 'master' branch and it is up-to-date with 'origin/master'. It lists 'Untracked files:' and shows 'readme.md' in red text. A hint suggests using 'git add <file>...' to track the file.

```
Phils-MacBook-Pro:cs-546-repo-example xueye$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    readme.md
```

Adding changes to be included in a commit

```
Phils-MacBook-Pro:cs-546-repo-example xueye$ git add readme.md
warning: LF will be replaced by CRLF in readme.md.
The file will have its original line endings in your working directory.
Phils-MacBook-Pro:cs-546-repo-example xueye$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   readme.md
```

Before we commit changes, we need to explicitly tell Git which files we want to track the changes of. We do this with the *git add* *PATH/TO/FILE.EXT* command.

For example, to add our new readme, it would be: *git add readme.md*

After the file is added, it is now in a state known as *staging*; this means that it's a change that will be included in a new commit. We can continue to edit this file and the new changes will not be included in that staging version of the file until we explicitly run *git add* again.

Committing Code

```
Phils-MacBook-Pro:cs-546-repo-example xueye$ git commit -m "Added readme to repository"
[master 8fc22c3] Added readme to repository
 1 file changed, 1 insertion(+)
 create mode 100644 readme.md
Phils-MacBook-Pro:cs-546-repo-example xueye$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
nothing to commit, working tree clean
Phils-MacBook-Pro:cs-546-repo-example xueye$
```

Once we make changes that we want to group together and store in our repository, we will perform what is known as a commit. A commit can be seen as a snapshot of our repository at a certain point in time.

Committing is very easy: *git commit -m "Our message here"* will snapshot all **staged** changes and make a commit with those changes.

Pushing up our changes

Version control is incredibly useful when using a single computer, but shines even more when we push our changes up to a remote repository. By default, when we clone, a reference to the online repository will be created; this repository is named *origin*.

We can push our changes by using the *git push* *REPOSITORY_NAME* *BRANCH_NAME*, like *git push origin master*

After pushing, our commits will be seen online

- Commits on Feb 16, 2017



Added readme to
xueye committed 8 mi



Initial commit
xueye committed 17 m

```
Phils-MacBook-Pro:cs-546-repo-example xueye$ git push
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 336 bytes | 0 bytes/s,
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/xueye/cs-546-repo-example.git
5d5d6fd..8fc22c3 master -> master
Phils-MacBook-Pro:cs-546-repo-example xueye$
```


Pulling changes

We can pull down changes in the same manner, by issuing a *git pull* *REPOSITORY_NAME* *BRANCH_NAME* command. If there are changes, you will automatically pull and merge these changes into the branch you are currently in.

```
Phils-MacBook-Pro:cs-546-repo-example xueye$ git pull origin master
From https://github.com/xueye/cs-546-repo-example
* branch      master      -> FETCH_HEAD
Already up-to-date.
```

An easy workflow

Let us pretend for a moment that we want to make a series of updates to our readme file. A common workflow is:

- Make a new branch devoted to all changes to the readme, such as *git checkout -b feature/readme*
- Make relevant updates to the codebase
- Make as many commits as needed until satisfied with the result
- As you commit, push your changes up to a remote branch; the first time you push to a new branch online, it will be created online!
- When done with the feature, issue a pull request for the code to be reviewed.
- When the code is reviewed and accepted, merge it into the main branch.

Workflow

Demonstration: Branching and saving changes

At this point, we have created a new branch using the *git checkout -b feature/readme* command; this creates a new branch named *feature/readme*; if we already had that branch and were just moving to the branch, we would omit the *-b* flag.

```
Phils-MacBook-Pro:cs-546-repo-example xueye$ git checkout -b feature/readme
Switched to a new branch 'feature/readme'
Phils-MacBook-Pro:cs-546-repo-example xueye$ nano readme.md
Phils-MacBook-Pro:cs-546-repo-example xueye$ git status
On branch feature/readme
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.md
```

Workflow Demonstration: Staging and Committing

On our new branch, we now stage the files by adding them to be committed; then, we commit the code.

```
Phils-MacBook-Pro:cs-546-repo-example xueye$ git add readme.md
warning: LF will be replaced by CRLF in readme.md.
The file will have its original line endings in your working directory.
Phils-MacBook-Pro:cs-546-repo-example xueye$ git status
On branch feature/readme
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   readme.md

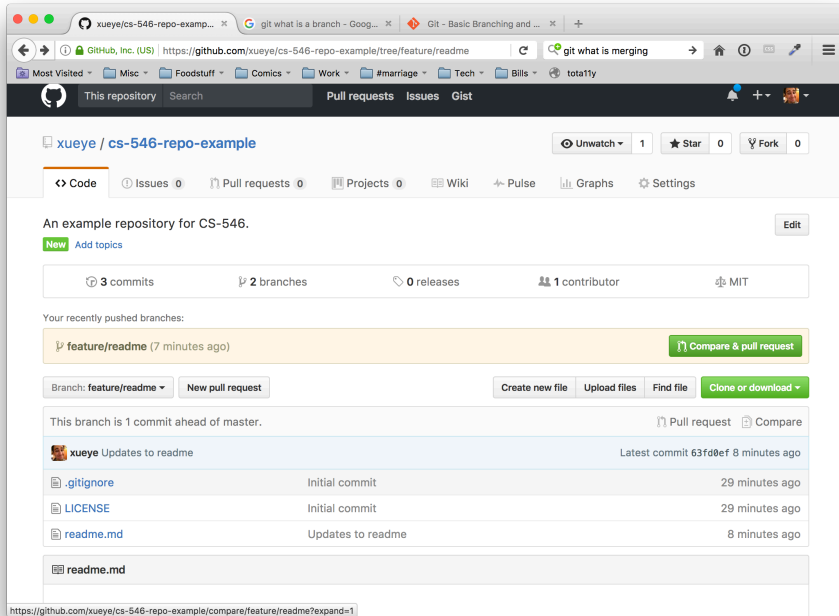
Phils-MacBook-Pro:cs-546-repo-example xueye$ git commit -m "Updates to readme"
[feature/readme 63fd0ef] Updates to readme
1 file changed, 4 insertions(+)
```

Workflow

Demonstration: Pushing code

We push the code online to our remote repository; we should remember to do this often!

```
Phils-MacBook-Pro:cs-546-repo-example xueye$ git push origin feature/readme
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 365 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local objects.
To https://github.com/xueye/cs-546-repo-example.git
 * [new branch]      feature/readme -> feature/readme
Phils-MacBook-Pro:cs-546-repo-example xueye$
```

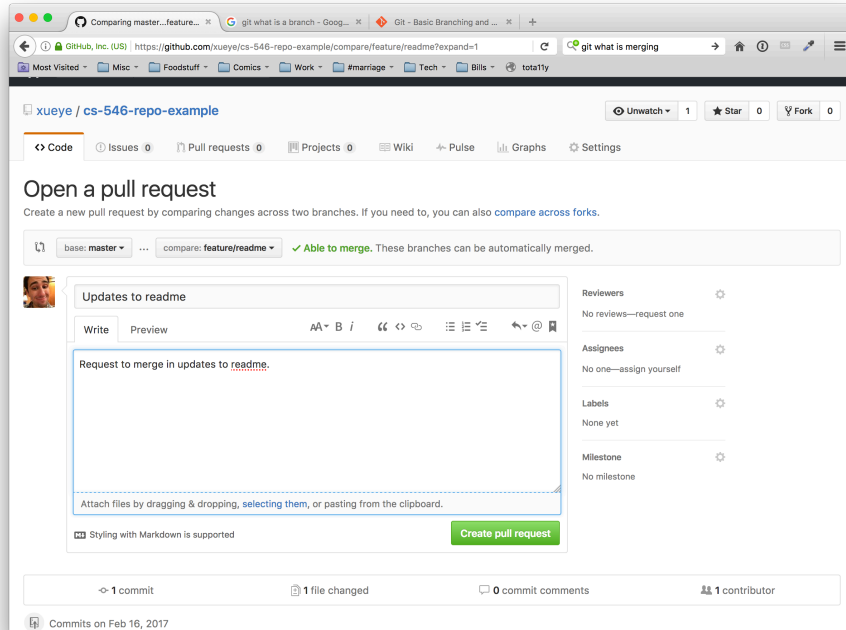


Making a pull request

On Github, we can create a pull request based on the branch. This will issue a request to have this changes merged into a different branch (in our case, *master*).

Making a pull request

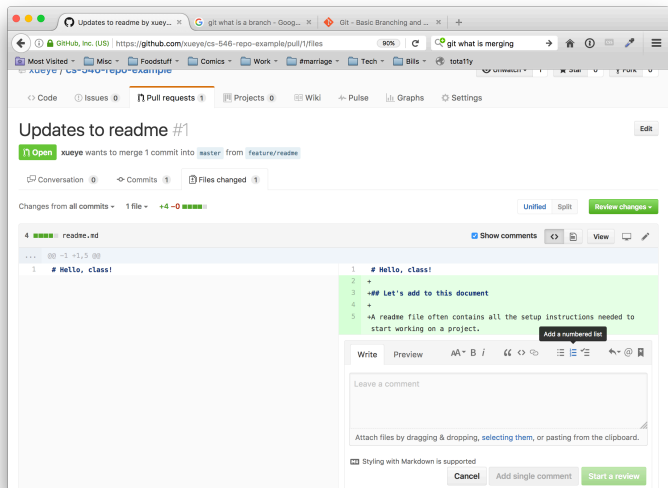
Pull requests should have a description of the set of changes, and when created will show the changes in that branch.



Reviewing a pull request

All pull requests should be reviewed by a different developer (and it's also good to have it reviewed by more than one developer); by clicking the *files changed* tab on the pull request page, we can leave comments to the developer about their pull request.

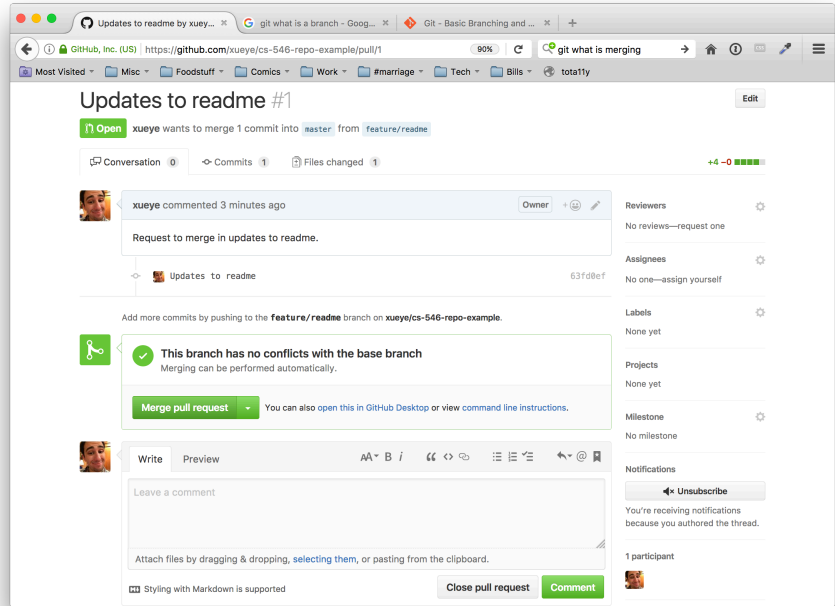
This is very useful for identifying inefficiencies, logical errors, bugs, etc. It also forces multiple developers to look at the approach to solving a problem or making a feature, so that more than one developer is familiar with each portion of the code in a project.



Merging in the pull request

If a pull request can be safely merged in without conflicts occurring, Github will allow you to merge in the changes when all developers are satisfied with the changes.

This will add the new commit data to the master branch of the online repository.



Pulling changes

At this point, the only repository with the new commit on the *master* branch is the online version; even the repository that we developed on does not have those changes on the master branch.

We need to routinely pull from the *master* branch to stay up to date.

```
Phils-MacBook-Pro:cs-546-repo-example xueye$ git pull origin master
remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), done.
From https://github.com/xueye/cs-546-repo-example
* branch      Convers master      <--> FETCH_HEAD 3 Files changed  1
   8fc22c3..0d8099c master        -> origin/master
Updating 8fc22c3..0d8099c
Fast-forward
  readme.md | 4 +++++
  1 file changed, 4 insertions(+)
```

Avoiding Issues

Many issues can occur when using version control, due to the nature of many people editing the same sets of files.

There are a few easy tricks to avoiding most common issues with Git:

- **Never develop on the master branch**; do all development in your own feature branches, and issue pull requests.
- **Pull master into your own feature branches commonly**; merge errors will occur that you will need to resolve by hand, but you will ultimately have to resolve these issues far less than if you were all working on the master branch
- **Isolate your work into small chunks**; do not wait to do a whole feature before you commit. Commit often, as you accomplish small, incremental changes.
- **Make new feature branches off of master**; master should always be the most up-to-date **working** code; it is prudent when starting a new feature to get an updated version of the master branch and make a new branch from that up-to-date master branch.
- **Pull often**; this is so important, that we're listing it twice. **Pull often!**

References

<http://rogerdudler.github.io/git-guide/>

<https://git-scm.com/doc>