**Key GitHub Features**

In addition to being a place to host and share your Git projects, GitHub provides features that make collaborating easier.

* The **Code View** lets you see your project directory and code, as well as its history.
* **Issues** are general places for discussions about the project, from high-level planning to bug tracking.
* **Pull Requests** let you discuss and review specific changes to the project files themselves.
* The **Pulse** gives you a high-level overview of how the project is proceding.
* You can use the **Wiki** to add documentation to your project, although most people just use markdown files in the repository.
* **Graphs** tell you who has been contributing what, when, to your project.

In this course we'll heavily focus on the first three features, but once you're done feel free to experiment with the others!

# The GitHub Ecosystem

You can also use all of your favorite tools with GitHub. You don't need to use a new text editor or bug tracker: your team gets to keep the tools they're comfortable with.

GitHub can also integrate into your current deployment process, and fully supports continuous integration and continuous deployment to help you and your team build software better, together.

For more information on integrating GitHub with the tools you already use, refer to [GitHub's Integrations page](https://github.com/integrations).

# GitHub Repositories

A repository is the most basic element of GitHub. It is easiest to imagine as a project's folder. However, unlike an ordinary folder on your laptop, a GitHub repository offers simple yet powerful tools for collaborating with others. A repository contains all of the project files, and stores the history of each file. Whether you are just curious or you are a major contributor, knowing your way around a repository is essential!

# Your First GitHub Repository

We've created a repository for you on GitHub. You'll use this to explore how GitHub works. When we ask you to complete tasks on GitHub, we'll be referring to this repository.

**There is a link to your repo at the top of this page in the navigation bar**. If you check the email associated with your GitHub account, you should find some emails that link you to the repository.

**Exploring a Repository: Recap**

* The repository (or "repo") holds everything in your project—code, documentation, other important files.
* Everything in the Code View of the repo is tracked by git version control.
* The README.md file is shown at the bottom of the main page of the code view.
* Issues are used to track bugs and feature requests, and can be assigned to team members.
* A pull request represents a change that the author would like to make to the repo. PRs are used to resolve issues.

**Using Issues**

Use GitHub issues to record and discuss ideas, enhancements, tasks, and bugs. They make collaboration easier in a variety of ways, such as:

* Replacing email for project discussions, ensuring everyone on the team has the complete story.
* Allowing you to cross-link to other issues and pull requests.
* Creating a single, comprehensive record of how and why you made certain decisions.
* Allowing you to easily pull the right people into a conversation.

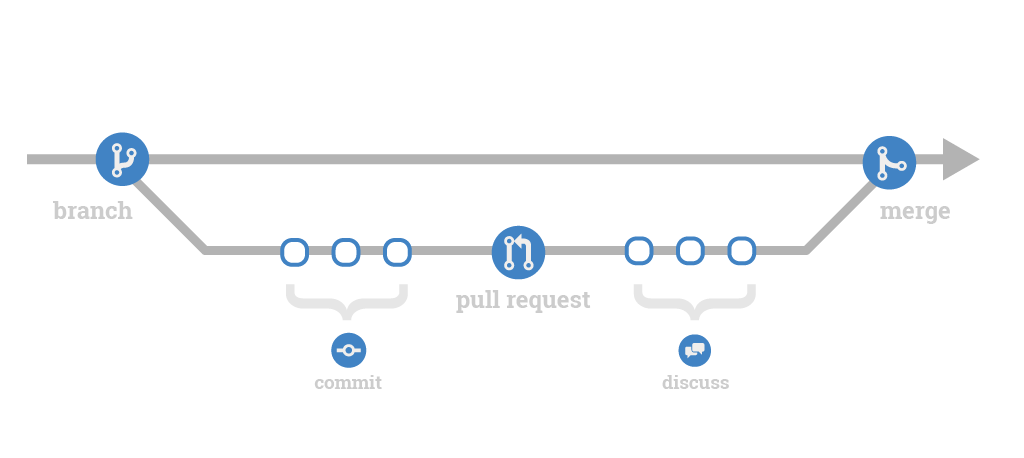
**Using Issues: Recap**

* You can use [Markdown](https://help.github.com/articles/markdown-basics/) to format issues.
* You can @mention somebody in order to alert them of the issue.
* The "Preview" tab shows how your comment will be rendered.
* Add screenshots or pictures to the issue by simply dragging and dropping them into the comment field.
* You can assign an issue to anyone with access to the repo.
* Labels can be used to organize issues.
* You can filter issues by label, author or assignee.

# Diving into GitHub

In this section we'll explore GitHub's functionality in depth.

You'll have the chance to practice all the skills that we teach in your GitHub repository, so make sure not to move on until you're sure you can perform these actions: you'll need them in your day-to-day job.



**Creating Files on GitHub: Recap**

* You can create a new file directly in the web interface.
* Once you're done editing the file, "commit" the changes.
* Make sure to include a commit message, and that you've chosen to commit to the branch you've already created.

**Creating a New File**

**1**

* Add a new bio file to the docs folder. Name the file username.md using your GitHub user name. Add the information from the Issue you created earlier to the file.

*Great job - you created a new file*

# Pull Requests

In this section we'll go over how to use Pull Requests. Pull Requests, or "PRs", are one of the most important parts of the GitHub Flow.

A PR is a request to merge one branch into another. They're used to discuss the changes made in the branch, and to continue to make changes until the branch is complete and the team agrees it can be merged. You'll use PRs frequently when working with your team, as they're the primary way to bring code on the master branch up to date with your feature branch once it's ready for production.

**Understanding Pull Requests: Recap**

* Pull Requests (or PRs) are used to add code to another branch, in this case master.
* A Pull Request asks that the team merge one branch into another.
* The "base" is the branch you want to merge into (often master), the "compare" is the branch you're merging, in this case the feature branch.
* Make sure to leave a comment to tell people what changes you made and why.
* In the comment, you should put number sign (#) in front of the number of the issue that you're fixing. This will automatically link your PR to that issue.
* Once you've made the PR, the Files changed view allows you to see what this PR will change on the base branch. Red means "deleted", green means "added". This view is often called the "diff".
* You can add comments directly to the diff, or in the Conversations view.

**Creating a Pull Request**

**1**

* Create a Pull Request to merge your firstname-lastname branch intomaster, @mentioning Hubyeti. Assign the Pull Request to yourself.

*Great job - you created a Pull Request*

**2**

* Go to the "Files changed" tab and add a line comment to the Pull Request mentioning that you should add your favorite color.

*It looks like you haven't commented on a pull request on GitHub yet.*

**Editing Pull Request Files: Recap**

* You will be notified of all comments on your own PRs, unless you've unsubscribed from that PR or ignored the whole repository.
* You can make changes to files directly in the Files changed view.
* These changes, once committed, will be included in the PR.

**Editing Pull Request Files**

**1**

* Go back to your branch and add your favorite color to your bio file. Feel free to lie if you don't want this info to be public. Commit the changes directly to your branch. Then go look at your PR for that branch.

*It looks like you haven't edited a file on your pull request on GitHub yet.*

# Merging Pull Requests

At this point, your pull request should be approved and you are ready to merge it in to the master branch. When you merge your branch, you are taking the content and history from your feature branch and adding it to the content and history of the master branch.

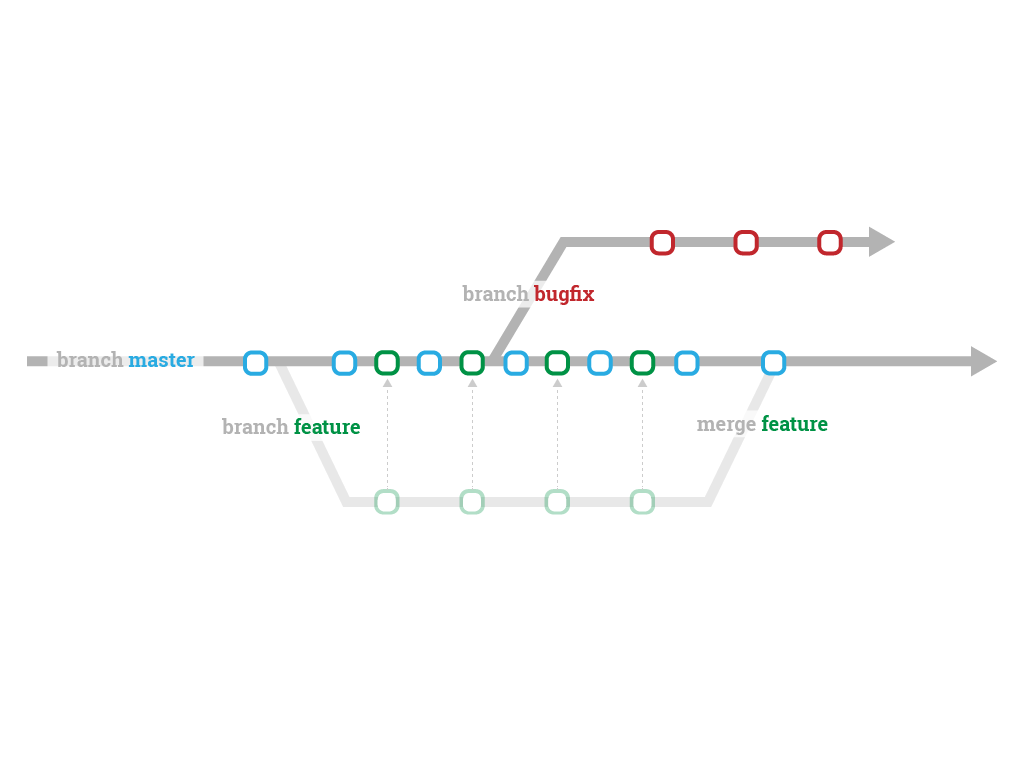
Many project teams have established rules about who should merge a pull request. Some say it should be the person who created the pull request since they will be the ones to deal with any issues resulting from the merge. Others say it should be a single person within the project team to ensure consistency. Still others say it can be anyone other than the person who created the pull request. There are also third party Continuous Integration (CI) tools you can integrate with GitHub to test the build before the merge is completed.

There are pros and cons to each approach and we will not attempt to prescribe a solution here, but these are good conversations to have within your project team.

Let's take a look at how you can merge the pull request and close the original issue at the same time.

**Merging Pull Requests: Recap**

* To merge your branch into the master branch, click the Merge pull request button in the Conversation view.
* Use the keyword Fixes followed by # and the issue number to close the issue at the same time as you merge the PR.
  + A full list of keywords can be found [here](https://help.github.com/articles/closing-issues-via-commit-messages/).
* Once the PR is successfully merged, you can delete your feature branch as it is no longer needed.



# [GitHub Help](https://help.github.com/)

* Version
* [**Contact Support**](https://github.com/contact)
* [**Return to GitHub**](http://www.github.com/)

###### [Commits](https://help.github.com/categories/commits) / Closing issues via commit messages

Top of Form



Bottom of Form

## **Closing issues via commit messages**

You can include keywords in your commit messages to automatically close issues in GitHub.

### Closing an issue in the same repository

To close an issue in the same repository, use one of the keywords in the list below followed by a reference to the issue number in the commit message. For example, a commit message with Fixes #45 will close issue 45 in that repository once the commit is merged into the default branch.

If the commit is in a non-default branch, the issue will remain open and the issue will be referenced with a tooltip.

#### Keywords for closing issues

The following keywords will close an issue via commit message:

* close
* closes
* closed
* fix
* fixes
* fixed
* resolve
* resolves
* resolved

#### Closing an issue in a different repository

To close an issue in another repository, use the username/repository#issue\_number syntax, as described in "[Autolinked references and URLs](https://help.github.com/articles/autolinked-references-and-urls/" \l "issues-and-pull-requests)".

For example, including Closes example\_user/example\_repo#76 will close the referenced issue in that repository, provided you have push access to that repository.

#### Closing multiple issues

To close multiple issues, preface each issue reference with one of the above keywords.

For example, This closes #34, closes #23, and closes example\_user/example\_repo#42 would close issues #34 and #23 in the same repository, and issue #42 in the "example\_user/example\_repo" repository.

**Using the Command Line**

So far, all the work we've done has been using GitHub's web interface. However, if you prefer to work on the command line, you can easily integrate GitHub into your current workflow.

In this section, we'll teach you how to use Git and GitHub effectively through the command line. We assume a working knowledge of the command line interface, so if you're rusty you might want to brush up on using the CLI before forging ahead.

To get to the command line, you should do the following:

* **Mac**: Type command-space to bring up Spotlight search, then search for Terminal.
* **Linux**: Open up your prefered shell, such as bash or fish.
* **Windows**: We recommend that you download [Git for Windows from GitHub](https://msysgit.github.io/" \t "_blank), as it includes a bash emulator. Without it, the Windows CLI will look a lot different from the Mac / Linux CLI.

If you've never used the command line before, you might want to [learn more about it here](http://lifehacker.com/5633909/who-needs-a-mouse-learn-to-use-the-command-line-for-almost-anything).

# Git vs. GitHub

When using the command line, you'll be working with Git. Git is the underlying version control system that GitHub makes use of. Some of the features we've discussed so far, such as pull request and issues, are functionality that GitHub adds. Others, like branching and commits, are Git functionality.

You can think of Git as a version control tool that you can use locally, and GitHub as a remote service that makes using that tool with other people on your team much more functional and seamless.

# Checking Git Version

Git is OS agnostic; the commands are essentially the same whether you are on Mac, Windows or Linux. You can use your favorite application to interact with the command line.

First, let's check the version of Git currently installed on your system by typing git --version into the command line. Compare that to the version of Git available [from Git's website](https://git-scm.com/).

If you don't have the right version, or if you get an error saying Git isn't installed, then go to [git-scm.com](http://git-scm.com/downloads) to download and install the latest version.

Once you've verified you have the latest version of Git installed, you're ready to move on to getting Git configured correctly

**Git Configuration Levels**

The first thing you should do when you get started using Git is to set your configuration options.

Git allows you to set configuration options at three different levels. The default value for Git config is --local.

* --system - These are system-wide configurations. They apply to all users on this computer.
* --global - These are the user level configurations. They only apply to your user account.
* --local - These are the repository level configurations. They only apply to the specific repository where they are set.

# Pre-viewing Configuration Settings

If you would like to see which config settings have already been added, you can type git config --list. This will automatically read from each of the storage containers for config settings and list them.

# Configuring User Name and Email

Go ahead and follow along as we set up our basic configurations. Git uses the config settings for your user name and email address to generate a unique fingerprint for each of the commits you create, so let's set our user name and email address first.

Type git config --global user.name "<your\_full\_name>" and type git config --global user.email "<your\_email>".

Make sure to use the same email address that's associated with your GitHub account.

# Configuring Default Editor

Next, we will add a default text editor. This is the text editor Git will use when you need to edit things like commit messages or handle merge conflicts.

We recommend using GitHub's own open source text editor, "atom". If you don't have atom and would like to use it, you can find it at [atom.io](https://atom.io/).

Typing git config --global core.editor "atom --wait" will tell Git to use the open source atom text editor once atom is installed.

If you have a hard time getting Atom set up or would like to use a different editor such as vim, you can look for instructions[here](https://help.github.com/articles/associating-text-editors-with-git/). Then when you see us reference atom in commands later on in the course, just use whichever text editor you decided upon.

**Configuring Autocrlf to Handle Whitespace**

Different systems handle line endings and line breaks differently. If you open a file created on another system and do not have autocrlf set, Git will think you made changes to the file based on the way your system handles this type of file.

Type git config --global core.autocrlf.

* If you are on a **Windows** machine, you will want to set this option to true.
* If you are on a **Mac** or **Linux** machine, you will set it to input.

And for those wondering, autocrlf stands for "auto carriage return line feed".

# Configuring Default Push Behavior

One final configuration option we will want to set is our default value for push.

When you push changes from your local computer to the remote you can choose whether you want Git to automatically push all of the local branches to their matching branches on the remote or whether you only want the currently checked out branch to be pushed.

The config setting we use to set this option is push.default. We can set the default to matching if we want to push all branches automatically. OR, we can set it to simple if we only want to push the branch we are on.

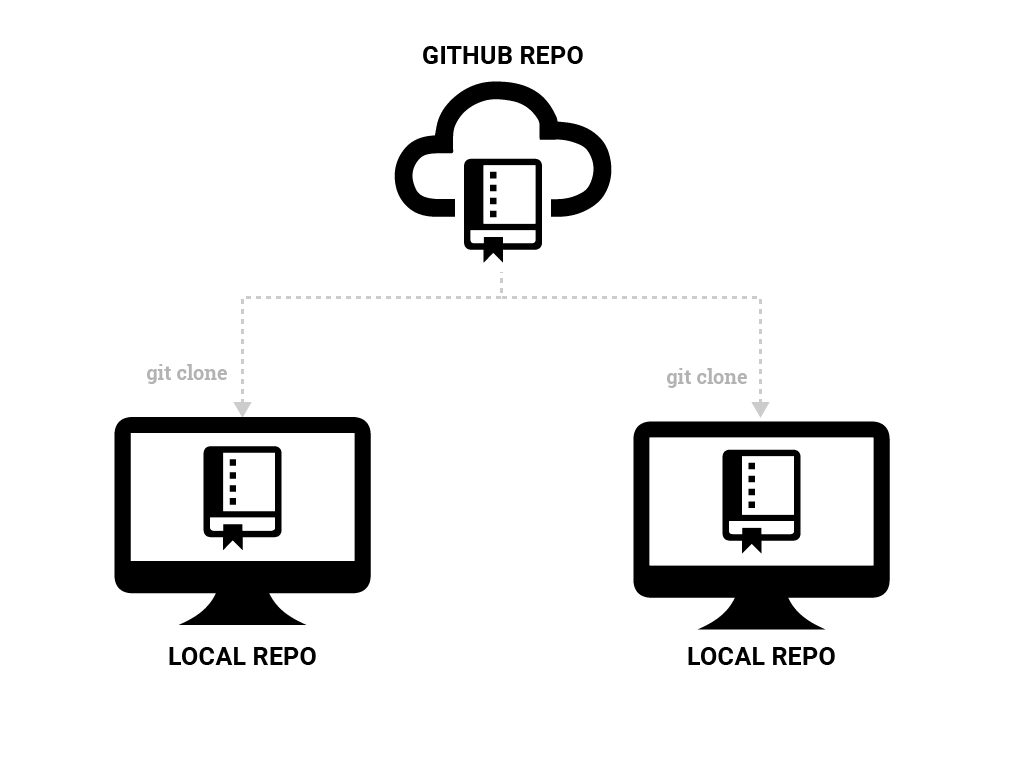
For now, let's use git config --global push.default simple. This way you don't make accidental pushes while you're still learning to use Git.

# Cloning a Repository

Part of what makes git such a well-loved VCS is that it easily lets everyone on the team have their own local working repository. It's expected that everyone will work on their own computer, and sync changes they've made with each other only when they need to.

But first, we have to make a local copy of your repository. To do this, we have to "clone" the repository that you have hosted on GitHub.

When you clone a GitHub repository you are creating a copy of everything in that repository, including its history. This is one of the benefits of a distributed VCS like git—rather than being required to query a slow centralized server to review the commit history, queries are run locally and are lightning fast.



**Cloning a Repository: Recap**

* Start a new branch on GitHub to work on.
* Copy the clone URL on GitHub.
* Using the command line, cd to the directory that you want to put your local repo in.
* Type git clone <clone\_url>.
  + You might be prompted for your password. You should enter your GitHub password at this point.
* cd into the new cloned repo and type git status.
* git branch will show you the available branches. Only master appears.
* git branch -a will show all the remote branches.
* git checkout <branchname> will create a local branch that matches the remote branch. All edits you make locally will be applied to that branch until you checkout another one.

**Cloning a Repository**

**1**

* Create a new branch on GitHub. Name the branch githubID-more-bio, using your GitHub ID.

*Great job - you created a branch.*

**2**

* Clone the repository to your desktop using the Command Line.

**3**

* Checkout a local copy of the branch you created on GitHub.

# Editing Local Files

Now that you have cloned the repository and checked out your branch, you are ready to make some changes to the local files. If you are familiar with using the command line to open and edit files, then much of this will be familiar to you. The only difference here is that we will be making our changes on a branch. Let's add more information to the bio we created earlier.

**Editing Local Files: Recap**

* git branch to make sure you're on the right branch.
* cd to the proper directory, then open the file with your text editor.
* Make the changes, then save them.

# The Two Stage Commit

After you have finished making your changes, it is time to commit them. When working from the command line, you will need to be familiar with the idea of the two stage commit.

# The Two Stage Commit

When you work locally, your files exist in one of four states. They are either untracked, modified, staged, or committed.

An untracked file is one that is not currently part of the version controlled directory—such as any new files or files whose names have been changed.

Modified files are any files that you have edited and saved since the last time you committed.

Untracked and modified files exist in your **working directory**.

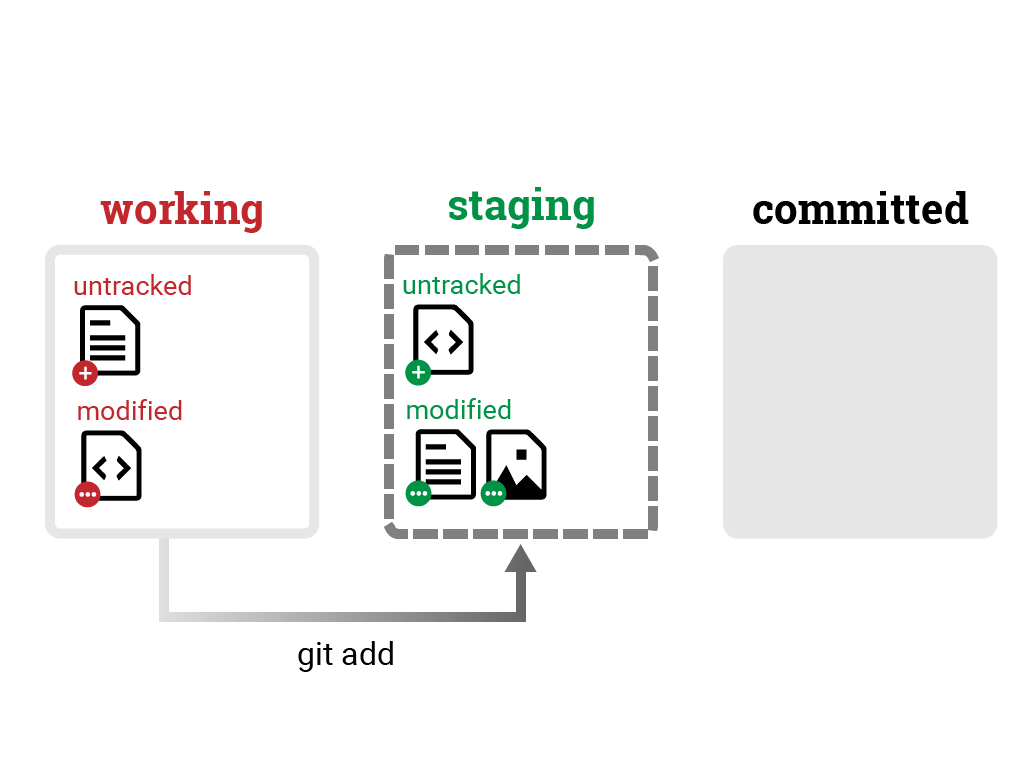
# The Two Stage Commit

To add these files to version control, you will create a collection of files that represent a discrete unit of work. This way if you want to revert something in the future, it's easier to find and revert only the mistaken changes.

We build this unit of work in the **staging area**.

To add files to staging, you use the git add <filename> command.

Staging is important because it lets you only stage some of your new or modified files. For instance, if you've added a new variable and fixed a typo, you might want to commit those two changes separately. To do this, you stage and commit each one separately.



# The Two Stage Commit

To add these files to version control, you will create a collection of files that represent a discrete unit of work. This way if you want to revert something in the future, it's easier to find and revert only the mistaken changes.

We build this unit of work in the **staging area**.

To add files to staging, you use the git add <filename> command.

Staging is important because it lets you only stage some of your new or modified files. For instance, if you've added a new variable and fixed a typo, you might want to commit those two changes separately. To do this, you stage and commit each one separately.

# The Two Stage Commit

When we are satisfied with the unit of work we have assembled, we will commit everything in the staging area.

To commit files, use the git commit command.

Once changes are committed, the changes are added to the history of that branch, so that you can later see those specific changes, and even undo them if need be.

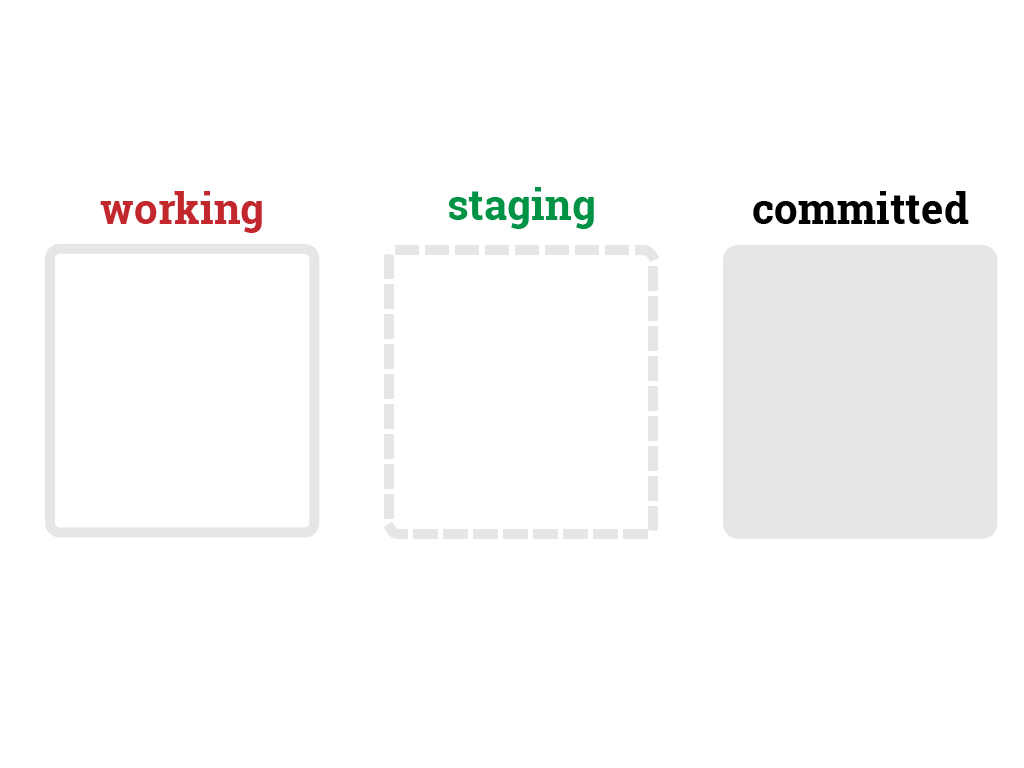
# The Two Stage Commit

To review:

In order to make a file part of the version controlled directory we will first do a git add of only those files we want to commit together, and then we will do a git commit to add those changes to the Git version control history. Once the changes are commited, you'll always have a record of them in that state, even after you've changed them again in the working directory.

All of this happens locally, on your computer.

Let's do it now.

* 

# The Two Stage Commit

When you work locally, your files exist in one of four states. They are either untracked, modified, staged, or committed.

An untracked file is one that is not currently part of the version controlled directory—such as any new files or files whose names have been changed.

Modified files are any files that you have edited and saved since the last time you committed.

Untracked and modified files exist in your **working directory**.