# The Official GitHub Training Manual

# Table of Contents

## Getting Started



## Project 1: Caption This



## Project 2: Merge Conflicts



## Project 3: GitHub Games

# Project 4: Adding More Tests



# Project 5: Local Repository



# Appendix

# Welcome to GitHub for Developers

> Note: This version of the manual is newer, and contains activities from our new curriculum. You can find the legacy version of our training manual here.

You can download a PDF version of this manual here.

Today you will embark on an exciting new adventure: learning how to use Git and GitHub.

As we move through today's materials, please keep in mind: this class is for you! Be sure to follow along, try the activities, and ask lots of questions!

## License

# Getting Ready for Class

While you are waiting for class to begin, please take a few minutes to set up your local work environment.

## Step 1: Set Up Your GitHub.com Account

For this class, we will use a public account on GitHub.com. We do this for a few reasons:

- We don't want you to "practice" in repositories that contain real code.
- We are going to break some things so we can teach you how to fix them. (therefore, refer to the bullet above)

If you already have a github.com account you can skip this step. Otherwise, you can set up your free account by following these steps:

1. Access GitHub.com and click Sign up.
2. Choose the free account.
3. You will receive a verification email at the address provided.
4. Click the link to complete the verification process.

## Step 2: Install Git

Git is an open source version control application. You will need Git installed for this class.

You may already have Git installed so let's check! Open Terminal if you are on a Mac, or PowerShell if you are on a Windows machine, and type:

```
$ git --version
```

You should see something like this:

```
$ git --version
git version 2.11.0
```

Anything over 2.0 will work for this class!

## Downloading and Installing Git

If you don't already have Git installed, you can download Git at www.git-scm.com.

If you need additional assistance installing Git, you can find more information in the ProGit chapter on installing Git: `http://git-scm.com/book/en/v2/Getting-Started-Installing-Git` .

## Where is Your Shell?

Now is a good time to create a shortcut to the command line application you will want to use with Git:

- If you are working on Windows, you can use `Git Bash` which is installed with the Git package or you can use Powershell with Posh-git.
- If you are working on a Mac or other Unix based system, you can use the terminal application.

Go ahead and open your command line application now!

## Step 3: Set Up Your Text Editor

For this class, we will use a basic text editor to interact with our code. Let's make sure you have one installed and ready to work from the command line.

## Pick Your Editor

You can use almost any text editor, but we have the best success with the following:

- Atom
- GitPad
- Vi or Vim
- Sublime
- Notepad or Notepad++

If you do not already have a text editor installed, go ahead and download and install one of the above editors now!

## Your Editor on the Command Line

After you have installed an editor, confirm you can open it from the command line.

If installed properly, the following command will open the Atom text editor:

```
$ atom .
```

> If you are working on a Mac, you will need to Install Shell Commands from the Atom menu, this happens as part of the installation process for Windows.

## Exploring

Congratulations! You should now have a working version of Git and a text editor on your system. If you still have some time before class begins, here are some interesting resources you can check out:

- *github.com/explore* Explore is a showcase of interesting projects in the GitHub Universe. See something you want to re-visit? Star the repository to make it easier to find later.
- *services.github.com/on-demand* Our On Demand Training courses are GitHub's open source training materials. The site contains additional resources you may find helpful when reviewing what you have learned in class! You can even make contributions to the materials or open issues if you would like us to explain something in greater detail. Find the open source repository here: `https://github.com/github/training-kit`

# Getting Started With Collaboration

We will start by introducing you to Git, GitHub, and the collaboration features we will use throughout the class. Even if you have used GitHub in the past, we hope this information will provide a baseline understanding of how to use it to build better software!

## What is GitHub?

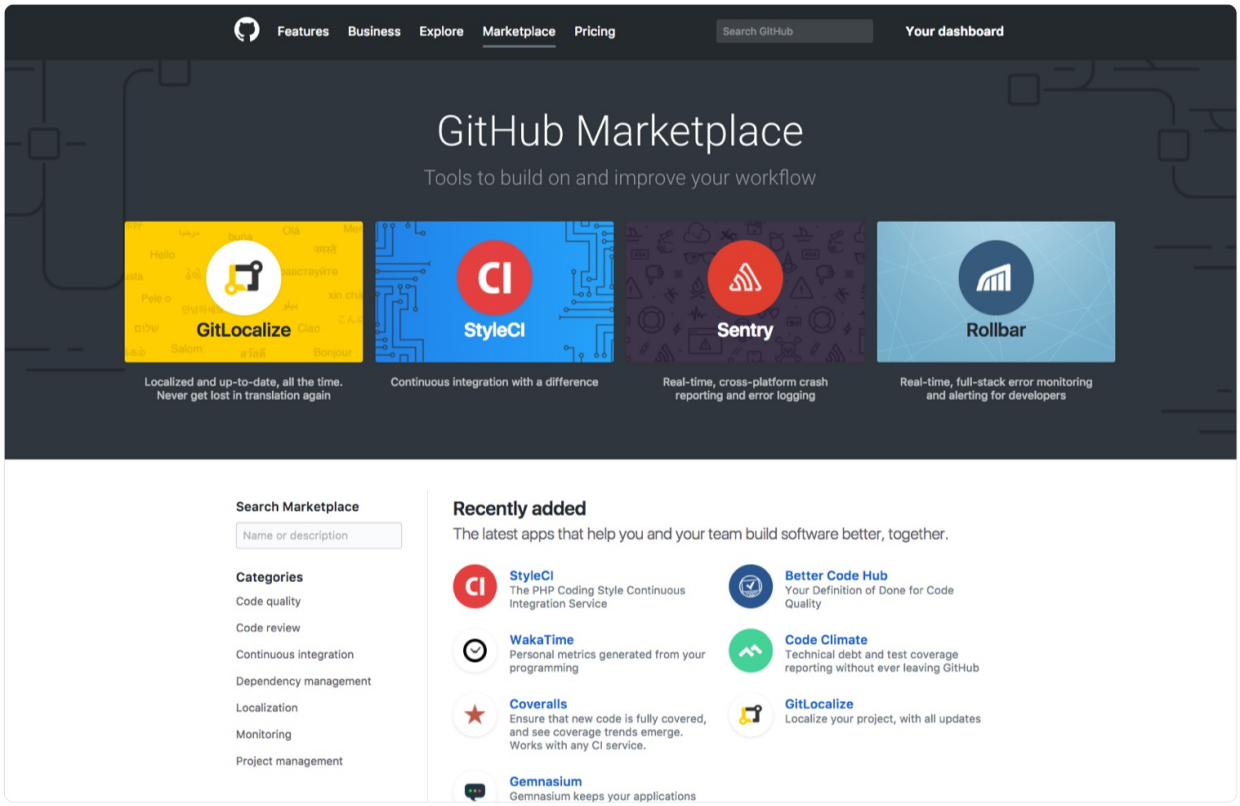
GitHub is a collaboration platform built on top of a distributed version control system called Git.

In addition to being a place to host and share your Git projects, GitHub provides a number of features to help you and your team collaborate more effectively. These features include:

- Issues
- Pull Requests
- Projects
- Organizations and Teams


## The GitHub Ecosystem

Rather than force you into a "one size fits all" ecosystem, GitHub strives to be the place that brings all of your favorite tools together. For more information on integrations, check out https://github.com/integrations.




You may even find some new, indispensable tools to help with continuous integration, dependency management, code quality and much more.

# What is Git?

**Git is:**

- a distributed version control system or DVCS.
- free and open source.
- designed to handle everything from small to very large projects with speed and efficiency.
- easy to learn and has a tiny footprint with lightning fast performance.

Git features cheap local branching, convenient staging areas, and multiple workflows.

As we begin to discuss Git (and what makes it special) it would be helpful if you could forget everything you know about other version control systems (VCSs) for just a moment. Git stores and thinks about information very differently than other VCSs.

We will learn more about how Git stores your code as we go through this class, but the first thing you will need to understand is how Git works with your content.

## Snapshots, not Deltas

One of the first ideas you will need understand is that Git does not store your information as series of changes. Instead Git takes a snapshot of your repository at a given point in time. This snapshot is called a commit.

## Optimized for Local Operations

Git is optimized for local operation. When you clone a copy of a repository to your local machine, you receive a copy of the entire repository and its history. This means you can work on the plane, on the train, or anywhere else your adventures find you!

## Branches are Lightweight and Cheap

Branches are an essential concept in Git.

When you create a new branch in Git, you are actually just creating a pointer that corresponds to the most recent snapshot in a line of work. Git keeps the snapshots for each branch separate until you explicitly tell it to merge those snapshots into the main line of work.

## Git is Explicit

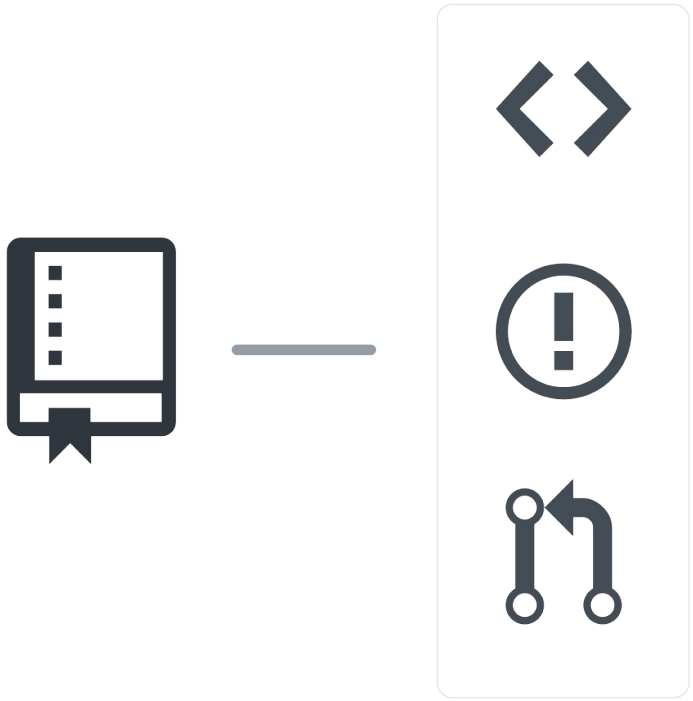
Which brings us to our final point for now; Git is very explicit. It does not do anything until you tell it to. No auto-saves or auto-syncing with the remote, Git waits for you to tell it when to take a snapshot and when to send that snapshot to the remote.

## Exploring a GitHub Repository

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 (including documentation), and stores each

file's revision history. Whether you are just curious or you are a major contributor, knowing your way around a repository is essential!



## User Accounts vs. Organization Accounts

There are two account types in GitHub, user accounts and organization accounts. While there are many differences in these account types, on of the more notable differences is how you handle permissions.

## User Accounts

When you signed up for GitHub, you were automatically given a user account. Permissions for a user account are simple, you add people as collaborators to specific repositories to give them full read-write access to the project.

## Organization Accounts

Organization accounts provide more granular control over repository permissions. In an organization account you create teams of people and then give those teams access to specific repositories. Permissions can be assigned at the team level (e.g, read, write, or

admin).

## Repository Navigation

## Code

The code view is where you will find the files included in the repository. These files may contain the project code, documentation, and other important files. We also call this view the root of the project. Any changes to these files will be tracked via Git version control.

## Issues

Issues are used to track bugs and feature requests. Issues can be assigned to specific team members and are designed to encourage discussion and collaboration.

## Pull Requests

A Pull Request represents a change, such as adding, modifying, or deleting files, which the author would like to make to the repository. Pull Requests help you write better software by facilitating code review and showing the status of any automated tests.

## Projects

Projects allow you to visualize your work with Kanban style boards. Projects can be created at the repository or organization level.

## Wiki

Wikis in GitHub can be used to communicate project details, display user documentation, or almost anything your heart desires. And of course, GitHub helps you keep track of the edits to your Wiki!

## Pulse

Pulse is your project's dash board. It contains information on the work that has been completed and the work in progress.

## Graphs

Graphs provide a more granular view into the repository activity, including who has contributed, when the work is being done, and who has forked the repository.

## README.md

The README.md is a special file that we recommend all repositories contain. GitHub looks for this file and helpfully displays it below the repository. The README should explain the project and point readers to helpful information within the project.

## CONTRIBUTING.md

The CONTRIBUTING.md is another special file that is used to describe the process for collaborating on the repository. The link to the CONTRIBUTING.md file is shown when a user attempts to create a new issue or pull request.

## ISSUE_TEMPLATE.md

The ISSUE_TEMPLATE.md (and its twin the pull request template) are used to generate templated starter text for your project issues. Any time someone opens an issue, the content in the template will be pre-populated in the issue body.

## Using GitHub Issues

In GitHub, you will use issues to record and discuss ideas, enhancements, tasks, and bugs. Issues make collaboration easier by:

- Replacing email for project discussions, ensuring everyone on the team has the complete story, both now and in the future.
- Allowing you to cross-link to related 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 with @ mentions and team mentions.

## Activity: Creating A GitHub Issue

Follow these steps to create an issue in the class repository:

1. Click the *Issues* tab.
2. Click *New Issue*.
3. Type the following in the Subject line: `YOUR-USERNAME Workflow`
4. In the body of the issue, include the text below:

```
YOUR-USERNAME will choose an image, add a caption, and add both to a file.

- [ ] Create a branch
- [ ] Edit the file
- [ ] Commit the changes
- [ ] Create a Pull Request
- [ ] Request a Review
- [ ] Make more changes
- [ ] Get an approval
- [ ] Merge the Pull Request
```

# Using Markdown

GitHub uses a syntax called **Markdown** to help you add basic text formatting to Issues, Pull Requests, and files with the `.md` extension.

# Commonly Used Markdown Syntax

**`# Header`**

The `#` indicates a Header. # = Header 1, ## = Header 2, etc.

**`* List item`**

A single `*` or `-` followed by a space will create a bulleted list.

**`**Bold item**`**

Two asterix `**` on either side of a string will make that text bold.

**`- [ ] Checklist`**

A `-` followed by a space and `[ ]` will create a handy checklist in your issue or pull request.

### `@mention`

When you @mention someone in an issue, they will receive a notification - even if they are not currently subscribed to the issue or watching the repository.

### `#975`

A `#` followed by the number of an issue or pull request (without a space) in the same repository will create a cross-link.

### `:smiley:`

Tone is easily lost in written communication. To help, GitHub allows you to drop emoji into your comments. Simply surround the emoji id with `:`.

## Introduction to GitHub Pages

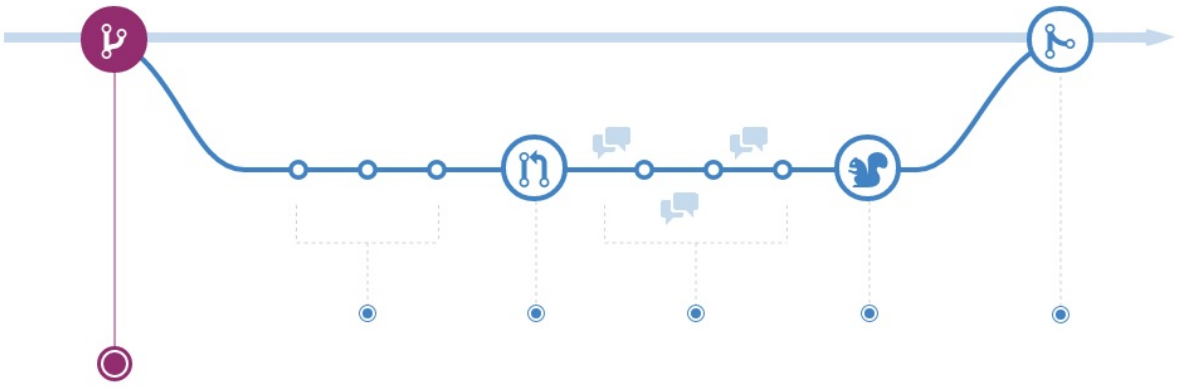
GitHub Pages enable you to host free, static web pages directly from your GitHub repositories. Several of the projects we use in class will use GitHub Pages as the deployment strategy. We will barely scratch the surface in this class, but there are a few things you need to know:

- You can create two types of websites, a user/organization site or a project site. We will be working with project websites.
- For a project site, GitHub will only serve the content on a specific branch. Depending on the settings for your repository, GitHub can serve your site from a `master` or `gh-pages` branch, or a `/docs` folder on the `master` branch.
- The rendered sites for our projects will appear at `githubschool.github.io/repo-name`.

# Understanding the GitHub Flow

In this section, we will discuss the collaborative workflow enabled by GitHub.

## The Essential GitHub Workflow



The GitHub flow is a lightweight workflow that allows you to experiment with new ideas safely, without fear of compromising a project.

Branching is a key concept you will need to understand. Everything in GitHub lives on a branch. By default, the "blessed" or "canonical" version of your project lives on a branch called `master`. This branch can actually be named anything, as we will see in a few minutes.

When you are ready to experiment with a new feature or fix an issue, you create a new branch of the project. The branch will look exactly like `master` at first, but any changes you make will only be reflected in your branch. Such a new branch is often called a "feature" branch.

As you make changes to the files within the project, you will commit your changes to the feature branch.

When you are ready to start a discussion about your changes, you will open a pull request. A pull request doesn't need to be a perfect work of art - it is meant to be a starting point that will be further refined and polished through the efforts of the project team.

When the changes contained in the pull request are approved, the feature branch is

merged onto the master branch. In the next section, you will learn how to put this GitHub workflow into practice.

## Exploring

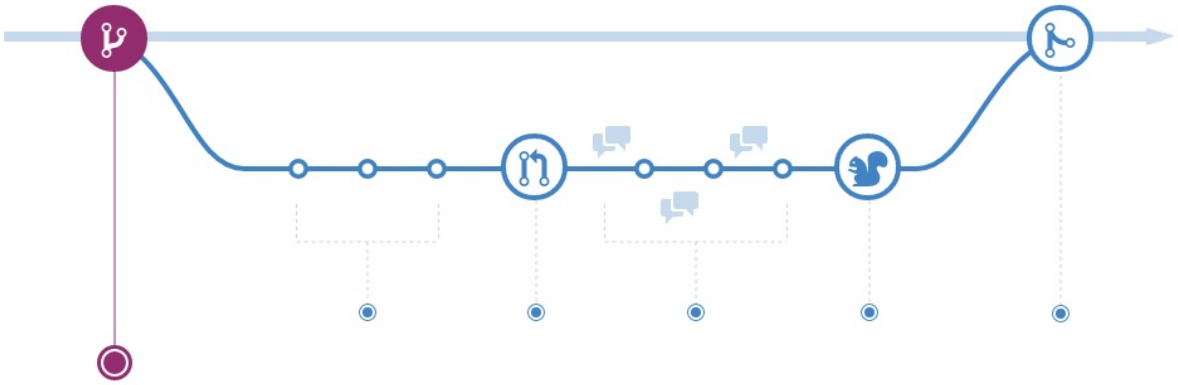
Here are some interesting things you can check out later:

- *guides.github.com/introduction/flow/* An interactive review of the GitHub Workflow.

# Branching with Git

The first step in the GitHub Workflow is to create a branch. This will allow us to experiment with new features without accidentally introducing untested changes on our production branch.

## Branching Defined



When you create a branch, you are essentially creating an identical copy of the project at that point in time. This isn't the same as creating a physical copy on disk. In the background, a branch is just a pointer.

Let's learn how you can create a new branch.

## Activity: Creating A Branch with GitHub

Earlier you created an issue about the file you would like to edit. Let's create the branch you will use to edit your file.

Follow these steps to create a new branch in the class repository:

> You will need to have collaborator access on the class repository before you can create a branch on GitHub.

1. Navigate to *Code* tab of the class repository.
2. Click the *branch dropdown*.
3. Enter the branch name 'github-username-caption'.

4. Press `Enter`.

When you create a new branch on GitHub, you are automatically switched to your branch. Now, any changes you make to the files in the repository will be applied to this new branch.

> A word of caution. When you return to the repository or click the top level repository link, notice that GitHub automatically assumes you want to see the items on the default branch. If you want to continue working on a feature branch, you will need to reselect it using the branch dropdown.

# Exploring

Here are some interesting things you can check out later:

- *https://youtu.be/H5GJfcp3p4Q* A GitHub Training Video on branching.

# Local Git Configuration

In this section, we will prepare your local environment to work with Git.

## Checking Your Git Version

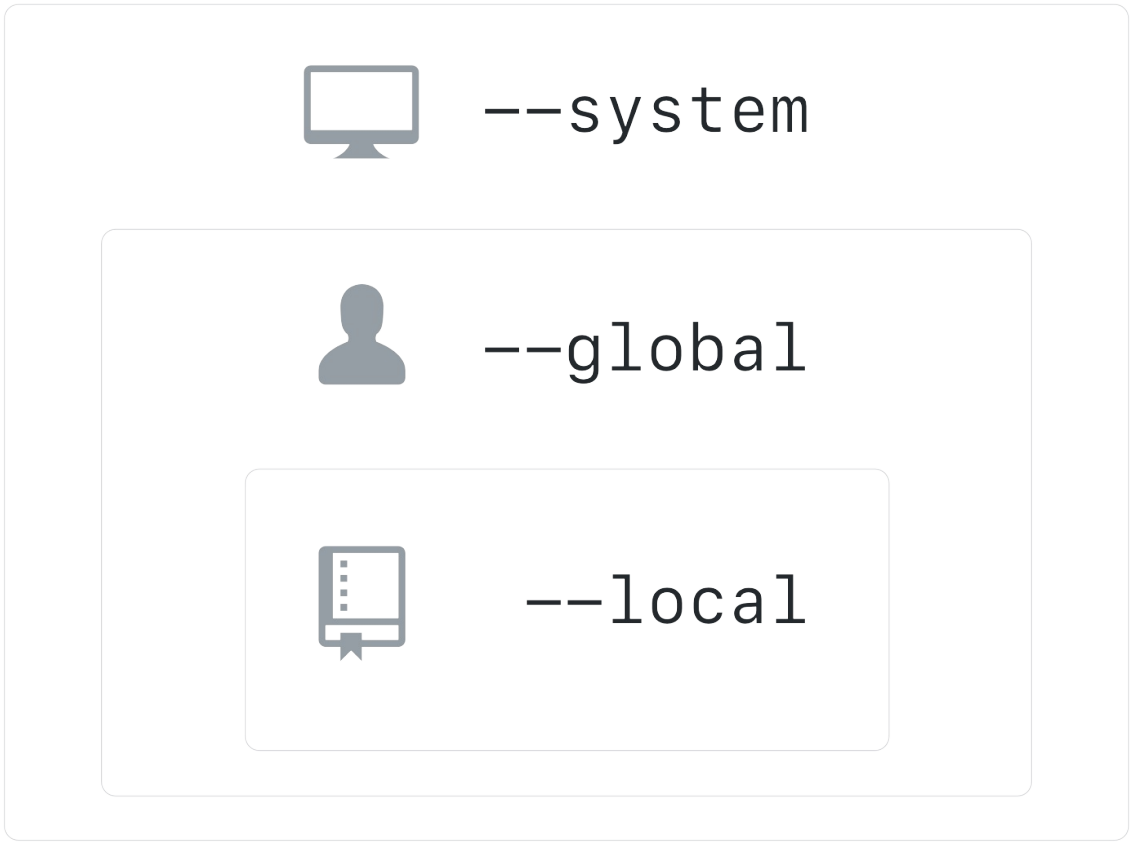
First, let's confirm your Git Installation:

```
$ git --version
$ git version 2.11.0
```

If you do not see a git version listed or this command returns an error, you may need to install Git.

> To get the latest version of Git, visit www.git-scm.com.

## Git Configuration Levels




Git allows you to set configuration options at three different levels.

## --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.

> The default value for git config is `--local`.

# Viewing Your Configurations

If you would like to see which config settings have been added automatically, you can type `git config --list`. This will automatically read from each of the three config files and list the setting they contain.

```
$ git config --list
```

You can also narrow the list to a specific configuration level by including it before the list option.

```
$ git config --global --list
```

## Configuring Your User Name and Email

Git uses the config settings for your user name and email address to generate a unique fingerprint for each of the commits you create. You can't create commits without these settings:

```
$ git config --global user.name "First Last"
$ git config --global user.email "you@email.com"
```

## Git Config and Your Privacy

The instructions for this exercise use the `--global` flag when identifying your `user.name` and `user.email` configuration settings. If you are currently using a computer without a private, personal account, don't apply the `--global` flag. This way, the settings will only be stored in our assignment repository. If you work in another repository on this same computer, you will need to set these configuration options again.

> For example:
>
> ```
> git config user.email "you@email.com"
> ```

Your name and email address will automatically be stored in the commits you make with Git. If you would like your email to remain private, GitHub allows you to generate a no-reply email address for your account. Click the **Keep my email address private** in the Settings > Emails section. After enabling this feature, you just need to enter the

automatically generated `ID+username@users.noreply.github.com` when configuring your email.

> For example:
>
> ```
> git config --global user.email 18249274+githubteacher@users.noreply.github.com
> ```

## Configuring autocrlf

```
$ //for Windows users
$ git config --global core.autocrlf true
$ //for Mac or Linux users
$ git config --global core.autocrlf input
```

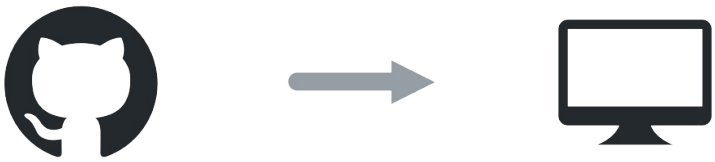
Different systems handle line endings and line breaks differently. If you open a file created on another system and do not have this config option set, git will think you made changes to the file based on the way your system handles this type of file.

> Memory Tip: `autocrlf` stands for auto carriage return line feed.

# Working Locally with Git

Using the command line, you can easily integrate Git into your current workflow.

## Creating a Local Copy of the repo



Before we can work locally, we will need to create a clone of the repository.

When you clone a repository you are creating a copy of everything in that repository, including its history. This is one of the benefits of a DVCS 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.

Let's go ahead and clone the class repository to your local desktop.

1. Navigate to the *Code* tab of the class repository on GitHub.
2. Click *Clone or download*.
3. Copy the *clone URL* to your clipboard.
4. Open your command line application.
5. Retrieve a full copy of the repository from GitHub: `git clone <CLONE-URL>`
6. Once the clone is complete, cd into the new directory created by the clone operation: `cd <REPOSITORY-NAME>`

## Our Favorite Git command: `git status`

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

`git status` is a command you will use often to verify the current state of your repository and the files it contains. Right now, we can see that we are on branch gh-pages, everything is up to date with origin/gh-pages and our working tree is clean.

## Using Branches locally

```
$ git branch
```

If you type `git branch` you will see a list of local branches.

```
$ git branch --all
$ git branch -a
```

If you want to see all of the branches, including the read-only copies of your remote branches, you can add the `--all` option or just `-a` .

> The `--all` and `-a` are actually synonyms for the branch command. Git often provides a verbose and a short option.

## Switching Branches

```
$ git checkout <BRANCH-NAME>
```

To checkout the branch you created online, type git checkout and the name of your branch. Git will provide a message that says you have been switched to the branch and it has been set up to track the same remote branch from origin.

> You do not need to type `remotes/origin` in front of the branch - only the branch name. Typing `remotes/origin` in front of the branch name will put you in a detached HEAD state. We will learn more about that later, but for now just remember this is not a state we want to be in.

## Activity: Edit Your File

Now it is time to put an image and a caption into your file:

1. Find your file, `named 2010-02-##-USERNAME.md`.
2. Open your file in your favorite text editor.
3. Copy and paste the image text into your file on line 6.
4. On lines 9 and 10, add code for your caption.

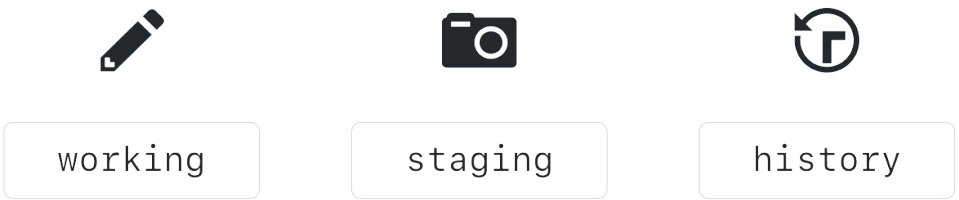
    ```
    ```
    CAPTION-HERE
    {: .fragment}
    ```
    ```

5. *Save* your file.

> Git doesn't care how you work with your files locally. You can work in your favorite IDE or text editor, or you can use VIM through the command line.
>
> To open your file from the command line, many IDEs and text editors offer shortcuts. For example, use `atom .` to open the project in Atom, and `code .` to open it in Visual Studio Code.

## The Two Stage Commit

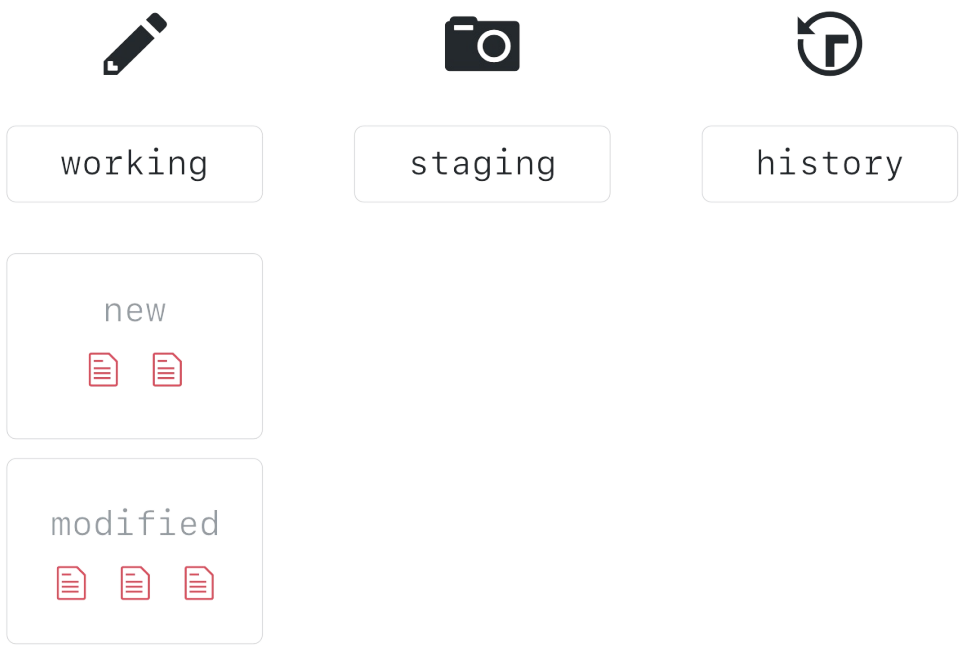
After you have created your file, it is time to create your first snapshot of the repository. When working from the command line, you will need to be familiar with the idea of the two stage commit.



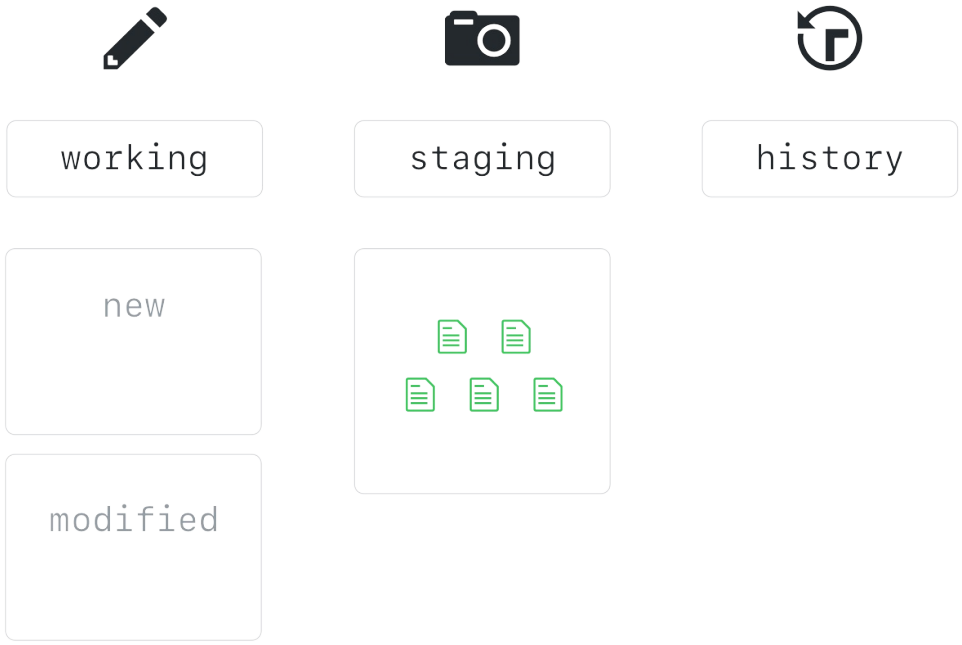
When you work locally, your files exist in one of four states. They are either untracked, modified, staged, or committed.

An untracked file is a new file that has never been committed.

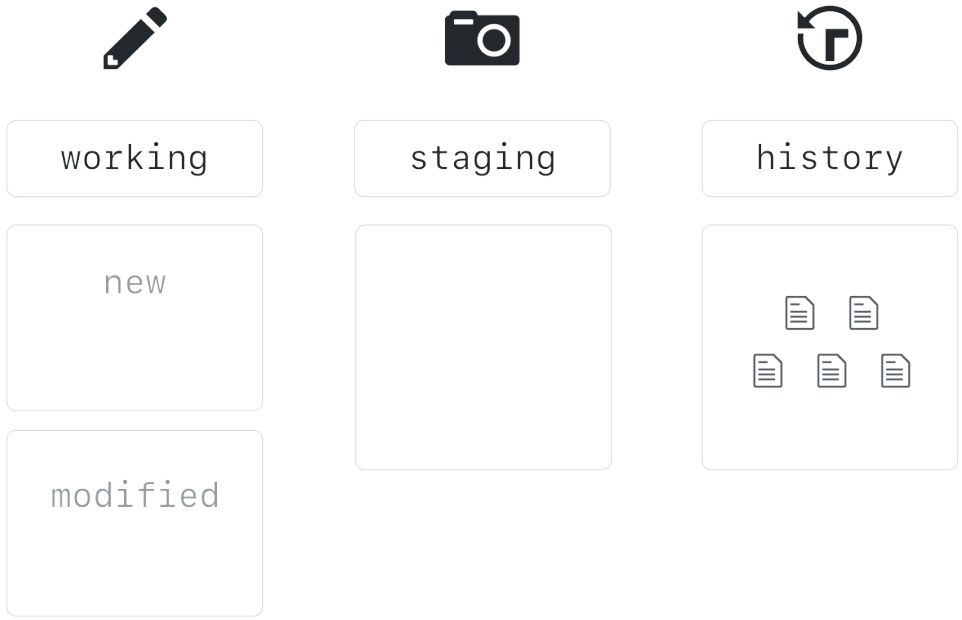
Git tracks these files, and keeps track of your history by organizing your files and changes in three working trees. They are Working, Staging (also called Index), and History. When we are actively making changes to files, this is happening in the working tree.




To add these files to version control, you will create a collection of files that represent a discrete unit of work. We build this unit in the staging area.




When we are satisfied with the unit of work we have assembled, we will take a snapshot of everything in the staging area. This is called a commit.




In order to make a file part of the version controlled directory we will first do a git add and then we will do a git commit. Let's do it now.

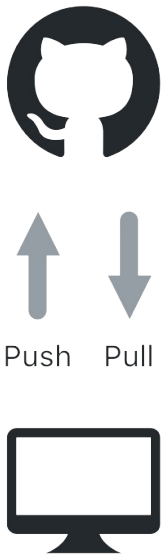
1. First, let's check the status of our working tree: `git status`
2. Move the file from the working tree to the staging area: `git add my-file.md`
3. Let's see what happened: `git status`
4. Now let's take our first snapshot: `git commit`
5. Git will open your default text editor to request a commit message. Simply type your message on the top line of the file. Any line without a # will be included in the commit message.
6. Save and close the commit message
7. Let's take another look at our repository status: `git status`

> Good commit messages should:
>
> - Be short. ~50 characters is ideal.
> - Describe the change introduced by the commit.
> - Tell the story of how your project has evolved.

# Collaborating on Your Code

Now that you have made some changes in the project locally, let's learn how to push your changes back to the shared class repository for collaboration.

## Pushing Your Changes to GitHub




In this case, our remote is GitHub.com, but this could also be your company's internal instance of GitHub Enterprise.

To push your changes to GitHub, you will use the command:

```
$ git push
```

> When you push, you will be asked to enter your GitHub username and password. If you would like Git to remember your credentials on this computer, you can cache your credentials using:
>
> - Windows: `git config --global credential.helper wincred`
> - Mac: `git config --global credential.helper osxkeychain`

# Activity: Creating a Pull Request

Pull Requests are used to propose changes to the project files. A pull request introduces an action that addresses an Issue. A Pull Request is considered a "work in progress" until it is merged into the project.

Now that you have started to change your file, you will open a pull request to discuss the file with your team mates. Follow these steps to create a Pull Request in the class repository:

1. Click the *Pull Request* tab.
2. Click *New Pull Request*.
3. In the *base* dropdown, choose `gh-pages`
4. In the *compare* dropdown, choose your branch.
5. Type a subject line and enter a comment.
6. Use markdown formatting to add a header and a checklist to your Pull Request.
7. Include one of the keywords: `closes` , `fixes` , or `resolves` followed by the issue number you created earlier to note which Issue the Pull Request should close. Example: `This resolves #3`
8. Click *Preview* to see how your Pull Request will look.
9. Assign the Pull Request to yourself.
10. Click *Create pull request.*

> When you navigate to the class repository, you should see a banner at the top of the page indicating you have recently pushed branches, along with a button that reads *Compare & pull request.* This helpful button will automatically start the pull request process between your branch and the repository's default branch.

# Exploring a Pull Request

Now that we have created a Pull Request, let's explore a few of the features that make Pull Requests the center of collaboration:

# Conversation view

Similar to the discussion thread on an Issue, a Pull Request contains a discussion about the changes being made to the repository. This discussion is found in the Conversation tab and also includes a record of all of the commits made on the branch as well as

assignments, labels and reviews that have been applied to the pull request.

## Commits view

The commits view contains more detailed information about who has made changes to the files. Clicking each commit ID will allow you to see the changes applied in that specific commit.

## Files changed view

The Files changed view allows you to see cumulative effect of all the changes made on the branch. We call this the `diff`. Our diff isn't very interesting yet, but as we make changes your diff will become very colorful.

## Code Review in Pull Requests

To provide feedback on proposed changes, GitHub offers three levels of commenting:

## General Conversation

You can provide general comments on the Pull Request within the *Conversation* tab.

## Line Comments

In the files changed view, you can hover over a line to see a blue `+` icon. Clicking this icon will allow you to enter a comment on a specific line. These line level comments are a great way to give additional context on recommended changes. They will also be displayed in the conversation view.

## Review

When you are making line comments, you can also choose to *Start a Review*. When you create a review, you can group many line comments together with a general message: Comments, Approve, or Request Changes. Reviews have special power in GitHub when used in conjunction with protected branches.

# Activity: Code Review

One of the best ways to ensure code quality is to make peer reviews a part of every Pull Request. Let's review your partner's code now:

1. Click the *Pull Request* tab.
2. Use the *Author* drop down to locate your partner's pull request.
3. Click the *Files Changed* tab.
4. Hover over a single line in the file to see the blue +. Click the + to add a line comment.
5. Comment on the line and click *Start review*.
6. Repeat these steps to add 2-3 comments on the file.
7. Click *Review* in the top right corner.
8. Choose whether to *Approve* or *Request changes*
9. Enter a general comment for the review.
10. Click *Submit review*
11. Click the *Conversation* view to check out your completed review.

# Editing Files on GitHub

Since you created the pull request, you will be notified when someone adds a comment or a review. Sometimes, the reviewer will ask you to make a change to the file you just created. Let's see how GitHub makes this easy.

## Editing a File on GitHub

To edit a pull request file, you will need to access the *Files Changed* view.

1. Click the *pencil icon* in the top right corner of the diff to edit the file using the GitHub file editor.
2. Make changes to the file based on the comments from your reviewer or your personal perspective.

## Committing Changes on GitHub

Once you have made some changes to your file, you will need to create a new commit.

1. Scroll to the bottom of the page to find the *Commit changes* dialog box.
2. Type a *Commit message*.
3. Choose the option to *Commit directly to your branch*.
4. Click *Commit changes*.

## Activity: Editing Files in Pull Requests

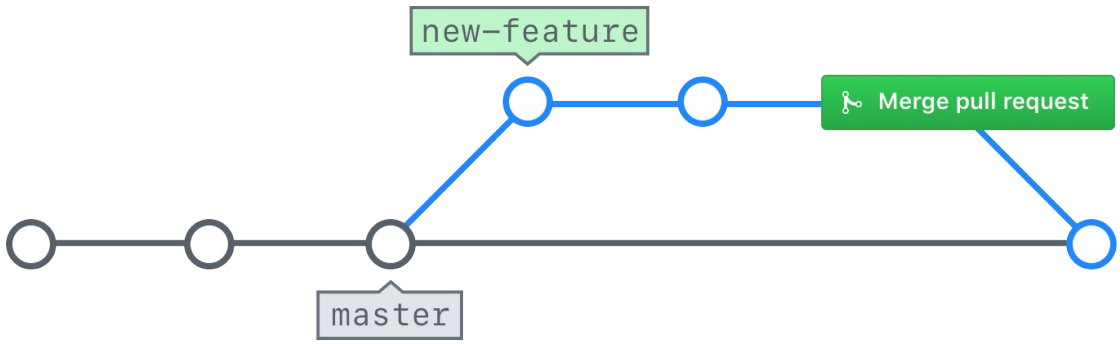
Go back to your Pull Request and make the edits requested by your collaborators.

# Merging Pull Requests

Now that you have made the requested changes, your pull request should be ready to merge.

## Merge Explained

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 `gh-pages` 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 to ensure at least one review has taken place.

This is a discussion you should have with the other members of your team.

## Merging Your Pull Request

Let's take a look at how you can merge the pull request.

1. Navigate to your Pull Request (HINT: Use the Author or Assignee drop downs to find your Pull Request quickly)
2. Click *Conversation*
3. Scroll to the bottom of the Pull Request and click the *Merge pull request* button

4. Click *Confirm merge*
5. Click *Delete branch*
6. Click *Issues* and confirm your original issue has been closed

> GitHub offers three different merge strategies for Pull Requests:
>
> - **Create a merge commit:** This is the traditional option that will perform a standard recursive merge. A new commit will be added that shows the point when the two branches were merged together.
> - **Squash and merge:** This option will take all of the commits on your branch and compress them into a single commit. The commit messages will be preserved in the extended commit message for the commit, but the individual commits will be lost.
> - **Rebase and merge:** This option will take all of the commits and replay them as if they just happened. This allows GitHub to perform a fast forward merge (and avoids the addition of the merge commit).

# Updating Your Local Repository

When you merged your Pull Request, you deleted the branch on GitHub, but this will not automatically update your local copy of the repository. Let's go back to our command line application and get everything in sync.

First, we need to get the changes we made on GitHub into our local copy of the repository:

1. Start by switching back to your default branch: `git checkout gh-pages`
2. Retrieve all of the changes from GitHub: `git pull`

`git pull` is a combination command that retrieves all of the changes from GitHub and then updates the branch you are currently on to include the changes from the remote. The two separate commands being run are `git fetch` and `git merge`

# Cleaning Up the Unneeded Branches

If you type `git branch --all` you will probably see that, even though you deleted your branch on the remote, it is still listed in your local copy of the repository, both as a local branch and as a read-only remote tracking branch. Let's get rid of those extra branches.

1. Take a look at your local branches: `git branch --all`
2. Let's see which branches are safe to delete: `git branch --merged`
3. Delete the local branch: `git branch -d <branch-name>`
4. Take another look at the list: `git branch --all`
5. Your local branch is gone but the remote tracking branch is still there. Delete the remote tracking branch: `git pull --prune`

> Adding the `--merged` option to the `git branch` command allows you to see which branches do not contain unique work when compared to the checked out branch. In this case, since we are checked out to gh-pages, we will use this command to ensure all of the changes on our feature branch have been merged to production before we delete the branch.

If you would like pruning of the remote tracking branches to be set as your default behavior when you pull, you can use the following configuration option: `git config --global fetch.prune true`.

# Viewing Local Project History

In this section, you will discover commands for viewing the history of your project.

## Using Git Log

When you clone a repository, you receive the history of all of the commits made in that repository. The log command allows us to view that history on our local machine.

Let's take a look at some of the option switches you can use to customize your view of the project history.

```
$ git log
$ git log --oneline
$ git log --oneline --graph
$ git log --oneline --graph --decorate
$ git log --oneline --graph --decorate --all
$ git log --stat
$ git log --patch
```

> Use the up and down arrows or press enter to view additional log entries. Type `q` to quit viewing the log and return to the command prompt.

# Streamlining Your Workflow with Aliases

So far we have learned quite a few commands. Some, like the log commands, can be long and tedious to type. In this section, you will learn how to create custom shortcuts for Git commands.

## Creating Custom Aliases

An alias allows you to type a shortened command to represent a long string on the command line.

For example, let's create an alias for the log command we learned earlier.

**Original Command**

```
$ git log --oneline --graph --decorate --all
```

**Creating the Alias**

```
$ git config --global alias.lol "log --oneline --graph --decorate --all"
```

**Using the Alias**

```
$ git lol
```

## Other Helpful Aliases

```
$ git config --global alias.co "checkout -b"
$ git config --global alias.s "status -s"
$ git config alias.dlb '!git checkout <DEFAULT-BRANCH> && git pull --prune &&
 git branch --merged | grep -v "\*" | xargs -n 1 git branch -d'
```

## Explore

Check out this resource for a list of common aliases:

- *git-scm.com/book/en/v2/Git-Basics-Git-Aliases* A helpful overview of some of the most common git aliases.

# Resolving Merge Conflicts

When you work with a team (and even sometimes when you are working alone) you will occasionally create merge conflicts. At first, merge conflicts can be intimidating, but resolving them is actually quite easy. In this section you will learn how!

## Local Merge Conflicts

Merge conflicts are a natural and minor side effect of distributed version control. They only happen under very specific circumstances.

- Changes to the same "hunk" of the same file
- Two different branches
- Changes on both branches happened since the branches have diverged

## Creating a Merge Conflict

Let's try to create a merge conflict, and fix it together. You and a partner will each create separate branches, create a file with the same name, and then try to merge. The first will merge cleanly, the second will have a merge conflict. Work together to resolve the merge conflict.

1. In our class repository, create the branch that you will be working on and name it something memorable like `USERNAME-conflict`.
2. On your branch, create a new file. The file name must be the same file name that your partner uses. Make sure the content inside of the file is different, and that neither file is empty.
3. Create a pull request in the class repository with `base: master` and `compare: USERNAME-conflict`.
4. You will see that the *first* pull request can merge well.
5. When you see the merge conflict in the *second* pull request, work together to resolve the merge conflict.
    i. Working locally, merge `master` into the feature branch.
    ii. When you see there's a conflict, that's OK! The files that have conflicts are listed under `Unmerged Paths`. Type `git status` to verify which file has the

conflict.

iii. Open that file in your text editor, and look for the merge conflict markers. ( `<<<<<<<` , `=======` , `>>>>>>>` )

iv. Both branches' versions of code are present - pick which one you want to keep, and save the changes.

v. Add and commit the saved changes to resolve the merge conflict.

vi. Push the feature branch up to the remote, and see the resolution in the pull request.

6. Merge the pull request.

> What is a merge message? In this example, we are doing a recursive merge. A recursive merge creates a new commit that permanently records the point in time when these two branches were merged together. We will talk more about Git's merge strategies a little later.

# Project: GitHub Games

In this section, we will work on a project repository called `github-games`.

A `github-games` repository has been created for you in the githubschool organization. You can access the repository at `https://github.com/githubschool/github-games-username`.

# What is CI/CD?

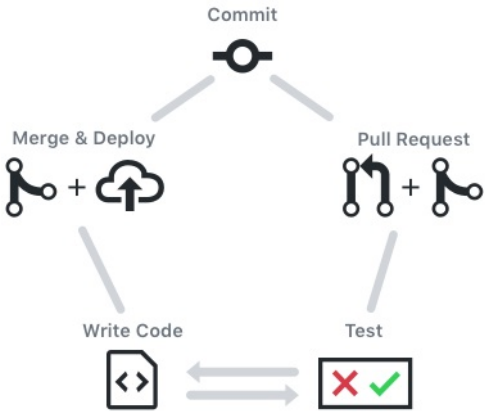
Continuous Integration and Continuous Deployment tools take manual tasks and do them automatically.

## Why run tests? Why use CI?

- Test code automatically
- Add structure to development process
- Test driven development (TDD) isn't just the addition of random tests, but rather about building tests as software requirements
- Builds are done on separate servers by the CI services
- Other benefits include consistent configurations, battery of tests, reduces "works on my machine"

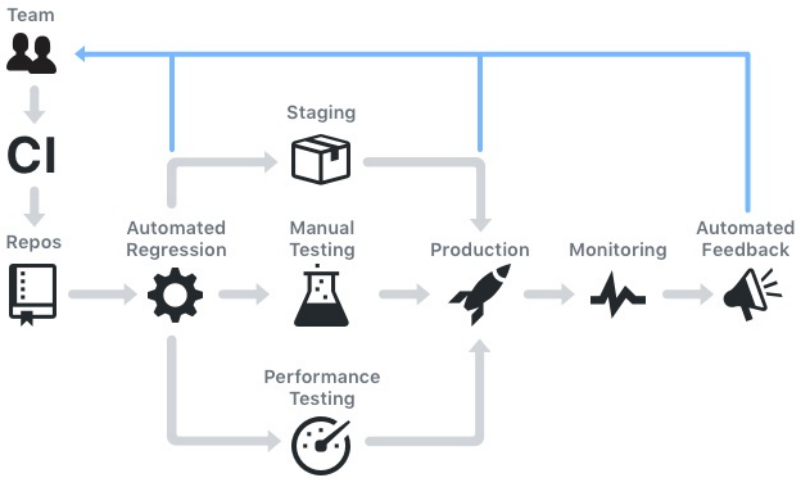
## How does CI/CD work with GitHub?

- When used with GitHub, tests can be run automatically on branches and pull requests every time there is a new commit, and return a status through GitHub's API
- Tests are run in consistent environment based on your software's production environment
- Protected Branches can serve as a gate, keeping pull requests without passing tests from being merged



# What is the difference between CI and CD?

- **CI**: Continuous integration
  - Continuous Integration is the practice of automatically kicking off tests with each push, rather than ad-hoc testing.
- **CD**: Continuous deployment
  - Code pushed to deployment automatically based on custom circumstances
  - Continuous Deployment is the practice of continuously deploying to production servers. In conjunction with CI, companies can move to a CD model by giving developers the freedom to deploy several times a day upon successful builds. Merging in and of itself isn't considered CD, unless it's tied to some deployment strategy (ex: Heroku automatically deploy anything that gets merged into master).
- We will cover more CI than CD in this course.
  - Though the concepts are different, CI and CD are frequently discussed and implemented together.
  - Typically CI and CD are implemented at the same time because many of the tasks are already defined. For example, a project may already have the tests written, and the steps for deployment. Once a CI/CD tool is introduced, it's simple to have them be done together.




# Other Things to Think About

- Protected branches
  - On GitHub, you may want to set up protected branches when you set up CI/CD.
  - Protected branches block code from being merged in to the master branch on the remote before it passes a set of configurable requirements, like passing CI/CD tests and builds, or having approved reviews.
- Automatic deploys
  - With CI/CD, you can also set up deploys to happen automatically when code passes the tests on the master branch.
  - Much like many integrations work with GitHub, many deployment options work with CI/CD services.
  - One example is Heroku. You could configure a project on Heroku with a CI/CD service, and whenever a build passes on the master branch, deployment to Heroku would be streamlined and taken care of automatically.

# Setting Up CI/CD

We are going to start with tests built in to `github-games`.

1. Visit the CI/CD integration site.
2. When prompted to create a new account or log in, authenticate with GitHub.
3. Add a new project, selecting the `github-games` repository. (Be careful to only select the appropriate repositories, not *all* repositories.)

You may not see anything happen just yet. CI/CD services look for a special `.yml` file with configuration information. This file will contain information about the language of the project, dependencies, what tests to run, and how to build any deployments.

# Merging Tests and the `.yml` file

Before we can begin using our Continuous Integration solution, we need to add the `.yml` file and some tests that we already created for our project.

1. Depending on which CI/CD service you are using, create a pull request into `base: master` from the appropriate branch, named `<service>-tests`.
2. **DO NOT** Merge the pull request.

> If you take a peek in the `.yml` file now, you'll see some important information. We'll go into this in more detail later. Just by looking at it, what do you think it all means? What questions do you have?

# Interacting with CI/CD

## Workflow Review: Updating the README.md to Pass the Tests

Now you will practice the GitHub Flow from beginning to end by updating the link in the README to point to your fork of the repository.

> Remember, your copy of the website will be rendered at `https://githubschool.github.io/github-games-username`.
>
> If you access this URL, you will see the text in the `README.md`. We have intentionally broken this repository so we can fix it together. The tests that we are merging also check for this URL.

Before we begin practicing the GitHub Flow however, you need to enable GitHub Pages in your repository and grab the URL for your GitHub Pages site.

1. Access your repository settings by clicking the *Settings* tab.
2. While on the *Options* pane (which is opened by default) scroll down to the *GitHub Pages* section.
3. Click the *None* drop-down under *Source* and select `master branch`.
4. Click the *Save* button.
5. Scroll back down to the GitHub Pages section, a new text box displays the URL for your published site. This is the URL you will be using to modify your `README.md` file later.

The current test is looking to see if the URL in the `README.md` is correct. Using the GitHub flow, create a pull request that will have a failing build and **doesn't** have the correct URL. Our current test is found here: `tests/test_verifyurl.rb`

The purpose of this section is to practice the workflow, looking at the test, and deciphering the error messages in a failing build. It can be overwhelming to look at all of the tool's feedback! If you'd like to dig in deeper, please do so, but if it feels like too much, focus on looking for the error message so you know what to fix.

1. Clone your copy of the repository: `git clone`

`https://github.com/githubschool/github-games-USERNAME.git`.

2. Checkout to the branch that is in the pull request, `git checkout CISERVICE-tests`.
3. Edit the URL in the `README.md` so the tests will *fail*.
4. Commit the changes to your branch.
5. Push your branch to GitHub: `git push`
6. See that the tests are failing.
7. Working locally again, change the URL to the URL that was created when you enabled GitHub Pages in your repository so the tests will *pass*. Commit those changes.
8. Push your branch up to the remote.
9. Let the tests run again. If they aren't passing, look at the test to see what changes you still need to make.
10. When all tests are passing, merge your Pull Request.
11. Delete the branch on GitHub.
12. Update your local copy of the repository: `git pull --prune`

> If desired, you can check this test locally by running `ruby tests/test_verifyurl.rb` on this branch. Notice how different it will look on everyone's machines, and how having the tests run externally will smooth out the process. *(For this to work, you will need Ruby installed locally.)*

# Protected Branches & `CODEOWNERS`

In some workflows, you will want to protect critical branches to ensure the code being merged to those branches has passed the required checks and received appropriate peer review. There are several methods for this, including **Protected Branches** and **Code Owners**.

## Protected Branches

Repository maintainers can prevent merges to specific branches that have not met pre-defined criteria. This criteria can include peer reviews, tests run by integrations such as a Continuous Integration services or code quality, or until a specific code owner has reviewed and approved changes.

Let's enable protected branches:

1. Select the **Settings** tab.
2. Select **Branches** from the menu on the left side of the screen.
3. Click the **Choose a branch...** dropdown and select the branch you would like to protect, for example, `master`.
4. Check the **Protect this branch** option.

Without checking any other options, basic branch protection prevents force-pushes and prevents it from being deleted. To learn more about the options available, check out the documentation for this feature.

## CODEOWNERS

Repository maintainers can define exactly which people and teams need to review sets of changes by creating a **CODEOWNERS** file. For example, you could use CODEOWNERS to ensure:

- your team's Javascript expert reviews all files with a `.js` extension
- your technical documentation team reviews all changes in the `docs/` folder
- your security team reviews any new dependencies listed in the `package.json` file

Let's create a CODEOWNERS file:

1. Create a new file in your repository titled `CODEOWNERS` (no extension necessary). You can add this to a `.github/` directory if desired.
2. On the first line, type `* @YOUR_USERNAME`
    - This means that you will be the default owner for everything in the repo, unless a later match takes preference.
3. On the next line, type `*.js @githubteacher`
    - Order is important. The last matching pattern for a given change takes precedence.
4. Now that you have created a CODEOWNERS file, go back to your branch protection settings and select the option to **Require pull request reviews before merging** and **Require review from Code Owners**. Remember to click **Save changes**.

For more information on how to format the CODEOWNERS file, check out the documentation

# Searching for Events in Your Code

In this section, we will learn how we can use `git bisect` to find the commit that introduced a bug into our repository.

## What is `git bisect` ?

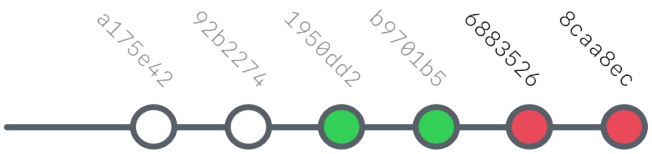
Using a binary search, `git bisect` can help us detect specific events in our code. For example, you could use bisect to locate the commit where:

- a bug was introduced.
- a new feature was added.
- a benchmark's performance improved.

## How it works

`git bisect` works by cutting the history between two points in half and then checking you out to that commit. You then check whether the bug/feature exists at that point and tell Git the result. From there, Git will do another division, etc until you have located the desired commit.





> When you are doing a bisect, you are essentially in a detached head state. It is important to remember to end the bisect with `git bisect reset` before attempting to perform other operations with Git. {: .warning}

# Finding the Bug in Our Project

# The Long Way

1. Initiate the binary search: `git bisect start` .
2. Specify the commit where you noticed the code was broken: `git bisect bad <SHA>` .
3. Specify the commit where you knew things were working: `git bisect good <SHA` .
4. Bisect will check you out to the midpoint between good and bad.
5. Run a test to see if the game would work at this point. Our test is to use `ls` to see if an `index.html` file exists.
6. If the game is still broken (there *is no* `index.html` file), type: `git bisect bad` .
7. If the game works (and there *is* an `index.html` file), type: `git bisect good` .
8. Git will bisect again and wait for you to test. This will happen until Git has enough information to pinpoint the first bad commit.
9. When Git has detected the error, it will provide a message that `SHA is the first bad commit.`
10. Exit the bisect process: `git bisect reset` .

# The Short Way

Bisect can also run the tests on your code automatically. Let's try it again using a shortcut command and a test:

1. `git bisect start <bad-SHA> <good-SHA>`
2. `git bisect run ls index.html`

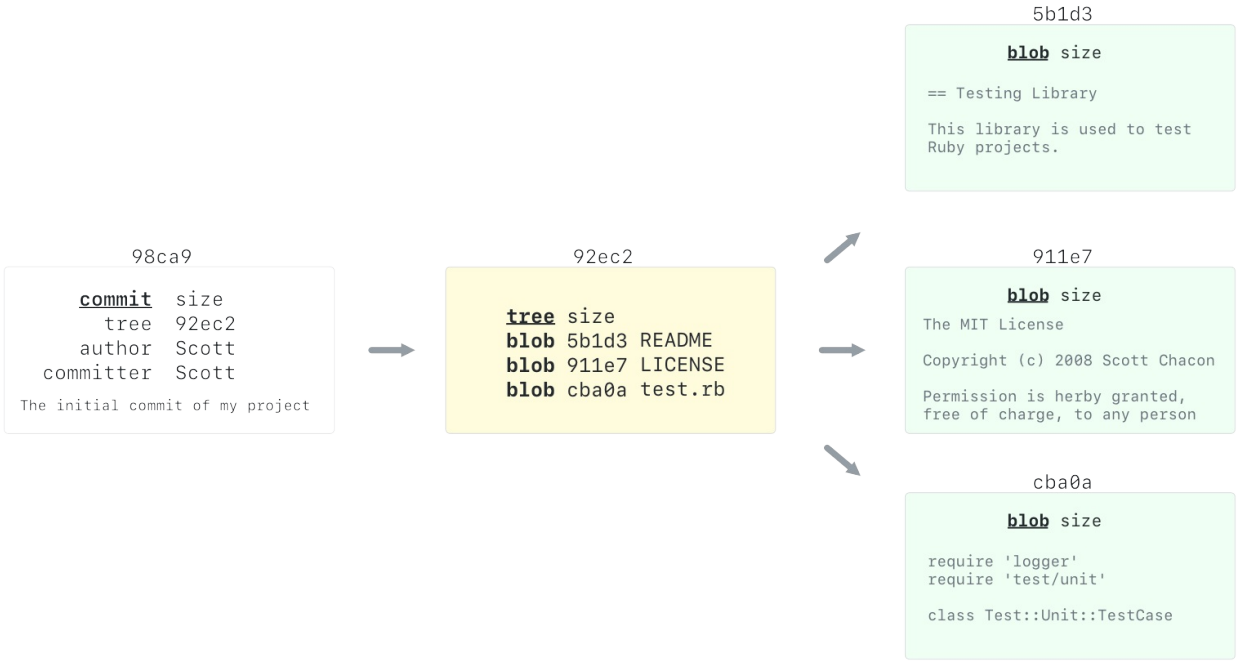
# Reverting Commits

In this section, we will learn about commands that re-write history and understand when you should or shouldn't use them.

## How Commits Are Made

Every commit in Git is a unique snapshot of the project at that point in time. It contains the following information:

- Pointers to the current objects in the repository
- Commit author and email (from your config settings)
- Commit date and time
- Commit message




Each commit also contains the commit ID of its parent commit.

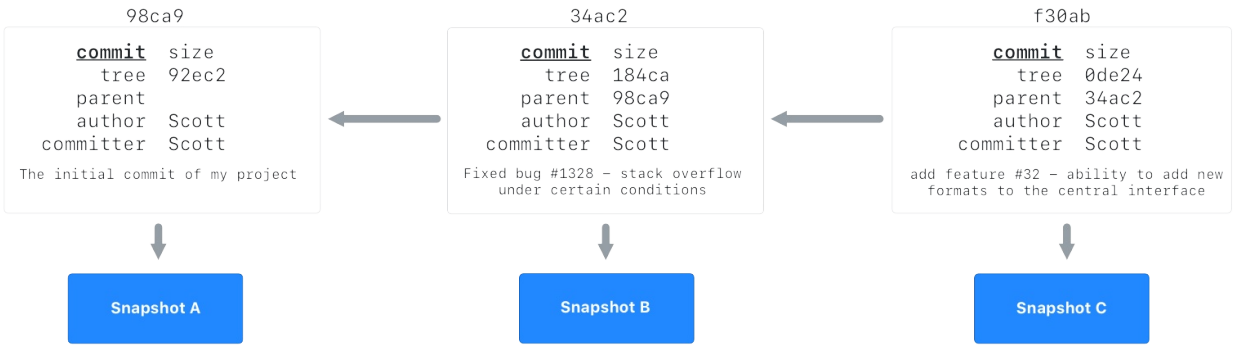



Image source: ProGit v2 by Scott Chacon

## Safe Operations

Git's data structure gives it integrity but its distributed nature also requires us to be aware of how certain operations will impact the the commits that have already been shared.

If an operation will change a commit ID that has been pushed to the remote (also known as a public commit), we must be careful in choosing the operations to perform.

## Guidelines for Common Commands

| Command | Cautions |
| --- | --- |
| `revert` | Generally safe since it creates a new commit. |
| `commit --amend` | Only use on local commits. |
| `reset` | Only use on local commits. |
| `cherry-pick` | Only use on local commits. |
| `rebase` | Only use on local commits. |

## Reverting Commits

To get your game working, you will need to reverse the commit that incorrectly renames `index.html`.

> **Warning**: Before you reverse the commit, it is a good idea to make sure you will not be inadvertently reversing other changes that were lumped into the same commit. To see what was changed in the commit, use `git show SHA`.

1. Initialize the revert: `git revert <SHA>`
2. Type a commit message.
3. Push your changes to GitHub.

# Helpful Git Commands

In this section, we will explore some helpful Git commands.

## Moving and Renaming Files with Git

1. Create a new branch named `slow-down`.
2. On *line 9* of the index.html file, change the background url to *(images/texture.jpg)*.
3. On *line 78*, change the timing for the game to speed it up or slow it down.
4. Save your changes.
5. See what git is tracking: `git status`
6. Create a new, empty directory: `mkdir images`
7. Move the texture file into the directory with git: `git mv texture.jpg images/texture.jpg`

## Staging Hunks of Changes

Crafting atomic commits is an important part of creating a readable and informative history of the project.

1. See what git is tracking: `git status`.
2. Move some parts of some files to the staging area with the `--patch` flag: `git add -p`.
3. Stage the hunk related to the image move: `y`
4. Leave the hunk related to the speed change in the working area: `n`

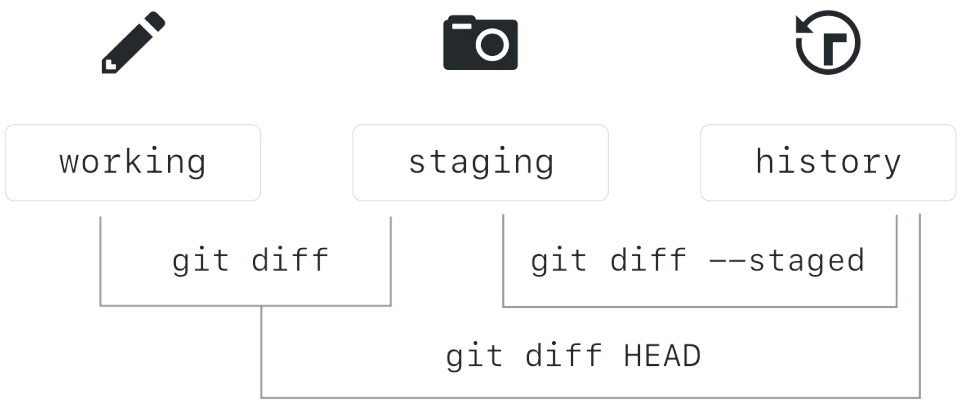
> Wondering what all of those other options are for the hunks? Use the `?` to see a list of options above the hunk.

# Viewing Local Changes

Now that you have some files in the staging area and the working directory, let's explore how you can compare different points in your repository.

## Comparing Changes within the Repository

`git diff` allows you to see allows you to see the difference between any two refs in the repository. The diagram below shows how you can compare the content of your working area, staging, and HEAD (or the most recent commit):




Let's try these commands on the repository:

```
$ git diff
$ git diff --staged
$ git diff HEAD
$ git diff --color-words
```

`git diff` will also allow you to compare between branches, commits, and tags by simply typing:

```
$ git diff <REF-1> <REF-2>
$ git diff gh-pages slow-down
$ git diff origin/gh-pages gh-pages
$ git diff 2710 b745
```

> Notice that, just like merges, diffs are directional. It is easiest to think of it as "diff back to starting at " or "see what is *not* in but *is* in ".

# Tags and Releases

You may want to put tags or releases on certain commits in your code's history to mark specific states or places in time. To do this, you could use Git's **tag** feature, or you could use GitHub's **release** feature.

## Tags

A tag is a pointer that points to a specific commit. Unlike commits, tags are *not* immutable. They can be moved and changed. Let's practice a bit with tags.

Tags can be created locally with Git, or on GitHub. When creating a tag from the command line, it's recommended to create an "annotated" tag. The following example creates an annotated tag with the `-a` flag, names the tag `v1.0`, and connects it to whichever commit SHA is included.

- `git tag -a v1.0 <SHA>`

To see all tags, type `git tag --list`.

Another caveat with tags is that they are not automatically pushed up with commits. To push tags, type `git push --tags`.

You can also set this as a default with configs using `git config push.followTags true` which will automatically push tags when their associated commits are pushed. Read more about this config setting..

## Releases

Releases are a GitHub feature that allow you to add an executable to the tag for easier access by visitors who just want to download and install your software. Releases are tags, because they point to a specific commit and can be named like any other tag. However, releases can also include attached binaries.

## Add a Release to GitHub-Games

1. On GitHub, navigate to the **Code** tab of the repository.

2. Under your repository name, click Releases.
3. Click Draft a new release.
4. Type a name for the tag. We recommend you use semantic versioning.
5. Select a branch that contains the project you want to release. Usually, you'll want to release against your master branch, unless you're releasing beta software. You can also select a recent commit by choosing the recent commits tab.
6. Type a title and description that describes your release.
7. If you're ready to publicize your release, click Publish release. Otherwise, click Save draft to work on it later.

> Notice that you could drag and drop or select files manually in the binaries box, or select "This is a pre-release" to notify users that it's not ready for production.

# Workflow Discussion

Now is a good time to discuss workflows - what works for you and your team, what might work, and what you've been doing in the past. Have a conversation either synchronously or in issues in the class repository about different workflows. If you need or want some inspiration for questions, take a look in the appendix of this manual.

# Changing the `.yml` file

Earlier in the course, we merged a `.yml` file into our `gh-pages` branch but we didn't really identify what the `.yml` file does. Continuous Integration providers use the `.yml` file to identify how you want your testing environment setup and what tests you want to run.

When we merged the existing `.yml` file into our project, it identified how the environment should be setup and included one test to ensure the README and link matched your username.

## Introduce Changes to the `.yml` File

We've created more tests on the in the `/tests` directory. Let's go through the process of adding a new test, and seeing what it's like to break it or test it.

1. If you look in the `tests/` directory, you'll see a few new tests. We're going to add the `tests/test_speedmax.rb` test to the `.yml` file.
2. Create a new branch based on `gh-pages`.
3. On that branch, open the `.yml` file. Add `tests/test_speedmax.rb` to the file in the same place that the other test, checking the URL, is being run.
4. Create a pull request.
5. Review the contents of the new test. What do you think it does? Why would we implement it? What could we do differently? When will it pass or fail?

*Note: If someone commits changes to the `.yml` file, those changes will be included in the tests. Since you cannot lock a file, it's important to remember to check any changes to the `.yml` file in pull requests.*

## Break the Speed Test

1. Based on what you know about that test file, make a change that you think will cause the test to fail. (Change line 78 `min:` to be anything outside of `0.0` and `1.0`.)

## Enable Protected Branches

1. See that you *could* merge the pull request even though the tests are failing.
2. In your repository, click on the `Settings` tab.
3. Select `Branches` on the left navigation panel.
4. Under `Protected Branches`, click `Choose a branch` and select `gh-pages`.
5. Select `Protect this branch` and then select whichever protections you'd like for your default branch.

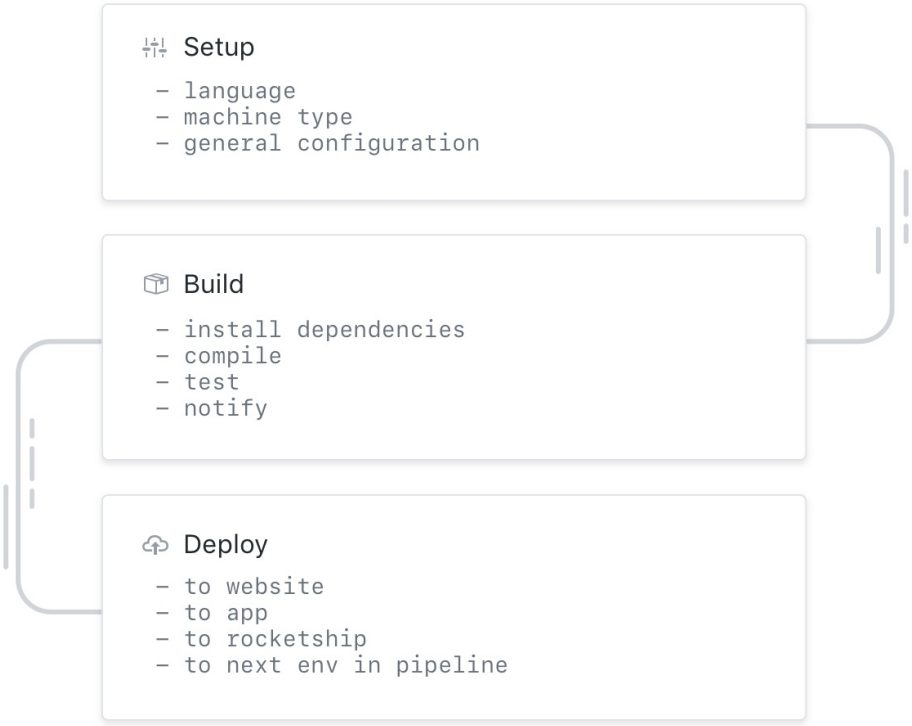
# Fix the Speed Test

1. After the status check returns from the CI/CD service, fix the file so the tests pass.
2. Merge the pull request.

# Understanding the `.yml` file

In the last section we had an opportunity to make a change to the `.yml` file, but what exactly is it and how does it help us?

In order to test your project, the CI service provider needs to know which tests to run on your code as well as the specifications for the build environment it should be tested within. Some service providers can automatically infer these settings based on your code, others require you to provide a detailed specification. These specifications are stored in the `<service-provider>.yml` file and are under version control just like the rest of your project.




The `.yml` is like a to-do list with three main parts:

- setting up the environment
- testing

- deploying

There are many other steps that can go around and in between as a part of these main sections. If there are some scripts you want to occur prior to the build, as part of the build, or based on the output or results of a build or test(s), that is all possible. The content of the `.yml` file will vary between service providers and you should consult their documentation when crafting your `.yml` file.

> Note: Our examples are relatively simple, but the capabilities of the testing environment go much deeper. The purpose of these testing environments is to test as closely to production as possible. The sky is the limit, if you want to test multiple versions, languages, environments, or machines, it is all possible! As we stated before, each CI vendor treats their .yml file differently, and you should consult their documentation when generating a `.yml` file that tests all the things.

# CircleCI

## CircleCI Documentation

Documentation for CircleCI is available at circleci.com.

## Support

CircleCI's community forums are available to all users, and premium support is available for teams who want more.

# Partner activity: Create a failing PR

You and a partner will work together through this activity. To help with this, we'll separate the instructions by indicating `Partner A` and `Partner B`. Get together with your partner and decide who'll take on each role.

## `PARTNER A` Create a broken build

- Add a reference to your partner's repository using:
  - `git remote add <name-of-remote> <url>`
    - Example: `git remote add partner https://github.com/githubteacher/github-games`
- Go to your partner's project and examine their `.yml` file. What is some code that would cause a pull request to fail based on their tests?
- Follow the GitHub flow to open a pull request introducing commits that will cause the tests to fail on your partner's repository.

## `PARTNER B` Respond to a failing build

- Find the pull request opened by your partner in your own repository. It should have a failing build status.
- Use failing build's output to figure out why the build is breaking.
- Review the pull request formally, requesting changes, with recommendations on what to change so that the build will pass.

## `PARTNER A` Fix the broken build

- After your partner has reviewed your pull request, make the changes they suggest (whether or not you think they are correct).

## `PARTNER B` Merge the successful build

- When the build is passing, merge the pull request and celebrate! 🎉

# Initializing a New Local Repository

Let's create a local repository that we can use to practice the next set of commands.

1. Navigate to the directory where you will place your practice repo ( `cd ..` to get back to the parent folder).
2. Create a new directory and initialize it as a git repository: `git init practice-repo`
3. CD into your new repository: `cd practice-repo`
4. Create an empty new file named `README.md` : `touch README.md`
5. Add and commit the README.md file.

Since we will be using this as our practice repository, we need to generate some files and commits. Here are some scripts to make this easier:

# Bash:

```
for d in {1..6}; do touch "file${d}.md"; git add "file${d}.md"; git commit -m
 "adding file ${d}"; done
```

# PowerShell:

```
for ($d=1; $d -le 6; $d++) { touch file$d.md; git add file$d.md; git commit -
m "adding file$d.md"; }
```

# Fixing Commit Mistakes

In this activity, we will begin to explore some of the ways Git and GitHub can help us shape our project history.

## Revising Your Last Commit

`git commit --amend` allows us to make changes to the commit that HEAD is currently pointing to. Two of the most common uses are:

- Re-writing commit messages
- Adding files to the commit

Let's see this in action:

1. Create a new file: `touch file7.txt`
2. When you are adding files to the previous commit, they should be in the staging area. Move your file to the staging area: `git add file7.txt`
3. `git commit --amend`
4. The text editor will open, allowing you to edit your commit message.

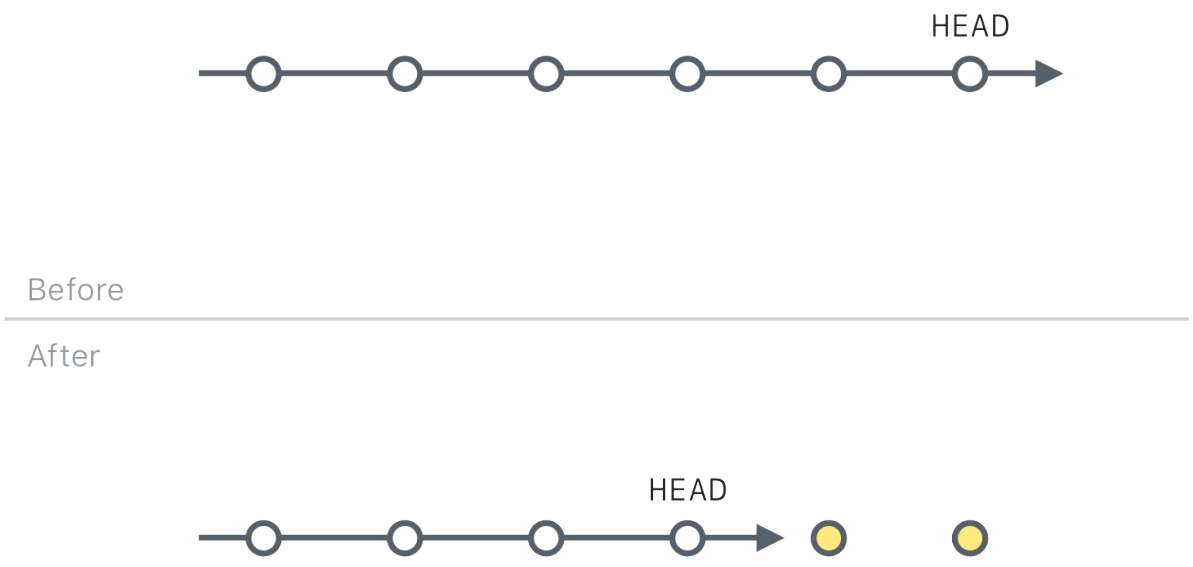
> You can actually amend any data stored by the last commit such as commit author, email, etc.

# Rewriting History with Git Reset

When you want to make changes to commits further back in history, you will need to use a more powerful command: `git reset`.

## Understanding Reset

Sometimes we are working on a branch and we decide things aren't going quite like we had planned. We want to reset some, or even all, of our files to look like what they were at a different point in history.



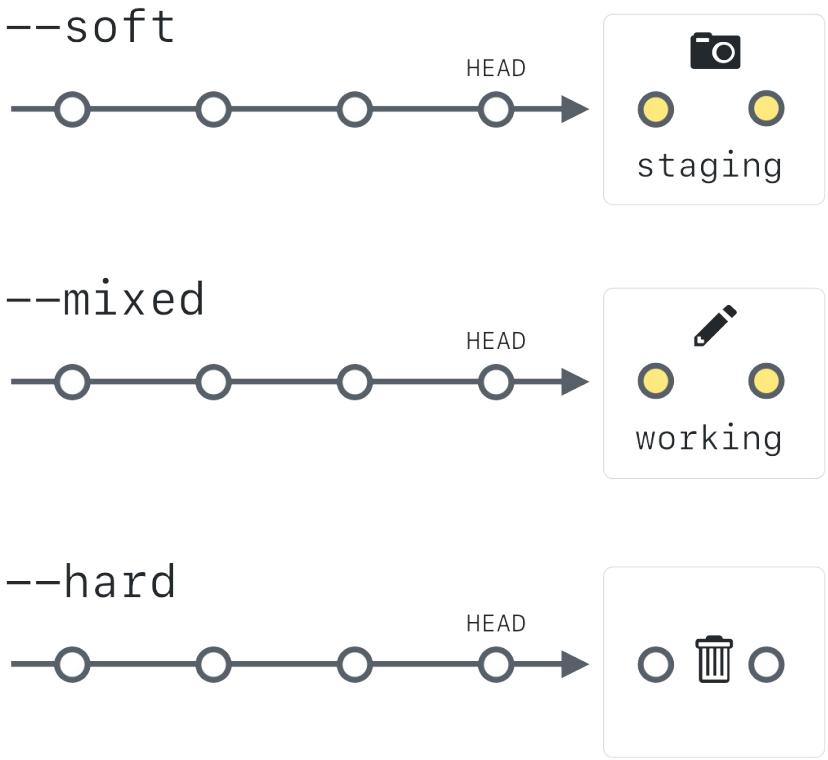

Remember, there are three different snapshots of our project at any given time. The first is the most recent commit (also known as HEAD). The second is the staging area (also called the index). The third is the working directory containing any new, deleted, or modified files.

The `git reset` command has three modes, and they allow us to change some or all of these three snapshots.

It also helps to know what branches technically are: each is a pointer, or reference, to the

latest commit in a line of work. As we add new commits, the currently checked-out branch "moves forward," so that it always points to the most recent commit.

## Reset Modes




The three modes for git reset are: `--soft` , `--mixed` , and `--hard` . For these examples, assume that we have a "clean" working directory, i.e. there are no uncommited changes.

### `--soft`

`git reset --soft <SHA>` moves the current branch to point at the `<SHA>` . However, the working directory and staging area remain untouched. Since the snapshot that current branch points to now differs from the index's snapshot, this command effectively stages all differences between those snapshots. This is a good command to use when you have made a large number of small commits and you would like to regroup them into a single commit.

### `--mixed`

`git reset --mixed <SHA>` makes the current branch *and* the staging area look like the `<SHA>` snapshot. *This is the default mode:* if you don't include a mode flag, Git will assume you want to do a `--mixed` reset. `--mixed` is useful if you want to keep all of your changes in the working directory, but change whether and how you commit those changes.

### `--hard`

`git reset --hard <SHA>` is the most drastic option. With this, Git will make all 3 snapshots, the current branch, the staging area, *and* your working directory, look like they did at `<other-commit>`. This can be dangerous! We've assumed so far that our working directory is clean. If it is not, and you have uncommitted changes, `git reset --hard` will *delete all of those changes*. Even with a clean working directory, use `--hard` only if you're sure you want to completely undo earlier changes.

## Reset Soft

Using the practice repository we created earlier, let's try a `reset --soft`.

1. View the history of our project: `git log --oneline --decorate`
2. Identify the current location of `HEAD`.
3. Go back two commits in history: `git reset --soft HEAD~2`
4. See the tip of our branch (and `HEAD`) is now sitting two commits earlier than it was before: `git log --oneline --decorate`
5. The changes we made in the last two commits should be in the staging area: `git status`
6. Re-commit these changes: `git commit -m "re-add file 5 and 6"`

> In this example, the tilde tells git we want to reset to two commits before the current location of `HEAD`. You can also use the first few characters of the commit ID to pinpoint the location where you would like to reset.

## Reset Mixed

Next we will try the default mode of reset, `reset --mixed`:

1. Once again, we will start by viewing the history of our project: `git log --oneline`
2. Go back one commit in history: `git reset HEAD~`
3. See where the tip of the branch is pointing: `git log --oneline --decorate`
4. The changes we made in the last commit have been moved back to the working directory: `git status`
5. Move the files to the staging area before we can commit them: `git add file5.md file6.md`
6. Re-commit the files: `git commit -m "re-add file 5 and 6"`

> Notice that although we have essentially made the exact same commit (adding file 5 and 6 together with the same HEAD and commit message) we still get a new commit ID. This can help us see why the reset command should never be used on commits that have been pushed to the remote.

## Reset Hard

Last but not least, let's try a hard reset.

1. Start by viewing the history of our project with: `git log --oneline`
2. Reset to the point in time where the only file that existed was the README.md: `git reset --hard <SHA>`
3. See that all of the commits are gone: `git log --oneline`
4. Notice your working directory is clean: `git status`
5. See that the only file in your repository is the README.md: `ls`

> **Warning:** Remember, `git reset --hard` overwrites your working directory, staging area, and history. This means that uncommitted changes you have made to your files will be completely lost. Don't use it unless you really want to discard your changes.

## Does Gone Really Mean Gone?

The answer: It depends!

```
$ git reflog
```

The reflog is a record of every place HEAD has been. In a few minutes we will see how

the reflog can be helpful in allowing us to restore previously committed changes. But first, we need to be aware of some of the reflog's limitations:

- **The reflog is only local.** It is not pushed to the remote and only includes your local history. In other words, you can't see the reflog for someone else's commits and they can't see yours.
- **The reflog is a limited time offer.** By default, reachable commits are displayed in the reflog for 90 days, but unreachable commits (meaning commits that are not attached to a branch) are only displayed for 30 days.

# Getting it Back: `git cherry-pick`

We just learned how reflog can help us find local changes that have been discarded. So what if:

## You Just Want That One Commit

Cherry picking allows you to pick up a commit from your reflog or another branch of your project and move it to your current branch. Right now, your file directory and log should look like this:

```
$ ls
README.md
$ git log --oneline
84nqdkq initializing repo with README
```

Let's cherry pick the commit where we added file 4:

1. Find the commit ID where you added file4.md: `git reflog`
2. Cherry-pick that commit: `git cherry-pick <SHA>`

Now when you view your directory and log, you should see:

```
$ ls
file4.md
README.md
$ git log --oneline
eanu482 adding file 4
84nqdkq initializing repo with README
```

Is the commit ID the same as the one you used in the cherry pick command? Why or why not?

> Remember, when using any commands that change history, it's important to make these changes before pushing to GitHub. When you change a commit ID that has been pushed to the remote, you risk creating problems for your collaborators. {: .warning}

# Oops, I Didn't Mean to Reset

Sometimes, you `git reset --hard` a little further than intended and want to restore that work. The good news is, that `git reset --hard` doesn't just work by going back in time, it can also go forward:

1. View the history of everywhere HEAD has pointed: `git reflog`
2. Reset to the point in time where the original `file6.md` was created: `git reset --hard <SHA>`
3. See your restored history: `git log --oneline`

Take a look at the commit IDs in `git log --oneline` compared to `git reflog`. What do you notice?

> Why didn't this command cause a merge conflict since we had already cherry-picked file 4. The reason is that `git reset --hard` is not trying to merge the two histories together, it is simply moving the branch to point to a new commit. In this case, this was what we wanted. In other cases, this could cause us to loose any work we may have done after the original reset.

# Merge Strategies: Rebase

In this section, we will discuss another popular merge strategy, rebasing.

## Understanding Git Merge Strategies

Git uses three primary merge strategies:

## Fast forward

A fast forward merge assumes that no changes have been made on the base branch since the feature branch was created. This means that the branch pointer for base can simply be "fast forwarded" to point to the same commit as the feature branch.

## Recursive

A recursive merge means that changes have been made on both the base branch and the feature branch and git needs to recursively combine them. With a recursive merge, a new "merge commit" is made to mark the point in time when the two branches came together. This merge commit is special because it has more than one parent.

## Octopus

A merge of 3 or more branches is an octopus merge. This will also create a merge commit with multiple parents.

## About Git rebase

`git rebase` enables you to modify your commit history in a variety of ways. For example, you can use it to reorder commits, edit them, squash multiple commits into one, and much more.

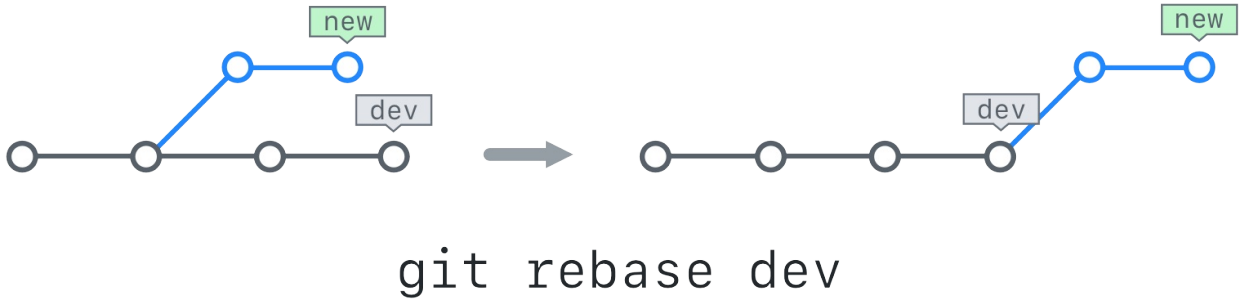
To enable all of this, `rebase` comes in several forms. For today's class, we'll be using interactive rebase: `git rebase --interactive` , or `git rebase -i` for short.

Typically, you would use `git rebase -i` to:

- Replay one branch on top of another branch
- Edit previous commit messages
- Combine multiple commits into one
- Delete or revert commits that are no longer necessary

## Creating a Linear History

One of the most common uses of rebase is to eliminate recursive merges and create a more linear history. In this activity, we will learn how it is done.




## Set Up

1. Find the SHA of the initial commit: `git log --oneline`
2. Reset to the SHA of the initial commit: `git reset --hard SHA`
3. Create a new branch and check out to it: `git checkout -b rebase-me`
4. Cherry-pick files 4-6 onto the `rebase-me` branch using the reflog.
5. Checkout to master: `git checkout master`
6. Cherry-pick files 1-3 onto the `rebase-me` branch using the reflog.
7. Look at your history: `git log --oneline --graph --decorate --all`
8. If you merged now, it would be a recursive merge.

## Begin the Rebase

1. Checkout to the `rebase-me` branch: `git checkout rebase-me`
2. Start the merge: `git rebase -i master`
3. Your text editor will open, allowing you to see the commits to be rebased.
4. Save and close the `rebase-todo` .
5. Watch your rebase happen on the command line.
6. Take another look at your history: `git log --oneline --graph --decorate --all`

7. If you merged now, it would be a fast-forward merge.

## Finish the Merge

1. Checkout to master, the branch you will merge into: `git checkout master`
2. Merge your changes in to master: `git merge rebase-me`

# Day 1 Activities

## Add a New Slide

Just like we did in class, open a new pull request adding a new slide to the deck.

1. Create a new branch, and checkout to that branch: `git checkout -b NEWBRANCHNAME`.
2. Create a new file in the `_posts` directory, named by date and description like YYYY-MM-DD-username-description.md. (The date has to be in the past - future dates won't show up.)
3. Follow the directions for the same activity we created together, where it says "Activity: Edit Your File".
4. Instead of just adding the caption, you will need to find a new image from the image list and use that to fill in this template:

    ```
    ---
    layout: slide
    ---

    IMAGEURL
    {: .center}

    CAPTION-HERE
    {: .fragment}
    ```

5. Save and commit your changes on your new branch.
6. If working locally, push your changes up to the remote: `git push -u origin NEWBRANCHNAME`.
7. Open a pull request In the body of the pull request, @ mention anyone you'd like to review your changes.
8. Once the tests pass on your pull request, merge the pull request.

## Add a Caption To an Existing Slide

Add a caption to an existing slide in someone else's pull request.

1. Find a pull request that you'd like to add a caption to.
2. See what image they have chosen by clicking 'files changed', and then 'view'.
3. Edit the file, either in the browser or locally, to add a line with your caption. (Please do not erase work that others have added to this file.)
    - If you are working locally, you will need to check out to the branch.
    - Make sure you have all of the remote changes updated in your local repository: `git pull`.
    - Look for the name of the branch in the pull request, and check out to that branch locally: `git checkout BRANCHNAME`.
4. Save the changes, and commit the file.
    - If you are working locally, push the changes up to the remote: `git push`.
5. Do not merge the pull request, simply `@` mention the user who opened the pull request to let them know about your changes.

## Improve the `README.md`

Improve our `README.md` by adding some of your favorite resources.

1. Create branch, and checkout to that branch: `git checkout -b NEWBRANCHNAME`.
2. Edit the README.md to be better in some way. This could mean adding a new resource, or making the existing descriptions more clear. Save and commit your changes.
3. If working locally, push your changes up to the remote: `git push -u origin NEWBRANCHNAME`.
4. Open a pull request In the body of the pull request, with `base: master` and `compare: NEWBRANCHNAME`.
5. `@` mention anyone you'd like to review your changes.
6. Once the tests pass on your pull request, merge the pull request.

## Restyle Slides

If you'd like a more advanced challenge, and you have an eye for style, change the colors, font, and other aspects of the class slide deck. **Note: If multiple participants attempt this, there may be merge conflicts.**

1. Create a new branch and check out to it: `git checkout -b NEWBRANCHNAME`.
2. Find the file `_sass/solarized/solarized.scss`.
3. Make changes in the file.
    - Lines 12-19 affect colors
    - Lines 33-35 affect font and font size
    - Lines 52-55 affect headers
4. Save and commit your changes on your branch.
5. Push your branch to the remote: `git push -u origin NEWBRANCHNAME`.
6. Open a pull request with `base: master` and `compare: NEWBRANCHNAME`.
7. `@` mention anyone you'd like to review your changes.
8. Once the tests pass on your pull request, merge the pull request.

# Day 2 Activities

## Merge Conflict Practice

Depending on how you're interacting with this manual, you may be in a class. The instructor may have set up a repository for you to practice merge conflicts. If this is the case:

- *Every person* has their own repository. Each person should fix the merge conflicts in their own repo. It will be called `github.com/githubschool/conflict-practice-username`, with username being your actual username.
- We won't make you turn in your homework, but we will run a script to see if the activities are completed later. 😉

## Work to resolve the merge conflicts in the conflicts repository.

1. Find your repository. It will be at `github.com/githubschool/USERNAME`, where your username is replacing the word USERNAME.
2. In your repository, navigate to the **Pull Requests** tab.
3. There are three open pull requests, and all of them have merge conflicts to fix. We recommend fixing them in this order:
    - Update README
    - Updates to game manual
    - Minor CSS fixes
4. View the pull request, and follow the steps to resolve the merge conflicts. When the merge conflict is resolved, merge the pull request.

# Discussion Guide: Team Workflows

Here are some topics you will want to discuss with your team as you establish your ideal process:

1. Which branching strategy will we use?
2. Which branch will serve as our "master" or deployed code?
3. Will we use naming conventions for our branches?
4. How will we use labels and assignees?
5. Will we use milestones?
6. Will we have required elements of Issues or Pull Requests (e.g. shipping checklists)?
7. How will we indicate sign-off on Pull Requests?
8. Who will merge pull requests?

# How to Generate Jekyll Sites Locally

## Before you Begin

The repositories we use in class are deployed using GitHub Pages, so you don't *need* to do any local serving. But, if you'd like to see what your changes look like locally before they actually are pushed to the default branch, you can.

The process may vary in difficulty based on your operating system, and we've found it's a smoother experience in macOS. We're working to improve these instructions for all platforms, so please let us know if you have any recommendations.

This script uses Ruby. If you don't already have Ruby installed locally, you should follow the detailed instructions on ruby-lang.org. In short:

- On Windows, you can do so from https://rubyinstaller.org, select version `2.3.3`, ensure you select to "Add Ruby executables to your PATH" during setup, and restart your machine.
- If you need to install Ruby on a Mac, install homebrew and then run `brew install ruby`.

## Generating Locally

1. Using a Bash-like terminal, `cd` to the class repository locally.
2. Check that Ruby is installed.
    - on your command line, run `ruby -v` and `gem -v`
    - If you see a version `2.3.x` for Ruby, you're 👍. If not, refer to directions above. ☝
3. If there is a `Gemfile.lock` file, delete it.
4. Install bundler: `gem install bundler`.
5. Type `script/setup` to install all of the proper gems.
6. Then, type `script/server`.
7. If all goes well, your terminal will tell you where to access your site in your browser. 🎉

# Add the Git branch to your terminal prompt

To show your active Git branch in your terminal prompt, you will need to do the following:

- If you are on a **Mac**, you can add the code shown below to your `.bash_profile` file.
- If you are on **Linux**, you will add the code shown below to your `.bashrc` file.
- If you are on **Windows**, you probably aren't reading this because Windows provides this behavior by default.

## The Script

```
parse_git_branch() {
  git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
export PS1="\w\[\033[36m\]\$(parse_git_branch) \[\033[00m\] > "
```

Or, another option:

```
function parse_git_branch () {
  git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}

RED="\[\033[0;31m\]"
YELLOW="\[\033[0;33m\]"
GREEN="\[\033[0;32m\]"
NO_COLOR="\[\033[0m\]"

PS1="$GREEN\u@\h$NO_COLOR:\w$YELLOW\$(parse_git_branch)$NO_COLOR\$ "
```

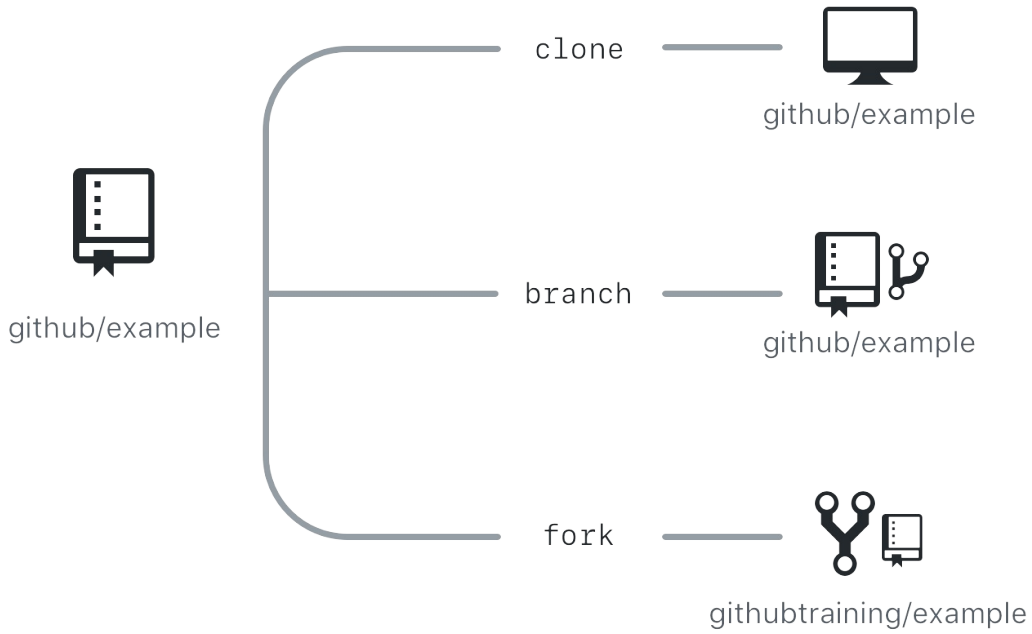
# Fork and Pull Workflow

Let's distinguish between a few vocabulary words in Git and discuss a common workflow that involves forking.

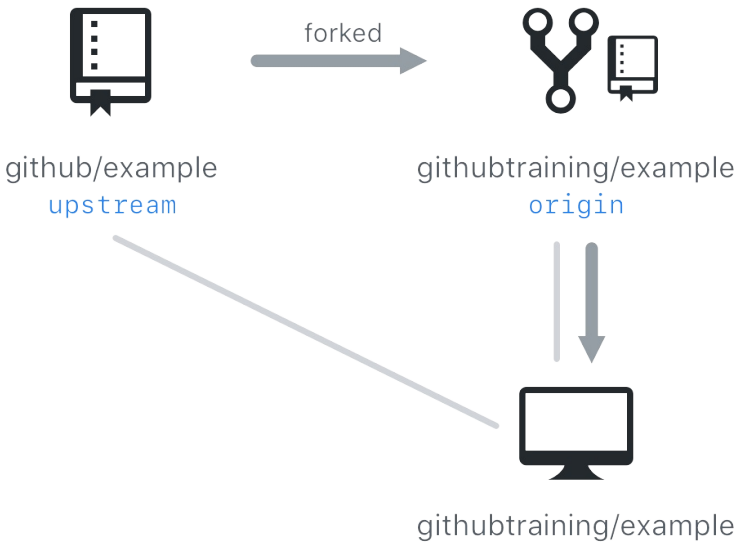
As in the picture below, all of our work generally starts from one parent repository. We call this the **parent** repository, but we can sometimes call it different things depending on *how* we work with it.

For example, when we **clone** a repository by creating a local copy on our machine, we refer to that original remote repository as **origin**.

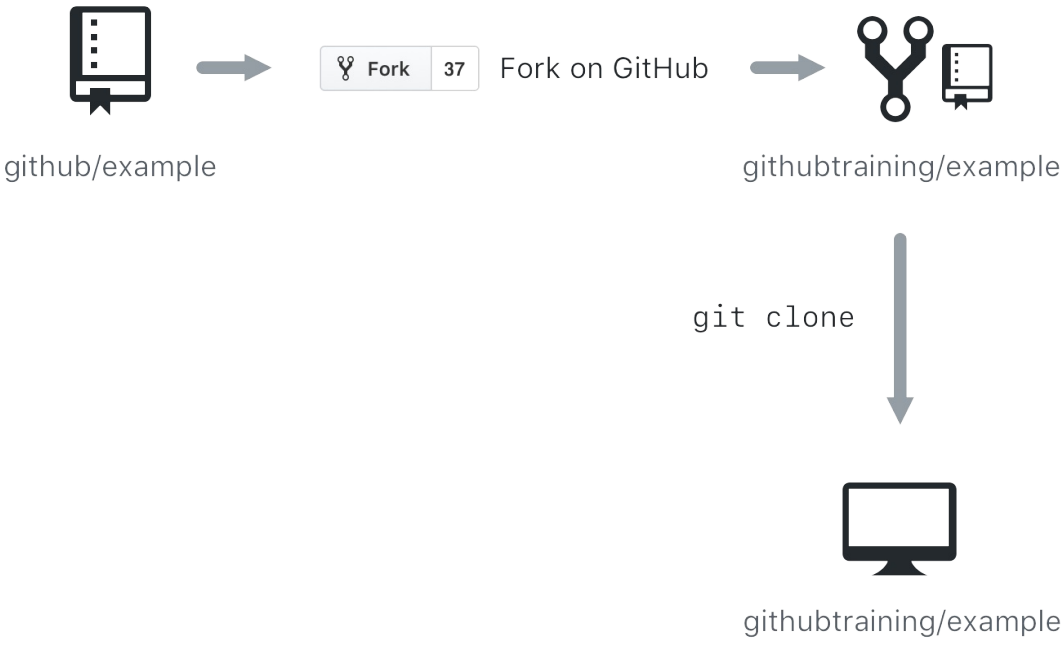
When we create branches, we can create them on our local **clone** *and* on the remote **origin**.



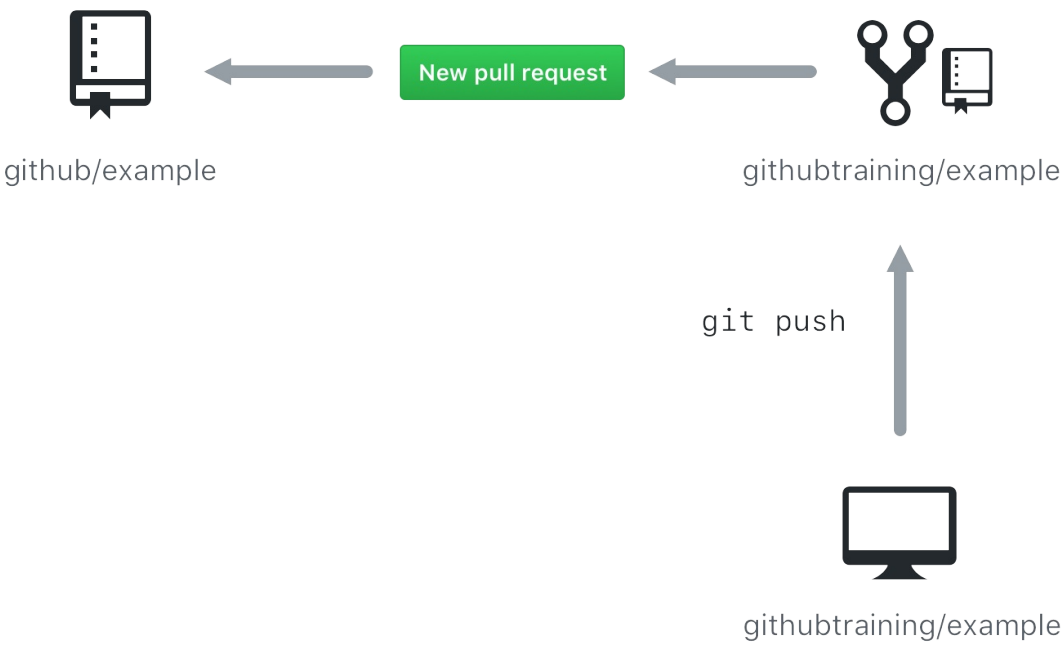

Things start to get tricky when we start talking about forking. A **fork** is a remote copy of a repository with a different owner. When you have a new remote **fork**, you would refer to the parent remote as **upstream**.




If you cloned the fork, you would refer to your own fork as **origin**, and the parent as **upstream**.




When you want to return work back to the **origin** or **upstream** repositories, you would push back to **origin**, then open a pull request between the remote forks.


New pull request
github/example
githubtraining/example
git push
githubtraining/example

# Cleaning Out Your Loaner PC

If you borrowed a laptop or used a virtual machine for this class, you will want to clear out your configs and credentials before you give it back. Here's how:

## Step 1: Clear your configs

1. `git config --unset --global user.name`
2. `git config --unset --global user.email`

## Step 2: Clear your command line history

- **Bash:** `history -c && history -w && exit`
- **Windows command prompt:** Alt+F7

## Step 3: Clear the credential manager

**For Windows**

1. From the **Start** menu, select **Control Panel**.
2. Click **User Accounts**.
3. Click **Manage my credentials**.
4. Click **Credential Manager**.
5. In the Windows Credentials and Generic Credentials section, remove any stored credentials referencing Git or GitHub.

**For Mac**

1. Open the **Keychain Access** app.
2. Delete anything with GitHub.