Git and GitHub - Collaboration

LESSON

Generate a quiz about this unit

Summarize this unit

Have any doubt about this content? Ask!

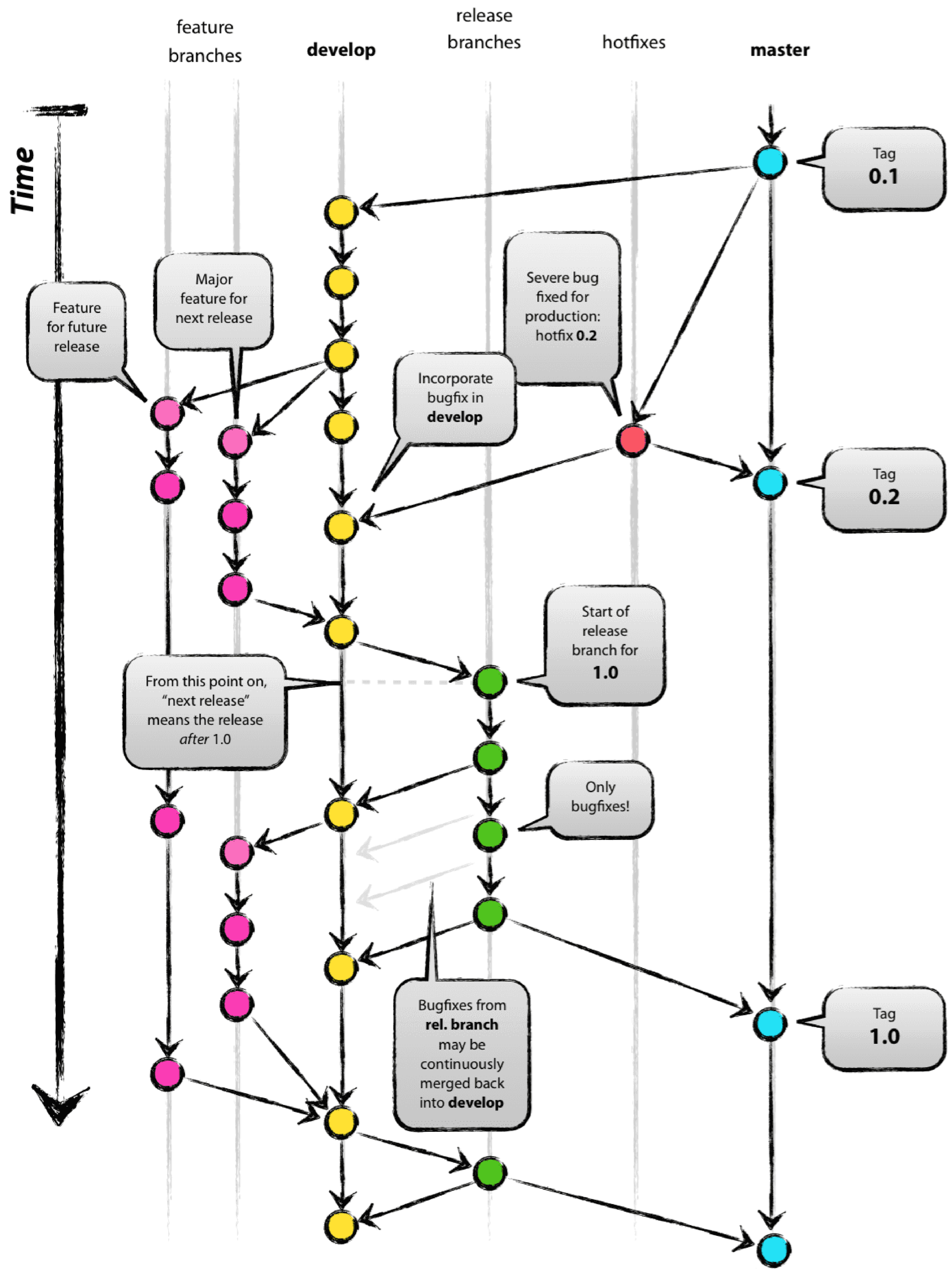
Learning goals

After this lesson, you will be able to:

- Create a new git branch
- Switch between branches
- Commit changes to a branch other than `master`
- Merge two branches together
- Solve Merge Conflicts
- Fork a repository
- Make a pull request

Introduction

Before we get started, let’s take a look at a diagram to illustrate an example of industry-standard best-practice workflow with git and GitHub. In this case, the diagram shows how **git flow** organizes its workflow process:


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Severe bug fixed for production: hotfix 0.2
Major feature for next release
Feature for future release
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

What could be simpler?

Okay, maybe it doesn't appear to be that simple. But all jokes aside, it's not as complicated as it looks. In this diagram, the Y axis is time, and time passes by as you move down through the diagram.

Each of the lines running vertically is a git branch. This diagram basically shows that, in a professional team setting, using git is not as simple as pushing everything to your `master` branch. Instead, `master` branch is used only for deploy purposes, while all the development takes place on other branches.

The different branches we have in the diagram are used for the following:

- `master` contains a complete version of the website, without any bug, and it's used to deploy the last version of our website to production.

- If we find a bug in the `master` branch, we will create a `hotfix` branch to solve it and push it into development as fast as we can.
- `develop` is a copy of the `master` branch, and over it will contain all the features we have already implemented in a development environment.
- `feature` branches are created to implement each new feature in our product. They help us keep a clean branch to do as many changes as we need to implement a feature. Once a feature is implemented, we merge this branch into `develop`.
- Last, but not least, the `release` branch includes all the changes that we have done in `develop`, and it will be deployed in an environment to try them all and be sure they are working. Once the release branch has been checked out, we can merge it into `master` and deploy it to production.

So in order to do something like this, you'll need to master the art of branching in git.

Create A Git Repository

Before we start making git branches, first, we'll need to create a git repository. Let's begin by making a folder on our desktop and navigating into it with the terminal (you used the terminal to create the folder in the first place, didn't you?).

Once inside the folder, run `git init` to initialize a git repository in that folder:

git init
We use `git init` to initialize a git repository:

```
$ mkdir my_folder
$ cd my_folder
$ git init
```

Copy

Explain this code

Add & Commit Some Changes

Before we do anything else, run `git status` to make sure you set up the git repository correctly. It should say something like `nothing to commit`. Now, let's make a change to our repository so we can add & commit the change to git:

```
$ touch sample.txt
$ open sample.txt
```

Copy

Explain this code

The file should open. Write some text in the file and save it. You can write whatever you want but for clarity's sake, you might want to write something like `Hello, from the master branch!` so we can tell which branch this was written in. Now if we run `git status` again, it should say something like this:

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        sample.txt
```

Copy

Explain this code

It means you're on the right track. Git is tracking everything that is happening in this folder and it knows that you just created a new file that git is not tracking yet.

You can add this file to the staging area by running `git add sample.txt` or you can run `git add .` to add every file in the folder to the staging area. (Right now we only have one file so they will do the same thing.)

Once you have added the file, if we run `git status` again we should see something like this:

Copy

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   sample.txt
```

Explain this code

Now that you've added the file, run `git commit -m "Added a random text file"` to commit your file and be able to push it into Github. The commit message must be helpful, so we can remember what we did on this commit.

git status
We use `git status` to see which files we have tracked in our repository:

```
$ git status
```

Copy

Explain this code

💡 **You can also use `gst` as a shortcut for `git status`.**
git add
We use `git add` to add files to the next commit we are going to do.
git commit
We use `git commit` to commit all the tracked files into our git repository.

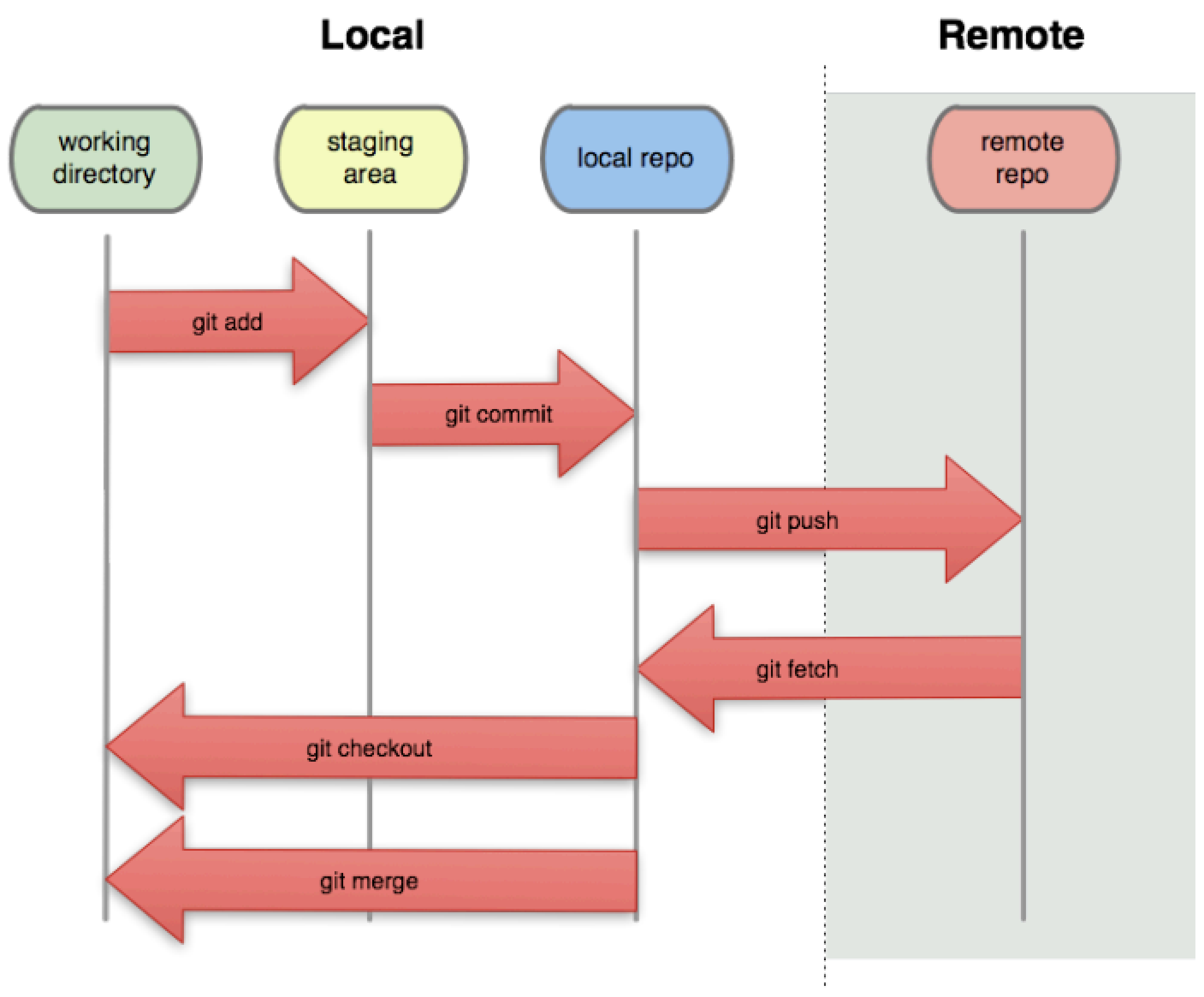
Working with Github

It can be confusing to discern between git and github. When people start programming, a lot of them don't know exactly which differences are between git and github.

Git is a software repository that you install in your local machine to track the different versions of your code you can have. What happens if our computer dies suddenly? Why should we track all the changes locally, if we can lose all our data?

Here is where Github plays an important role. Github is a remote software repository. You can think about Github like a copy of your local repositories on the Internet. Github also allow our team to work over the same software, sharing all the files, commits, versions, branches, etc.

In the following diagram, you can see how adding and committing changes works in git and github. Local refers to git, while Remote refers to Github.



As you can see, adding and committing are processes that take place on your local machine and do not make use of an internet connection. It is only the `git push` stage that will sync your Github repository with what is on your local machine. Similarly, `git fetch` will take information from Github, and put it on your local machine. `git checkout` and `git merge` will be covered shortly.

Create A New Branch

Now that we've committed something to the master branch, let's make a new branch so we can make some conflicting changes. (Yay, conflict!)

- Run `git branch <branch-name>` to create a new branch in your project.
- Run `git checkout <branch-name>` to change the current branch and start using the new one we just create.
- Run `open sample.txt` to open the file again, and see the message we originally added.
- Now, delete the text in the file and replace it with `hello from the new branch`. Make sure you save the file and close it.

Now if you run `git status` you should see something like this:

```
On branch newBranch
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:   sample.txt
```

Copy

Explain this code

- Go ahead and add, then commit these changes. **If you don't commit the changes, you won't be able to switch to another branch.**
- Now, let's switch back to the master branch `git checkout master`.
- Now that we're on the master branch, run `open sample.txt` what happens?

It has gone back to saying `hello from the master branch`. But we changed it, right? Where did the other message go?

Well, if we close the file and run `git checkout <branch-name>` and then we open the `sample.txt` file again, we'll see that on this branch, the file still has the other message. **We have a different version of the document on each of the 2 branches**.

So let's go back to the master branch and merge our changes from `<branch-name>` into `master` so we can see how that works.

- Run `git checkout master` so that we're back on the master branch.
- Then, run `git merge <branch-name>`. When merging with git, always navigate first to the destination branch, and then merge another branch into it, like you are importing it.
- Now, if you run `open sample.txt` it should say `hello from the new branch` because git has looked at both versions, and you've told it that you want to take the changes you've made in the new branch and incorporate them into the master branch. So at this point, the 2 branches should be equal.

git checkout
We use `git checkout` to switch between branches:

```
$ git checkout <destination-branch>

```

Copy

Explain this code

💡 **You can also use `gco` as a shortcut for `git checkout`.**
git checkout -b
We user [`git checkout -b <branch-name>`] to create a new branch and automatically switch to the new branch.

```
$ git checkout -b <new-branch>

```

Copy

Explain this code

Merge Conflicts

So... what happens when merging doesn't go quite so smoothly? Let's take a look at merge conflicts and how to resolve them.

- First of all, navigate to the master branch with `git checkout master`.
- Open the `sample.txt` file.
- Add some other text into the file. Don't erase what you had before, but add some more stuff. We can add, for example, a to do list:

```
to do:
eat
sleep
code
repeat
```

Copy

Explain this code

- Now add and commit the changes you just made.
- Once you've done that, save and close the file, then switch to the other branch.
- Open the file again. It should say `hello from the new branch` but all the changes you just made should be missing.
- Make some changes to this version that are **different** than the changes you made on `master`.

```
to do:
watch TV
play video games
eat pizza
```

Copy

Explain this code

- Save these changes, then close the file and add, then commit the changes you've made.
- Now, navigate back to the master branch. If you open the file, it should have the original to do list you made, the one without pizza.
- Now, close the file and run `git merge newBranch` You should see something like this:

```
Auto-merging sample.txt
CONFLICT (content): Merge conflict in sample.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Copy

Explain this code

Now we have to open the file and resolve the conflicts. So run `open sample.txt`, and you will find something like this:

```
hello from the new branch

to do:
<<<<<<< HEAD
eat
sleep
code
repeat
=======
watch TV
play video games
eat pizza
>>>>>>> newBranch
```

Copy

Explain this code

So, here, git is telling us that it cannot figure out what you want on this file because on one branch you changed it one way, but on another branch, you changed it a different way.

The === separates one version from the other. Pick which version you would like to keep and delete the other version, or, keep a combination of the two. It's up to you.

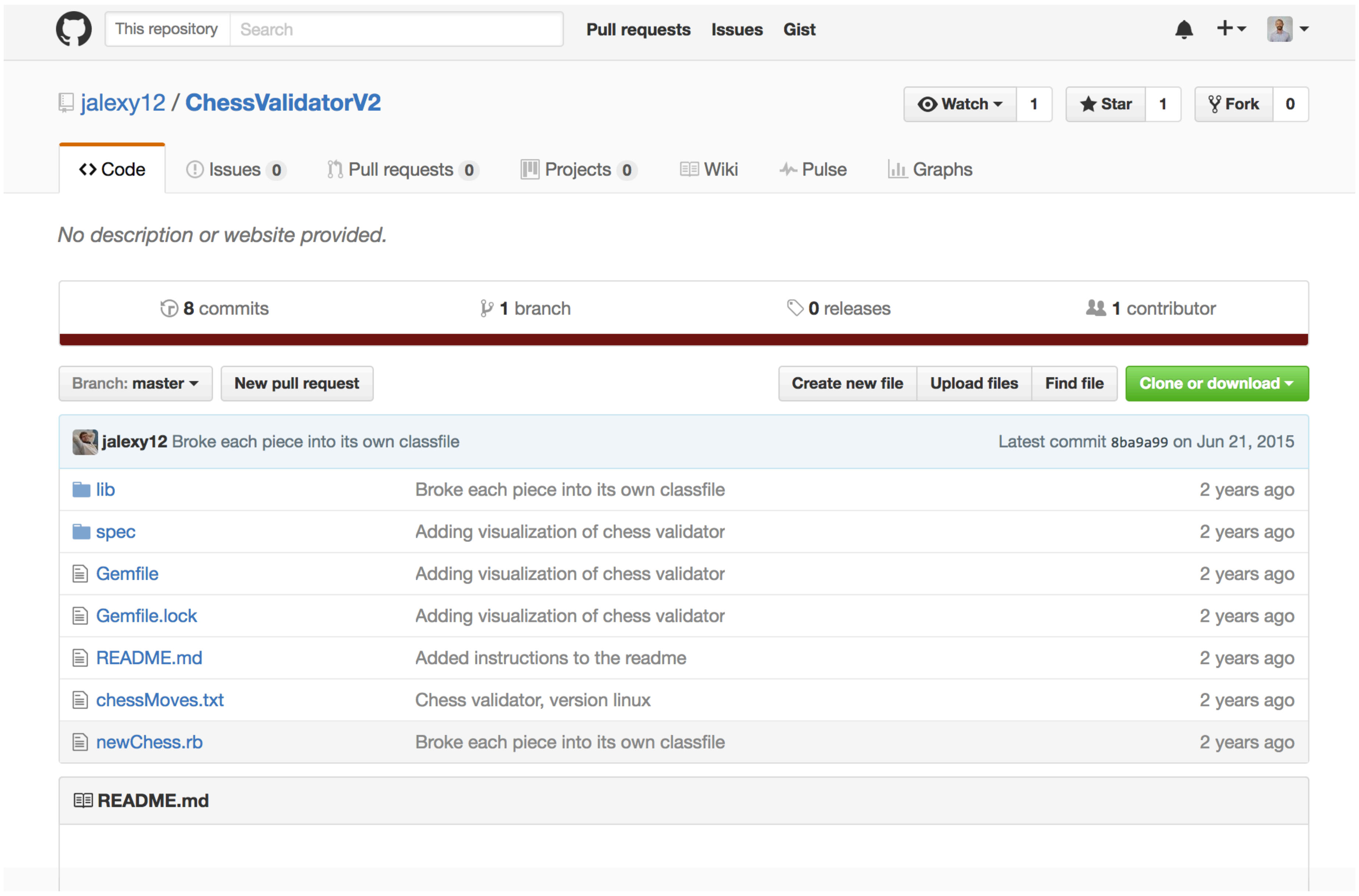
Once you have your to-do list how you want it, save the file and close it, deleting the ==== as well as the <<<< HEAD... before.

Once you've done that, add, then commit your changes, and you're all done. Now, if you run `git merge <branch-name>` it will say `Already up-to-date.`

One thing to keep in mind is that `<branch-name>` still has its own version of this file. But from now on, if you merge `<branch-name>` into `master`, it already knows how to handle this particular conflict, so it won't ask you again.

Forking Repositories

Before we go around making pull requests, we'll first need to fork a repository. Forking a repository simply creates an exact copy of an entire GitHub repository on your GitHub account.



This repository | Search | Pull requests | Issues | Gist

jalexy12 / ChessValidatorV2 — Watch 1 | Star 1 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Pulse | Graphs

No description or website provided.

8 commits | 1 branch | 0 releases | 1 contributor

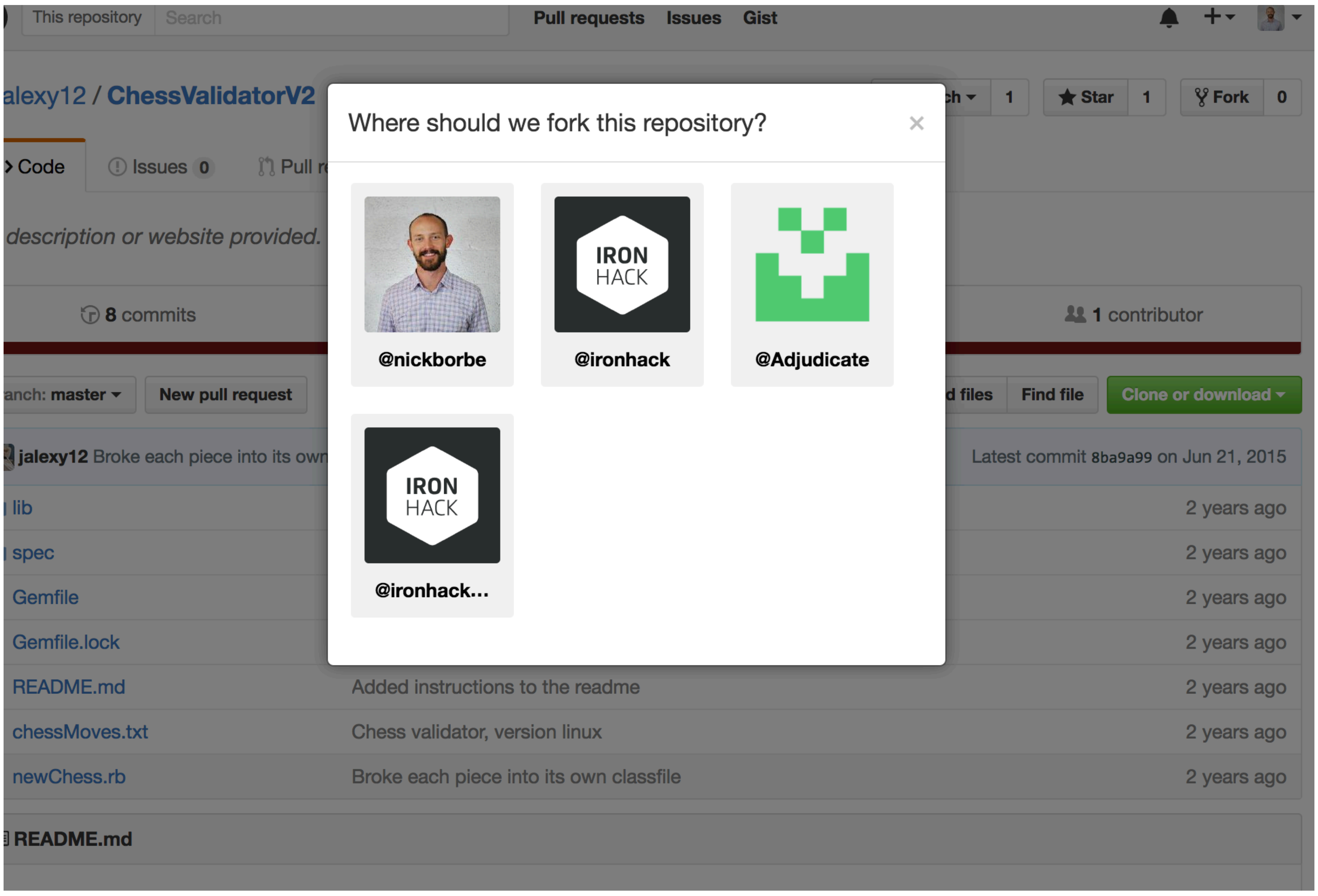
Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

jalexy12 Broke each piece into its own classfile — Latest commit 8ba9a99 on Jun 21, 2015

lib	Broke each piece into its own classfile	2 years ago
spec	Adding visualization of chess validator	2 years ago
Gemfile	Adding visualization of chess validator	2 years ago
Gemfile.lock	Adding visualization of chess validator	2 years ago
README.md	Added instructions to the readme	2 years ago
chessMoves.txt	Chess validator, version linux	2 years ago
newChess.rb	Broke each piece into its own classfile	2 years ago

README.md

To fork a repo, simply visit the repo's page on GitHub, and click the fork button in the top right corner. Once you click the button, you should get something like this asking you where to fork the repo.

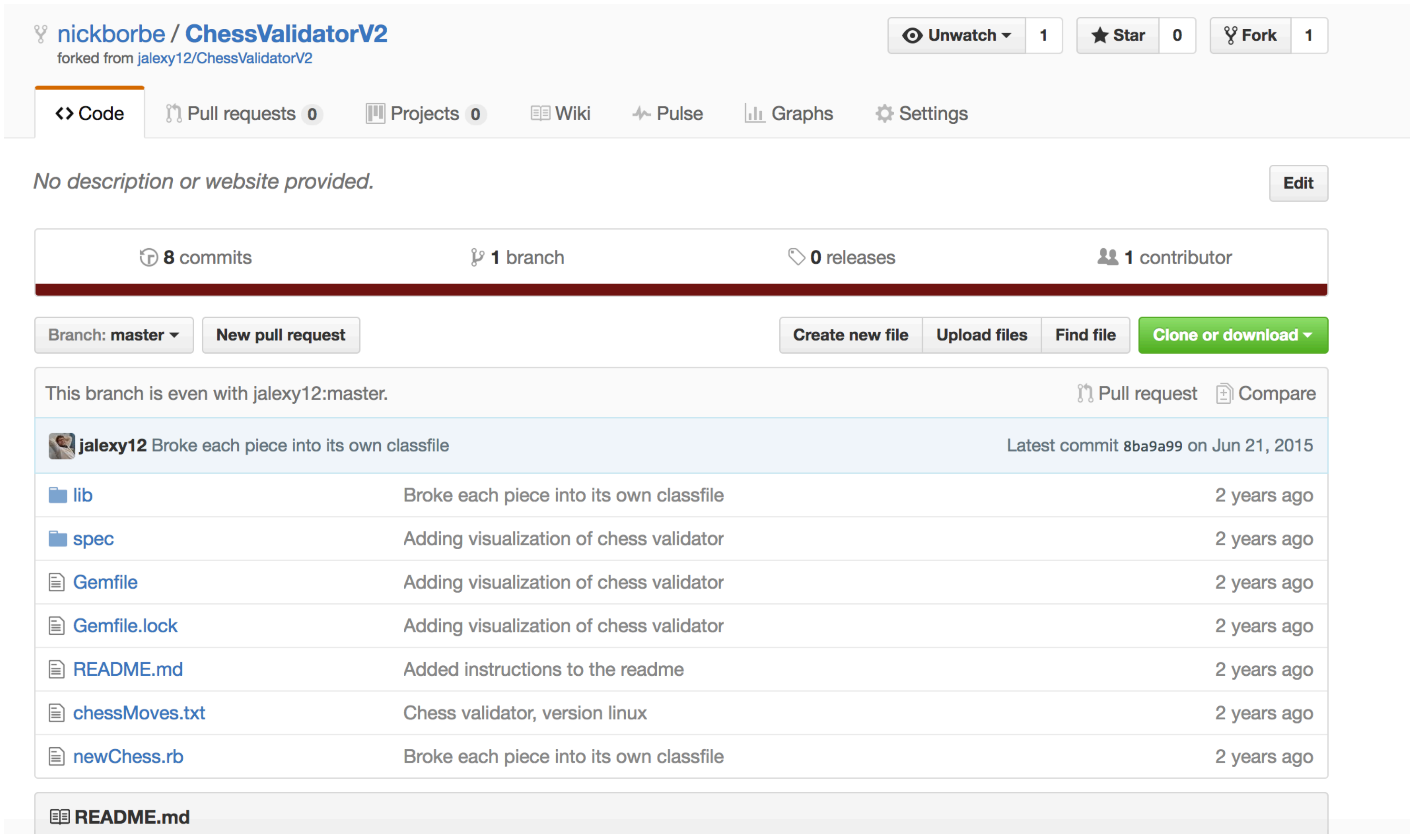


Where should we fork this repository?

@nickborbe | @ironhack | @Adjudicate | @ironhack...

If you're new to GitHub, you might not get this prompt. In any case, I will click my own account's icon, and then GitHub will take me to my new repo.

This screen looks pretty similar to the one where we forked the repository originally, however, now we should be on our own account.


nickborbe / ChessValidatorV2
forked from jalexy12/ChessValidatorV2

Unwatch 1 | Star 0 | Fork 1

Code | Pull requests 0 | Projects 0 | Wiki | Pulse | Graphs | Settings

No description or website provided.

Edit

8 commits | 1 branch | 0 releases | 1 contributor

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

This branch is even with jalexy12:master. Pull request Compare

jalexy12 Broke each piece into its own classfile — Latest commit 8ba9a99 on Jun 21, 2015

lib	Broke each piece into its own classfile	2 years ago
spec	Adding visualization of chess validator	2 years ago
Gemfile	Adding visualization of chess validator	2 years ago
Gemfile.lock	Adding visualization of chess validator	2 years ago
README.md	Added instructions to the readme	2 years ago
chessMoves.txt	Chess validator, version linux	2 years ago
newChess.rb	Broke each piece into its own classfile	2 years ago

README.md

If you want to work on the code from this repo on your computer (which is probably the entire reason you're doing this), you'll need to click the green button on the right side that says `Clone or download`. When you click on it, a link should appear with the text **Copy this link**.

Once you've copied the link, go into your terminal, navigate into where you want to put the folder that will house this project and type `git clone <remote git link>`, pasting the link we just copied.

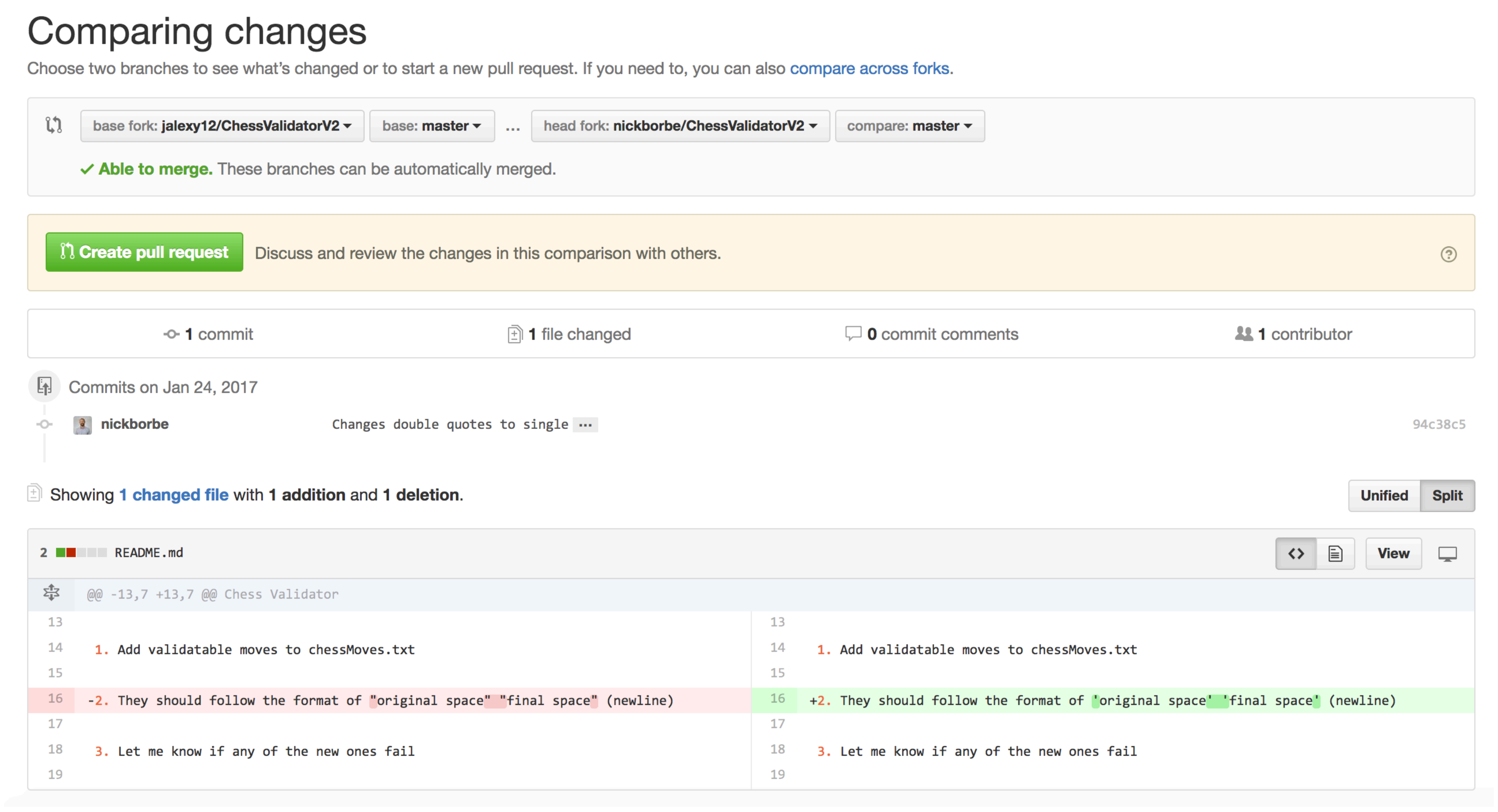
This will create a folder on your computer with a git repository already set up, hooked up with the new repository you just forked onto your GitHub profile. For the purposes of this course, when you are working on a forked repository on your own computer, simply commit changes to the `master` branch.

Pull Requests

Once you've made all the changes you're going to make, and you want to submit your work back to the GitHub repo you originally forked from, you'll need to submit a pull request.

It's called a pull request because, if you could, you would like to push your changes. However, since the original repo is not yours, and you don't have permission to push to it, you submit a request, asking the original owner to pull in your changes.

To submit a pull request, click the button that says `new pull request`, located on the same page as the `Clone or download` button, but on the left side. Once you click this button, you will be taken to a screen that looks like this:



Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

base fork: jalexy12/ChessValidatorV2 ▾ | base: master ▾ | ... | head fork: nickborbe/ChessValidatorV2 ▾ | compare: master ▾

✓ Able to merge. These branches can be automatically merged.

Create pull request — Discuss and review the changes in this comparison with others.

1 commit | 1 file changed | 0 commit comments | 1 contributor

Commits on Jan 24, 2017

nickborbe — Changes double quotes to single … — 94c38c5

Showing 1 changed file with 1 addition and 1 deletion. Unified | Split

2 README.md

```
@@ -13,7 +13,7 @@ Chess Validator

 1. Add validatable moves to chessMoves.txt

-2. They should follow the format of "original space" "final space" (newline)
+2. They should follow the format of 'original space' 'final space' (newline)

 3. Let me know if any of the new ones fail

```

This page shows you the changes you've made and how they compare to the original. The original version is in red, while your changes are in green. If you are satisfied with the changes, click the green button that says `Create pull request`. Once you've done that, it's up to the owner of the original repository to accept your pull request.

Summary

In this learning unit, we have learned how branches are organized in big developer teams. We have seen how we can create a new git repository in our local machine, how we can add and commit the changes we do in this repository and which is the relation between git and github.

We have also seen how to create branches, and how we can merge between them. When merging, sometimes we can find merging conflicts, and we have explained how to solve these conflicts. To finish up this lesson, we have learned how to fork a repository and how to create a pull-request over it.

Extra Resources

- Git flow

Any doubt? Ask our AI Chatbot!

Lesson Completed

Mark as not completed

How would you rate the content of this unit?

PREVIOUS
← Git and GitHub - Fork and Clone

NEXT
Environments →