Version Control Systems

Thursday, February 28, 2019 9:52 PM

Why Version Control?

- Collaboration
- Storing Versions
- Restoring Previous Versions
- Understanding What Happened

Version control systems are tools that help manage changes to files over time. In this course, version control for source code management will be examined. Generally speaking, these tools exist everywhere. Even Sharepoint, Dropbox, and Google Drive offer versioning which allows you to roll back as needed to prior versions of a file.

Most importantly, version control systems enable efficient collaboration for those persons contributing to a project. Imagine you and your team are working on a development project. Without version control systems, you're probably working with shared folders containing the whole project and possibly replicating it few times in another location as a backup. In a situation like this, multiple people of your team may work on the same file at the same time, potentially causing many issues.

These problems are mitigated by using a VCS. Essentially, each member keeps a full replica of the project on their local computer resulting in an inherently distributed version control system. When using version control systems, it also becomes much easier to see who is making changes, who has approved them, and how a given file looked on a specific date.

- Concurrent Versions System

1. Originally was a "first come, first serve system"

- Apache Subversion

1. Atomic operations
2. Slow comparative speed

- Git

1. Originally built to support Linux Kernel developing

- Mercurial

1. Originally made to compete with Git

The first Version Control system was developed around the 1970s. Since then, many VCSs have been developed, and today there are many available possibilities for those organizations who want to use Version Control.

- The Concurrent Versions System was first designed in the 1980s. Originally, it handled conflicting situations. For example, when two engineers who worked on the same file, CVS allowed only the latest version of the code to be worked on and updated. As such, it was a first come, first serve system.
- Apache Subversion (SVN) was created as an alternative to CVS. SVN uses atomic operations, meaning that either all changes that are made to the source are applied or none are applied. No partial changes are allowed, avoiding many potential issues. A drawback of SVN is its slower speed.

- Git was originally built to support the Linux Kernel developing. Git will be explored in this lesson.
- Mercurial was originally made to compete with Git for Linux kernel development. Unlike most Version Control Systems, it’s written in Python and not in C.

Overview of Git

Saturday, March 2, 2019 11:20 PM

- Created by Linus Torvalds in 2005
- First version built in about 10 days
- Version Control
- Code collaboration
- Minimize mistakes

Git was created by Linus Torvalds in 2005. Linus is also the creator of the Linux Operating System, so the history of Git is closely connected to Linux as well. In fact, when Linux was first being developed, Linus and his team struggled with managing large codebases maintained by many engineers.

Then, the Linux kernel was unusual when compared to most commercial software projects. There was many committers and a high variance of contributor involvement. Also, the knowledge of the existing codebase was very limited.

At first, they used a system that is called BitKeeper which enabled a local copy of projects along with a central server where the final code resided and where distributed changes were pushed. BitKeeper had some technical issues, and worse, it was not open source. Therefore, there were people who did not want to use it.

These few reasons are what drove Linus to build his own Version Control System: git. The first version of the system was built in a very short time, likely about 10 days. The actual amount of that early code is fairly small because they focused on getting the basic ideas right. The goal was to design a robust system to solve issues with existing tools and to help improve developer life.

Git is a distributed version control software that keeps track of every modification to the code. If a mistake is made, developers can look back and compare earlier versions of code to help fix the mistake minimizing disruption to all team members. Version control protects source code from any kind of errors that may have serious consequences if not handled properly.

It is distributed, so every team member has their own copy of the project files. This file is a complete copy of the full project, not just the files being worked on. These changes may be incompatible with those changes made by another engineer, and Git helps track every individual change by each contributor to prevent work from conflicting.

Moreover, you have full control of your local repository, so you may decide to commit and push only some parts of your work and leaving some others on your local system. This practice may be the case of a file containing confidential data, credentials and so on.

When should you use it?

Have you ever:

- Maintain multiple versions of a product?
- See the difference between two (or more) versions of your code?
- Prove that a particular change broke or fixed a piece of code?

- Review the history of some code?
- Submit a change to someone else's code?
- Share your code, or let other people work on your code?
- See how much work is being done, and where, when and by whom?
- Experiment with a new feature without interfering with working code?

So when should git be used? In reality, git should be used for any engineering-related project – it does not have to specifically be for code management. Regardless of the file types being used, git proves to be valuable for team-based projects in which they require error-tracking, code backup and recovery, code history, change logging, and an easier way to experiment with code.

> *In these cases, and no doubt others, a version control system should make your life easier.* *

*http://stackoverflow.com/a/1408464

Does it matter for Networking?

- Configuration Files
- Scripts
- Variable Files
- Playbooks
- Manifests

Version Control and Git can be useful not only for developers, but for networking teams as well.

For example, it's useful to have a history of all the configurations that are active on network devices. Using a git-based system allows you to see the configurations, how and when they changed, and of course, who made the change. Also, as more automation tooling is used, for example, Ansible, you can use git to version control playbooks, variables, and configuration files. As you collect information from switches such as operational data, you can store that data in text files. You can easily track and see the changes in operational data such as changing neighbor adjacencies, routes, and the like.

Git Architecture

- Working Directory
- Staging Area
- Local Repository
- Remote Repository

Git Architecture is composed of several different components: **Remote Repository**, **Local Repository**, **Staging Area, and Working Directory**.

- **Remote Repository:** A remote repository is where the files of the project reside, and it is also where all other local copies are pulled from. It can be stored on an internal private server or hosted on a public repository such as GitHub or BitBucket.
- **Local Repository:** A local repository is where snapshots, or commits, are stored on each individual person's local machine.
- **Staging Area:** The Staging Area is where all the changes you actually want to perform are placed. You, as a project member, decide which files Git should track. For example, you can decide to add and commit files to fully become part of your local repository by moving them to

the staging area first, without including all files in your project.

- **Working Directory:** A Working Directory is a directory that is controlled by git. Git will track differences between your working directory and local repository, and between your local repository and the remote repository.

Git Commands

Saturday, March 2, 2019 11:21 PM

One of the most important elements of Git is a **repository**. A repository is a directory that is initialized with git.

A repository can contain anything such as code, images, and any other types of files. This understanding is the foundation of getting started with git locally on your machine.

When you create a new project, `git init` must be used locally on your machine in order to initialize a project to work with git. You must initialize a working directory with the `git init` command for git to start tracking files.

Note
When executing the `git init` command, it also creates a subdirectory called **.git** that contains all of the local snapshots (commits) as well as other meta data about the project.

Git Command	Description
`git init`	Initialize a directory for a git project
`git config <params>`	Configure git params such as username/email
`git status`	Check status of your project
`git add <file/dir>`	Start tracking files and add them to the staging area
`git commit –m <message>`	Create a local snapshot

Once you initialize a directory using the `git init` command, it's recommended to configure git with a username and email address. This information is used in the git history and during commits making it possible to see who's made changes. Configurations for git can be per project or for all projects on the system. For example: global. Note that it is also possible to configure the same settings using the **.gitconfig** file instead of using terminal commands.

To see a list of all configurable options for a **git config**, you can use the Linux command `«man git-config»`.

The git status command should be used often. It allows you to see the status of your working directory and local repository. It shows you the files that need to be staged, the files that are staged, which branch you are on, and if a commit is required. Also, it will show the files that are not being tracked by git.

The figure shows two outputs of using the git status command. The first is from a directory that was just initialized with git. For example, there have been no commits. The second shows the output after a commit.

If a new file is added to the directory after the initial commit, you would see an output like the following depicting that there is now a file not yet tracked.

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

Once the project is created and initialized, you will use the git add command as you update the project. The git add command performs two primary functions: (1) starts to track files and (2) adds files to the staging area.

If you want to add two new switch configuration files to a project, you would add them as follows.

```
$ git add s1.cfg
$ git add s2.cfg
```

Note
You can use wildcards and could have done git add s*.cfg. There are other flags to use within the **git add** command to add all files in a given a directory, but our focus is on the foundational topics, thus they are out of the scope of this course.
After the new configuration files have been added, if you re-issue the **git status** command, you would see the following:

```
$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
new file:   s1.cfg
            new file:   s2.cfg
```

In the configuration snippet above, you can see that the files have been added and are ready to be committed.

Once the files have been added to the staging area, they are ready to be committed. Use the **git commit** command to commit the staged changes. What the **git commit** command is really doing is creating a point-in-time local snapshot of the project. All incremental changes are stored in the **.git** directory that was automatically created when the **git init** command was executed.

When you use the **git commit** command, you are required to include a commit message. This practice is shown in the graphic using the **–m** flag. When you commit your changes, it creates a commit object representing the complete state of the project, including all files in the project.

Git Command	Description
git remote <params>	Add (view) remote repositories
git push <params>	Push local snapshot (commit) to remote repository
git pull <params>	Patch and merge changes from remote repository (combines **git fetch** and **git merge**)
git clone <params>	Copy another (remote) project to your local machine

At this point, after the git commit command is used, you have a local snapshot. You can see that there is still value in versioning even if you are not working on a team-based project. That said, it is most common to create a local snapshot, and then push it upstream to a git server. When you create your own git project as has been shown, you need to use the git remote command to add one or more remotes. These remotes are where the git servers reside.

In the example, you will use a different directory on the server, but in reality, this practice will be a private server or a cloud service such as GitHub. This example uses the command:

```
$ git remote add local ~/local_remote/:
```

The above command contains several important elements:

- **git remote add** is the command to add a remote.
- **local** is an alias of the remote URL – you define what you want to call this alias. **«origin»** is also quite common, but you can use more relevant names especially if you have multiple remotes for a given project.
- **~/local_remote/** is the directory that will be used as the remote. For example: This directory is **/home/cisco/local_remote**.

Note
In order to use **~/local_remote** as a git repository, the command **$ git init –bare** was used to initialize it.
Once a remote has been added, you can now push your local snapshot to a remote git repository.

After one or more remotes have been configured, you are ready to push your changes. You can push your changes using the **git push** command. The command that is used in figure is as follows:

$ git push local master
Note that **local** refers to the alias that was defined when then remote was created and that **master** refers to the branch. By default, there is only a master branch in a repository, but note that you can push specific branches if you are working on a project that does have multiple branches.

The examples that have been used thus far were started by creating a new project. Remember that the first command used was the **git init** command.

It's also possible that you don't want to create a new project, but rather copy another pre-existing project. For example, you want to test an open source project or just some code you have found on GitHub. You can run this test by using the command:

git clone https://github.com/<userid>/<project>
Since this example uses a local directory as a remote, you can navigate to a new directory and and clone the same directory.

cisco@cisco:~/newdir
$ ls cisco@cisco:~/newdir$
Note
*Note that there are no files in the **newdir** directory.
cisco@cisco:~/newdir$ git clone ~/local_remote/
Cloning into 'local_remote'...
done.
cisco@cisco:~/newdir$ ls local_remote/
s1.cfg s2.cfg
The project has been cloned into a new directory. By using the **ls** command, it can be seen that you now have both files that were originally pushed from another directory into the remote called **local**. Since the project is a clone, there is no need to use the **git init** command as the project was already initialized for git.

Note
You can add an optional argument to the clone command if you want to store the cloned project in another directory that is not equal to the name of it's directory. For example, **git clone ~/local_remote/ mylocal** would clone the project into **mylocal/** instead of **local_remote/**. This can be helpful for testing, otherwise, you would have duplicate directory names.
If you're collaborating with other engineers on a project, for example, in a follow-the-sun model, you would commit and push your changes before you leave for the day. When your counterpart comes in for the day as you're leaving, they need to get the updates you pushed to the repository. The easiest way to do that is using the **git pull** command. This command will automatically pull the updates down from the remote.

The **git** command is actually performing two operations. It's fetching all updates from the project and merging them with your repository. You can also perform these two operations manually using **git fetch** and **git merge** commands.

Git Workflow

Saturday, March 2, 2019 11:21 PM

```
$ git init
$ git config --global user.name "John Smith"
$ git config --global user.email john_smith@cisco.com
$ git add switch.cfg
$ git commit –m "initial commit"
```

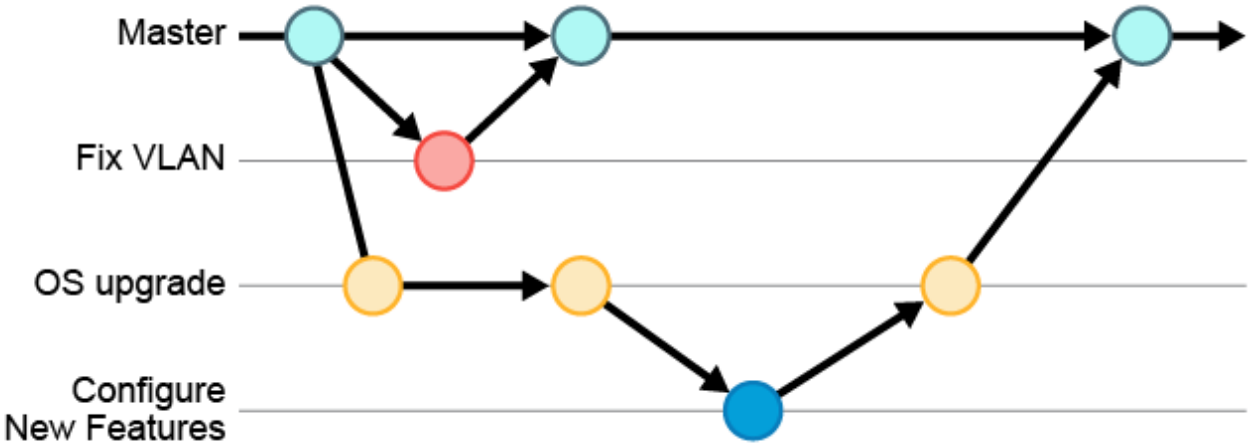
Two examples of remotes:

```
*$ git remote add local ~/local_remote/
$ git remote add origin https://github.com/<user>/<repo>
$ git remote –v   # view configured remotes
$ git push local master
```

*Assumes ~/local_remote has been initialized as a bare repository

After reviewing each of the commands already, you can see a sample workflow that starts with initializing a directory for git, adding a file, committing it to a local snapshot, and then finally pushing it to a remote a git server.

Split the project into more manageable pieces – diverge from the mainline to add specific features, fix bugs, etc.



In a larger team-based project, you will want to leverage branches. For example, think about the development of Cisco IOS. As new features are added and bugs are fixed, the relevant code that is required for each could go into their own branch. Once the branching is considered complete, it can be merged into the mainline, or master branch.

There are three main commands to be aware of when working with branches including **git branch**, **git checkout**, and **git merge**. These commands are reviewed in upcoming graphics.

Git Branches

Saturday, March 2, 2019 11:23 PM

Git Command	Description
git branch	List, create, or delete branches
git checkout	Move between branches
git merge	Merge one branch into another

The **git checkout** command lets you navigate among branches. Since by default you have only a master branch, you can also use **git checkout** to automatically switch to that branch when you use the **–b** flag. In a given project, you can check to see which branch you are on by simply using the **git branch** command as follows:

```
cisco@cisco:~$ git branch
* master
cisco@cisco:~$
```

If you need to fix a bug which could be a config file or code, you can create a specific branch. You can create this new branch and automatically navigate to it as follows:

```
cisco@cisco:~$ git checkout -b fix_aaa_bug
Switched to a new branch 'fix_aaa_bug'
cisco@cisco:~$
cisco@cisco:~$ git branch
* fix_aaa_bug
  master
cisco@cisco:~$
```

At its starting point, the branch has all files from the master branch. Once you update the files with the fix, commit them and push them back using a command such as **git push local fix_aaa_bug**.

If you don't need to navigate directly to the newly created branch, you can simply create a branch with the **git branch**command too. For example, **git branch fix_snmp** would create the branch:

```
cisco@cisco:~$ git branch fix_snmp
cisco@cisco:~$ git branch
  fix_aaa_bug
  fix_snmp
* master
```

In the example, a new branch was made that is called **fix_aaa_bug** and added a new file that is called **aaa.cfg** which serves as the fix. Once you **fix** the project in the branch and it is fully tested. You can then navigate back to the master branch. The following shows the full workflow for merging the fix back into the master branch.

1. Fix applied to fix_aaa_branch
2. Navigate back to the master branch

   ```
   cisco@cisco:~$  git checkout master
   Switched to branch 'master'
   cisco@cisco:~/testgit$ ls
   newdir  s1.cfg  s2.cfg
   cisco@cisco:~$
   ```
3. Merge the branch back into the master branch:

   ```
   cisco@cisco:~$ ls
   aaa.cfg  newdir  s1.cfg  s2.cfg
   ```
4. Confirm that you see the updated changes (here it is **aaa.cfg**):

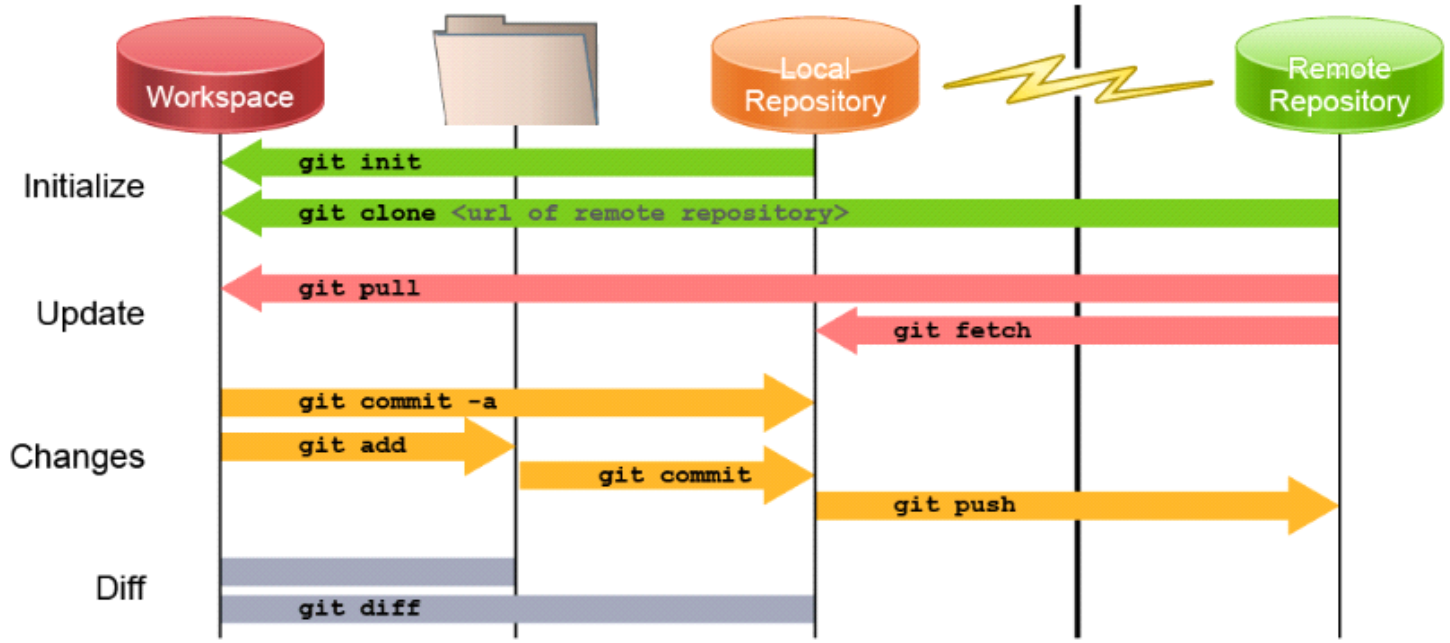
   git branch workflow

```
cisco@cisco:~$ git branch
* master
cisco@cisco:~$ git checkout -b fix_aaa_bug
Switched to a new branch 'fix_aaa_bug'
cisco@cisco:~$
cisco@cisco:~$ git branch
* fix_aaa_bug
  master
cisco@cisco:~$
cisco@cisco:~$ git checkout master
cisco@cisco:~$ git merge fix_aaa_bug
```

This figure shows using the commands that are previously reviewed, but all in a single workflow. Performing merges through a web interface (client) such as GitHub is much more common than doing this procedure via the command line as shown.

Using Git

Saturday, March 2, 2019 11:23 PM



This figure summarizes the commands that have been reviewed thus far including: **git init**, **git clone**, **git pull**, **git commit**, **git fetch**, **git add**, **git push**, and introduces a new command **git diff**.

You can use the **git diff** command to view the differences of files in your working directory that is compared to the same file in your local repository (last snapshot commit). The most important point to understand as you get started with **git** is that you have a local and remote repository. Remember that when you stage and track files with **git add**, they are being added to the staging area, and then when you commit them, they are added to your local repository. It's not until you perform a **git push** so that your local snapshot (repo) is pushed to the remote repository.

From <https://ondemandelearning.cisco.com/cisco-cte/npdesi10/sections/23/pages/7>

Collaborating with GitHub

Saturday, March 2, 2019 11:24 PM

What is GitHub?

- Distributed Version Control System based on Git that is a web-based hosting service
- Free Version for public files / code repositories
- Git + Code Review
- GitHub Enterprise

GitHub is a distributed version control system that is based on Git that is a web-based hosting service. It is the leading git platform for independent and open source projects. While GitHub offers unlimited free repositories as long as the repository is public, they offer subscriptions for those organizations who want private repositories.

In addition to being a git remote, GitHub also offers code review functionality. They support a feature such as a “Pull Request” which allows people to contribute to projects that they don’t maintain (and own). In this process, you have the ability to perform **code review**. Code review occurs when contributions to the repository are being made. There is a forum-like interface that allows the project owners and contributors to communicate during the review process (until the contribution is merged).

Anyone can create a free account, create public repositories, and start sharing projects. Thanks to this structure, it’s a great way to show your skills and interests and it can be considered a public resume where future employers can see exactly what types of contributions and projects you have worked on.

Public Git Cloud Platforms

- GitHub
- Stash
- BitBucket

While GitHub is the leading platform for remote git repositories, you should understand that others do exist. In addition to GitHub, one other service is BitBucket. BitBucket supports both git and mercurial version control systems. Atlassian, a popular cloud service that offers many developer friendly products, acquired BitBucket in 2010. In 2015, Atlassian renamed their Enterprise git platform from Stash to BitBucket Server, which competes with GitHub Enterprise.

- You get to use all the git commands already reviewed using a GitHub repository as a remote
- GitHub Pull Request

1. Offers the ability to propose changes to a repository that you do not maintain
2. Fork & Pull Method

Working with a git remote that is stored on GitHub is no different than working with local remotes (as covered already). You still use all the same commands such as **git clone**, **git commit**, **git status**, **git push**, and the like. The value of using GitHub is that it has a web interface that offers easy access to view all files in a repository, the state of the repository at a given point in time (before and after commits), but also offers the ability for people to contribute to projects that they do not own. One specific way that GitHub allows you to do this procedure, is through the **Fork & Pull** model.

Fork & Pull Model

- Copy an existing repository to your own GitHub account (Fork)
- You are the owner of this new repository
- Clone your new repository
- Make changes as necessary
- Push to your master
- Issue a Pull Request within GitHub UI

  The ability to **fork** a repository allows anyone to obtain a full replica of a project they don't own. Once you fork a repository, GitHub copies it for you, and you now have full access to clone/push changes to this repository. Once you push changes to your copy of the repository, GitHub recognizes that your copy is *ahead* of the upstream original repository. You can now issue a **Pull request** to the original repo with the goal to contribute your work back to the original project.

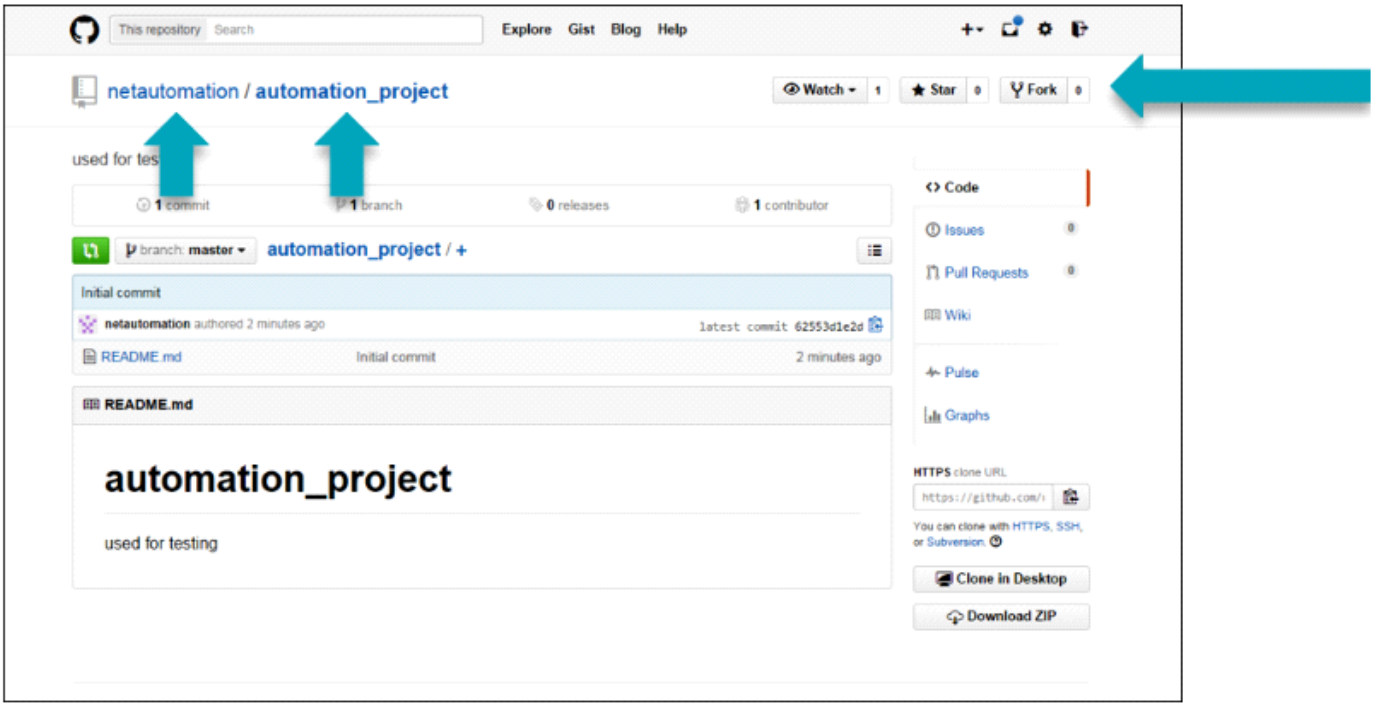
GitHub Pull Request: Fork and Pull

Saturday, March 2, 2019 11:24 PM

- User Point of View
- The person who wants to submit a change to another person's repository.

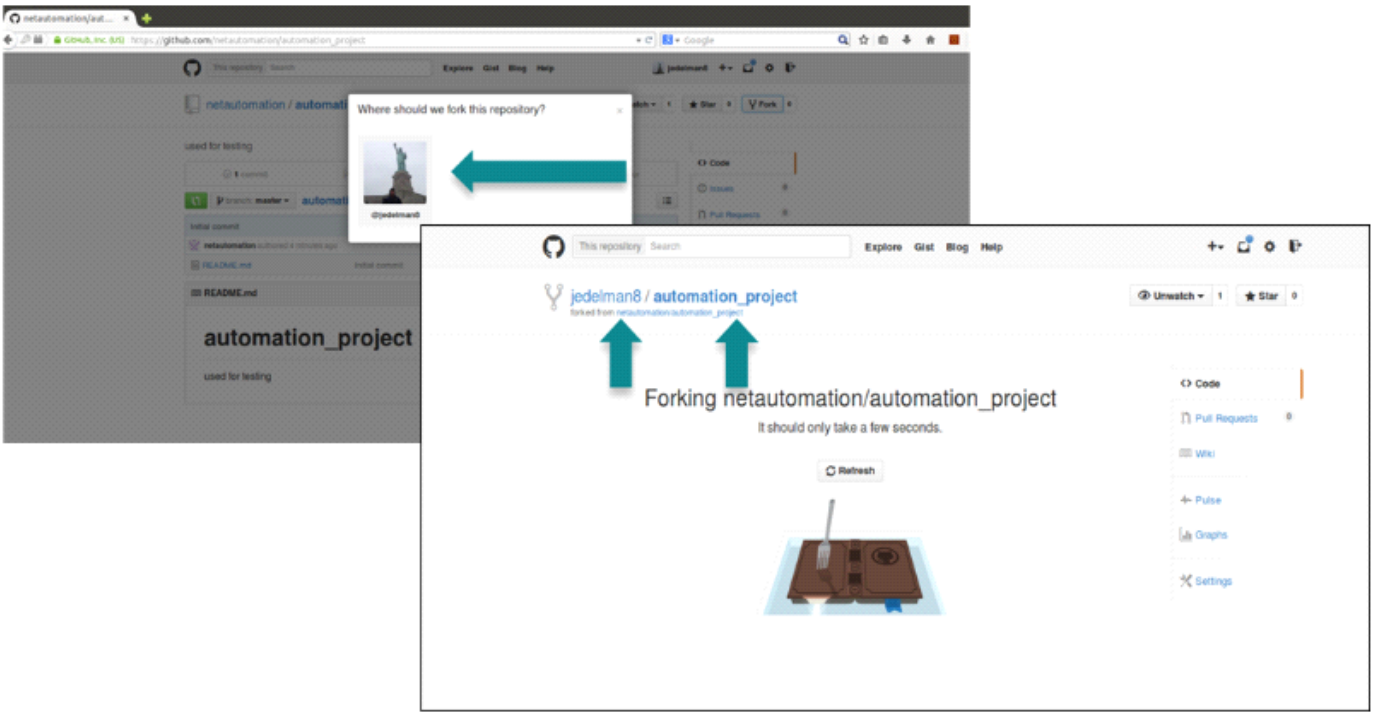
  The next few graphics walk through step by step the process of how to perform a pull request using the GitHub web interface.

  Step 1: Navigate to the Repository & Fork



First, click the Fork button to copy the **netautomation/automation_project**. Remember that **netautomation** is GitHub user that owns the project.

Step 2: Fork the Repository to your Account



Once you click the Fork button, you can choose where you want to fork it. If you're a member of any GitHub teams, you can choose to fork it to a Team account or your personal account. Once it's forked you can see that you now have a full copy of the project.

The original project was **netautomation/automation_project** and the new copy here is **jedelman8/automation_project**.

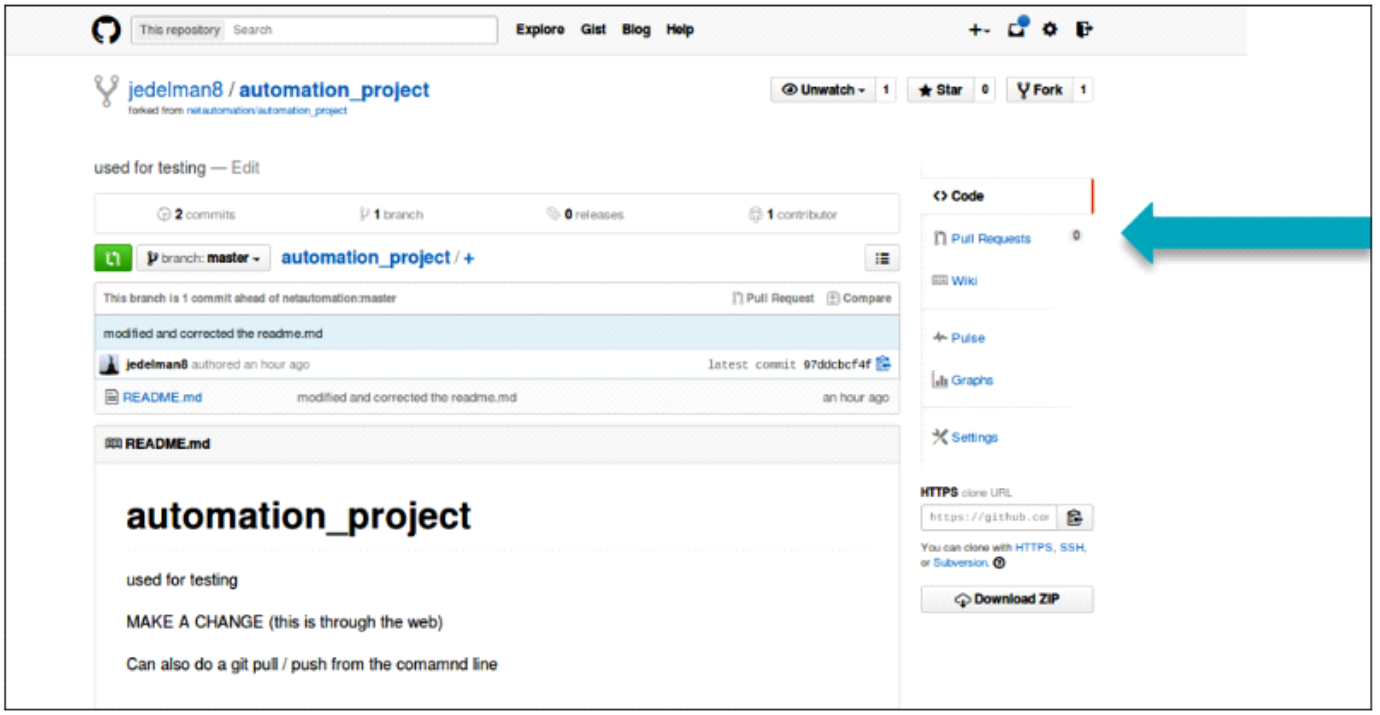
Step 3: Make a change to YOUR new repository

```
git clone https://github.com/<username>/automation_project
cd automation_project
touch router.cfg  (make a change)
git add router.cfg
git commit –m 'added new config file'
git push origin master
```

Since you own the project, you can now clone it just like you would a project that you created. For example, you can issue the following commands:

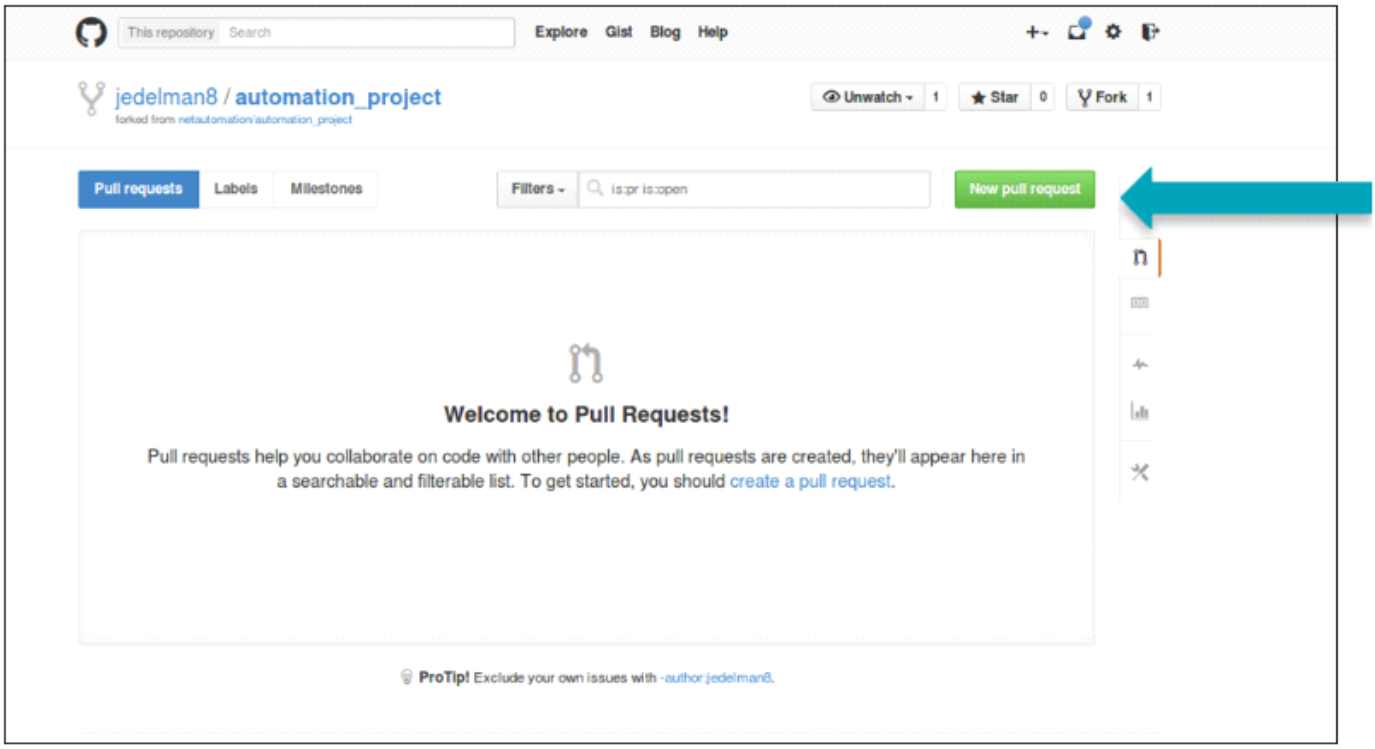
```
git clone https://github.com/jedelman8/automation_project
cd automation_project
touch router.cfg  (make a change)
git add router.cfg
git commit –m 'added new config file'
git push origin master
```

Step 4: Create the Pull Request



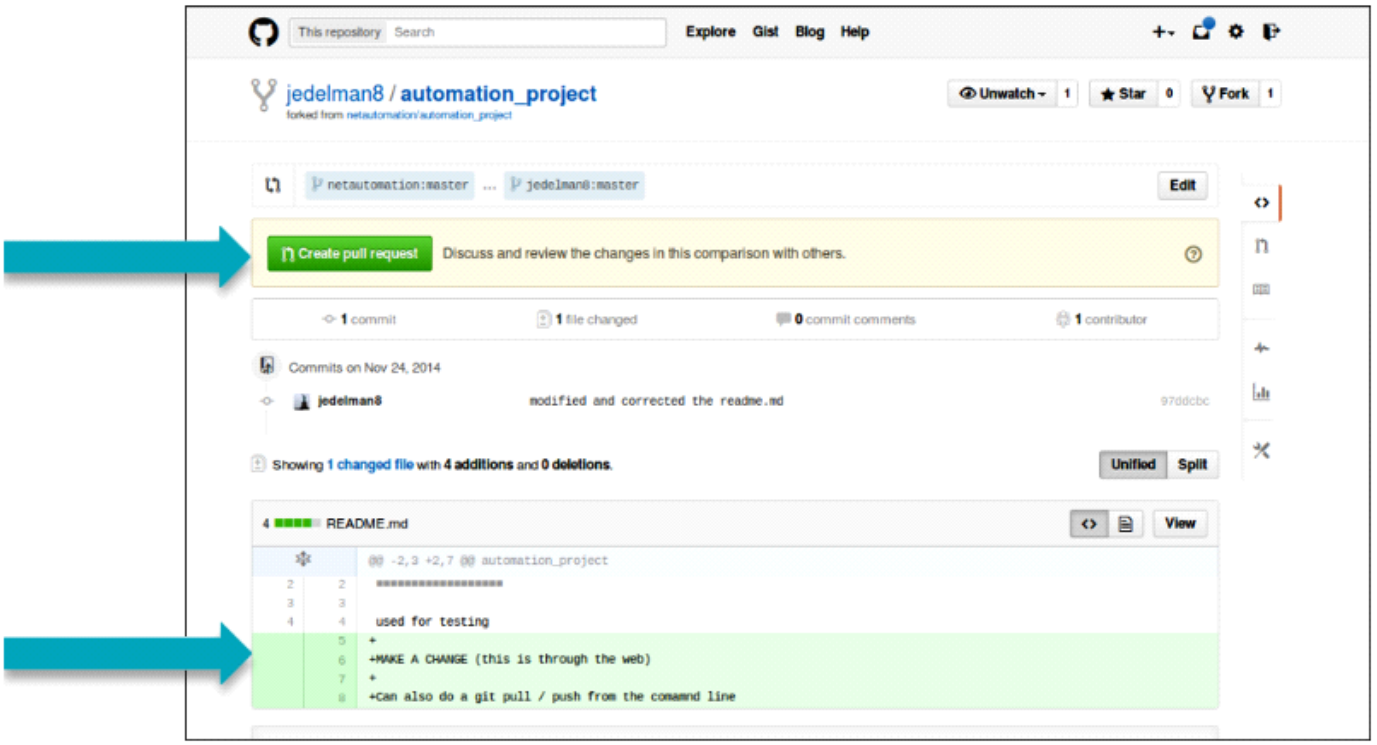
Once you push your changes to your repository, you can navigate back to the GitHub Web UI and create a Pull Request. First, you need to click the Pull Requests link in the GitHub UI.

Step 5: Click “New Pull Request”



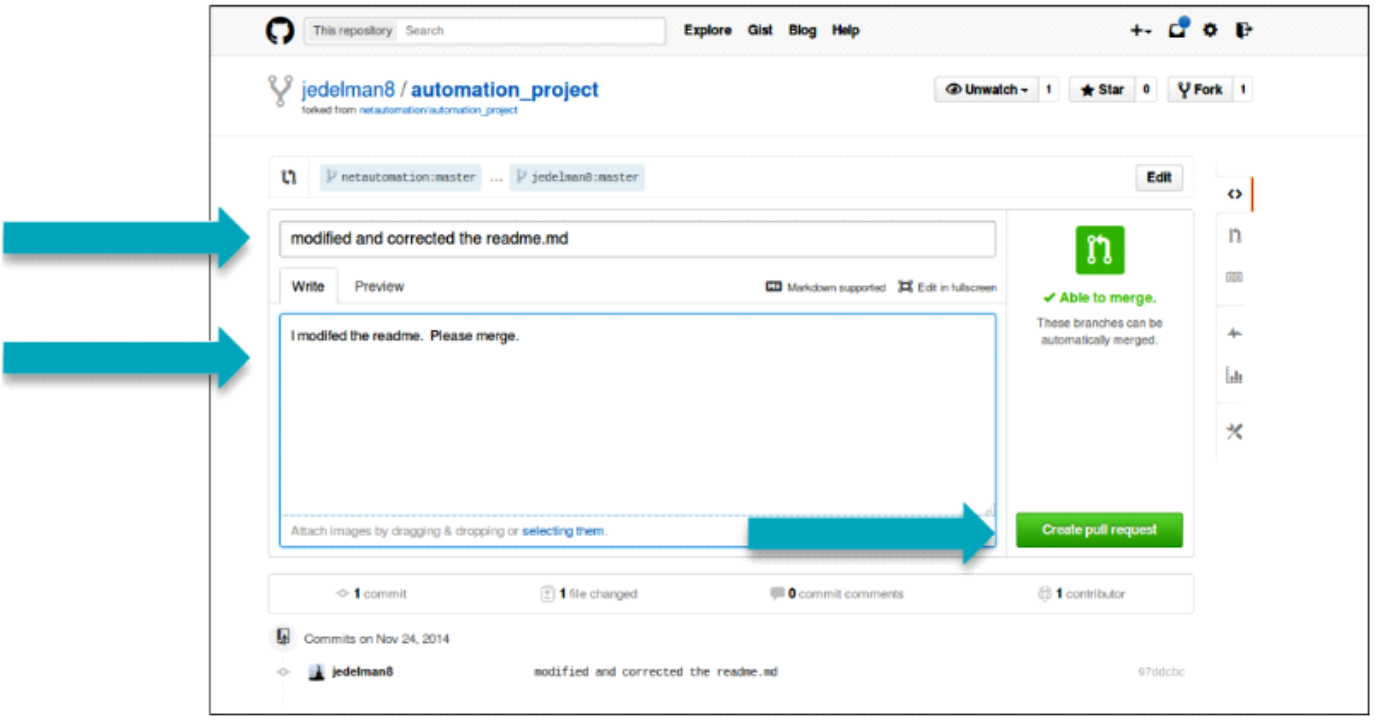
First, you need to create a New Pull Request.

Step 6: Review the Pull Request and Create



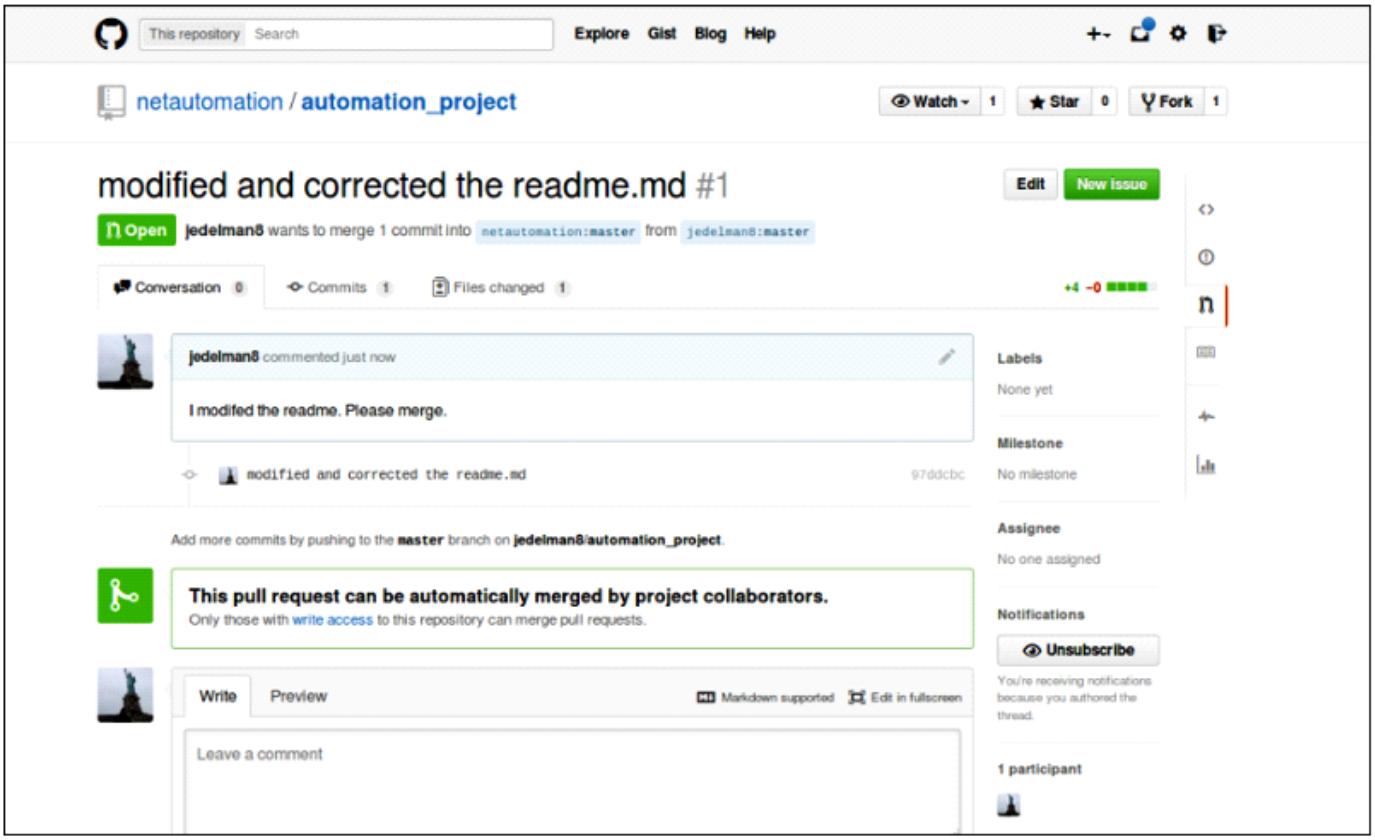
You can then view the files and updates that are getting included as part of the Pull Request. Once they are reviewed, you can click the Create Pull Request Button.

Step 7: Add Comments to the Pull Request



After clicking “Create Pull Request,” you provide a title and description for the changes you are requesting. Finally, click Create Pull Request.

Pull Request Succeeds



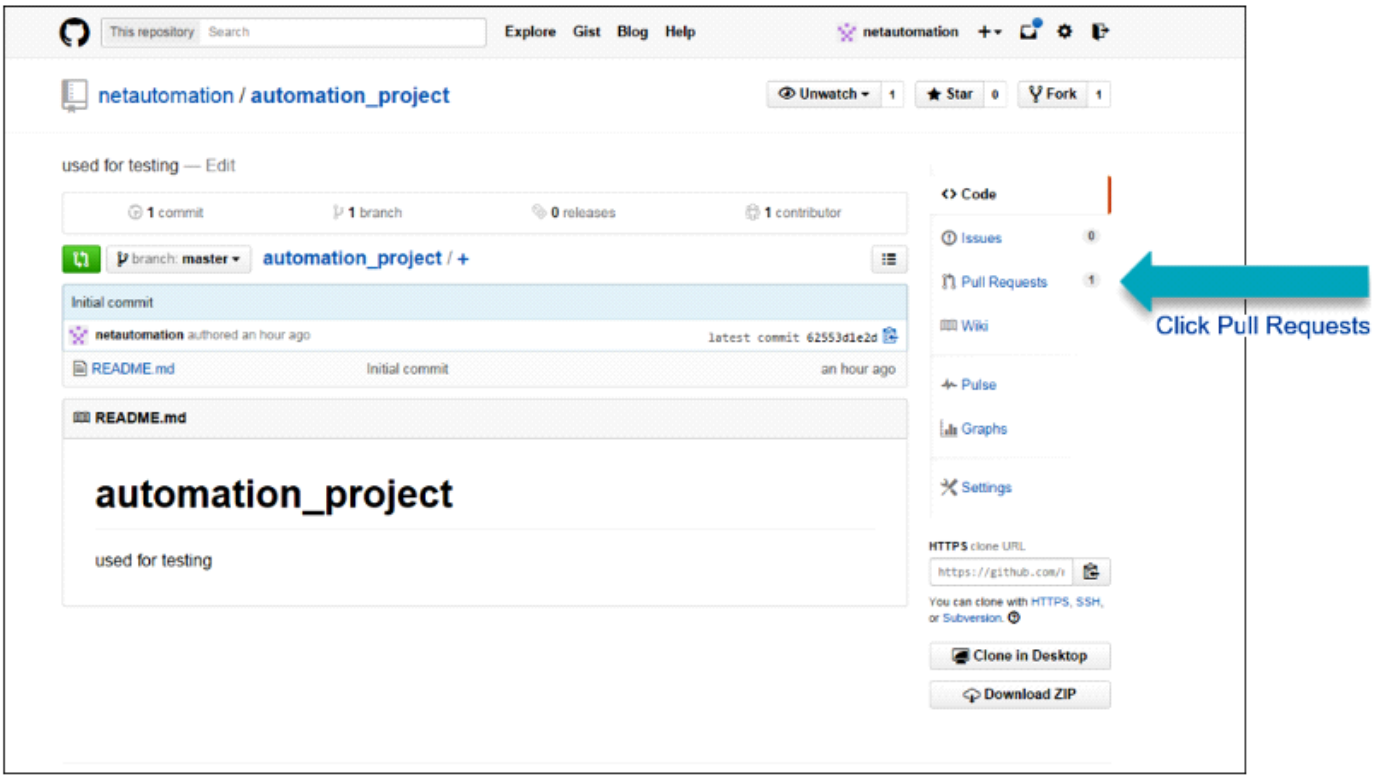
You will then see a screen that shows the Pull Request was created successfully.

Changing Views

Project Maintainer Point of View

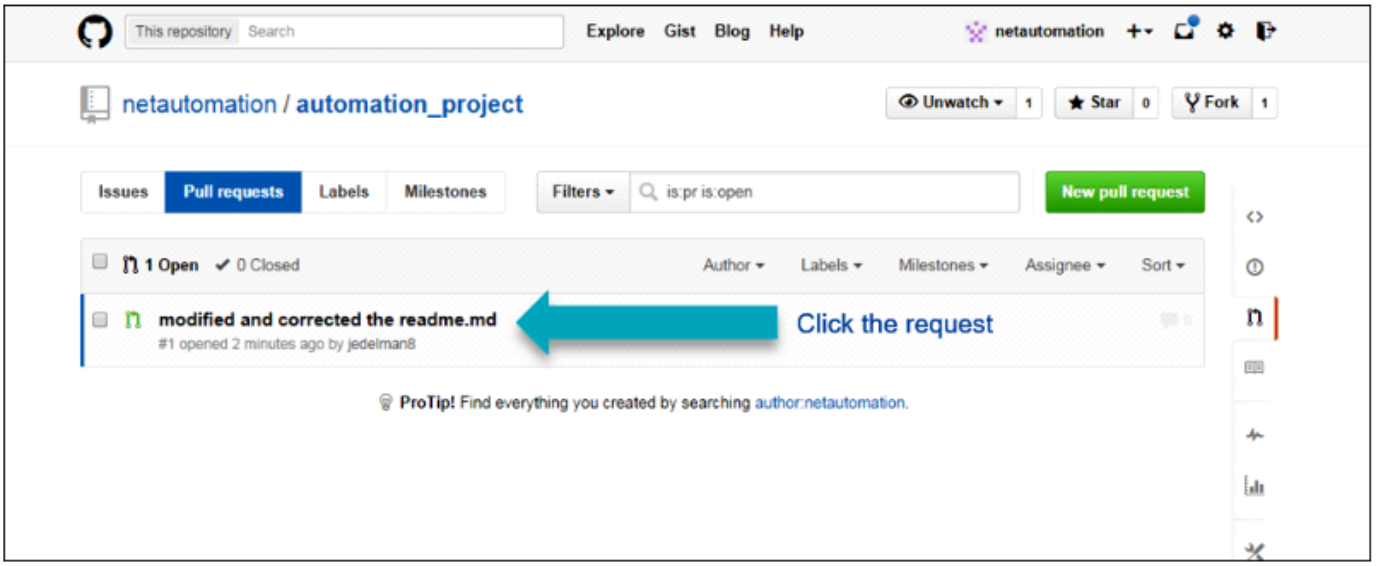
- The person who will review and merge the pull request.

  Step 1: Browse to Correct Repository



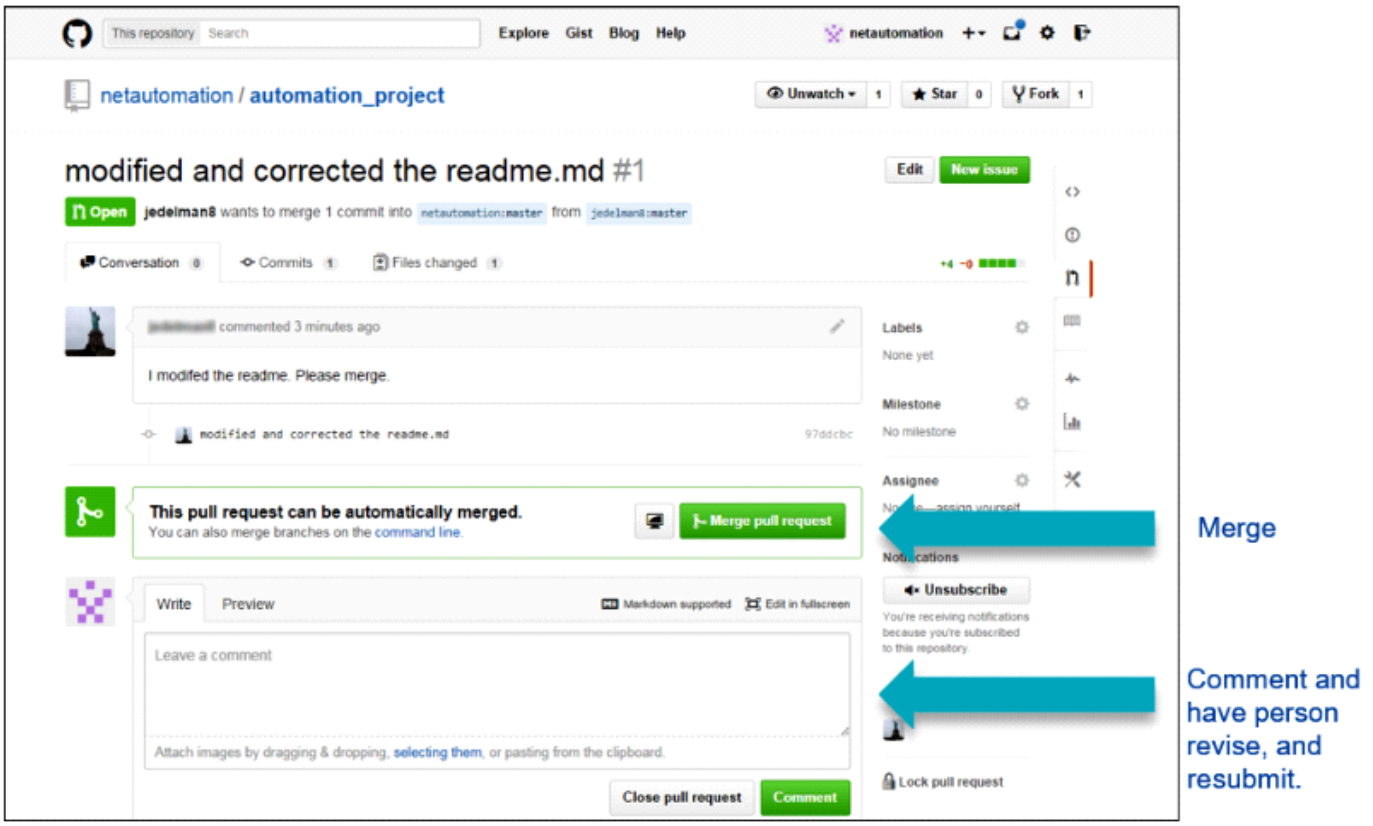
As a project owner, you simply Log in to GitHub, go to project, and you will see the Pull Requests link, which will show how many open Pull Requests there are for this project.

Step 2: Examine List of Pull Requests



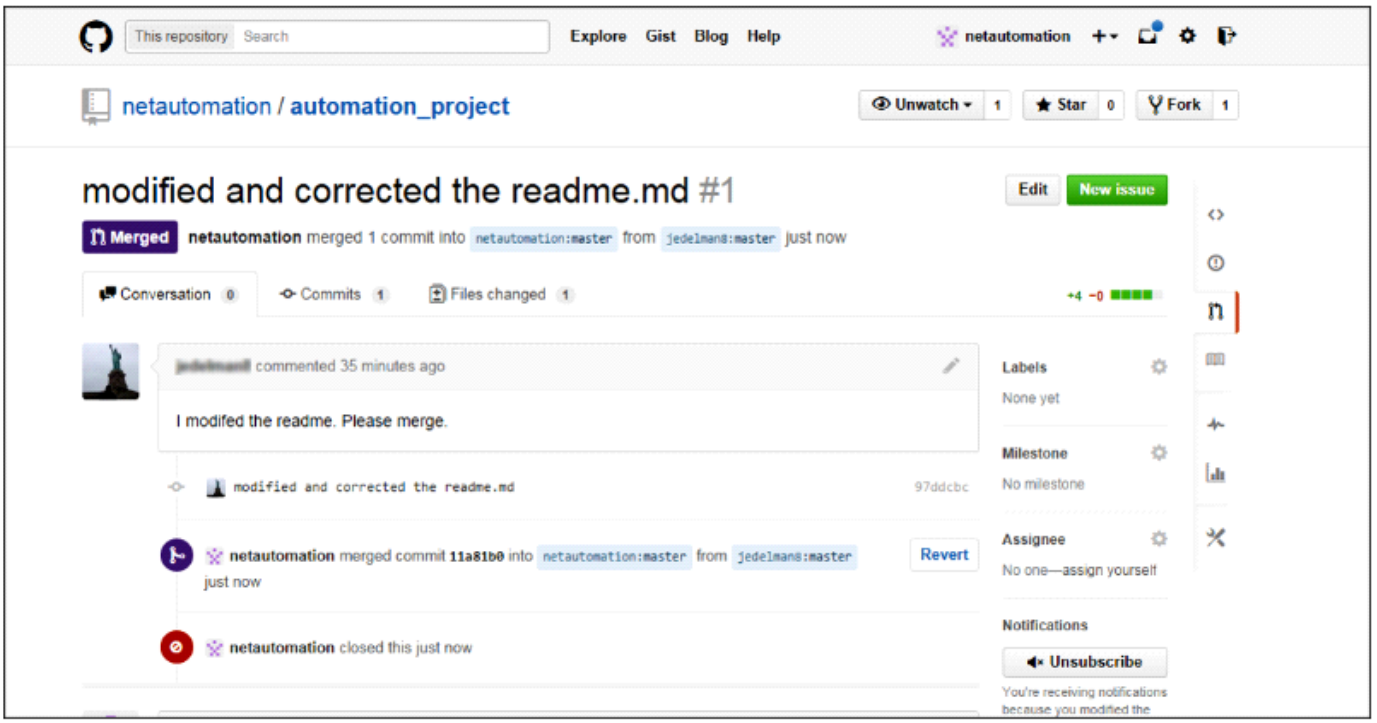
Here you can see that a single PR has been opened.

Step 3: Review (comment/approve) the Pull Request



Once you click the Pull Request, you view the description and all the changes that are made in the commit. For example: view the diffs, and the like. If needed, you can provide feedback to the contributor as part of the code review process. If everything looks good, you can then press the "Merge Pull Request" button.

Final View of Merge



Finally, you can see that PR has been merged after clicking the Merge Pull Request button.

From <https://ondemandelearning.cisco.com/cisco-cte/npdesi10/sections/23/pages/9>