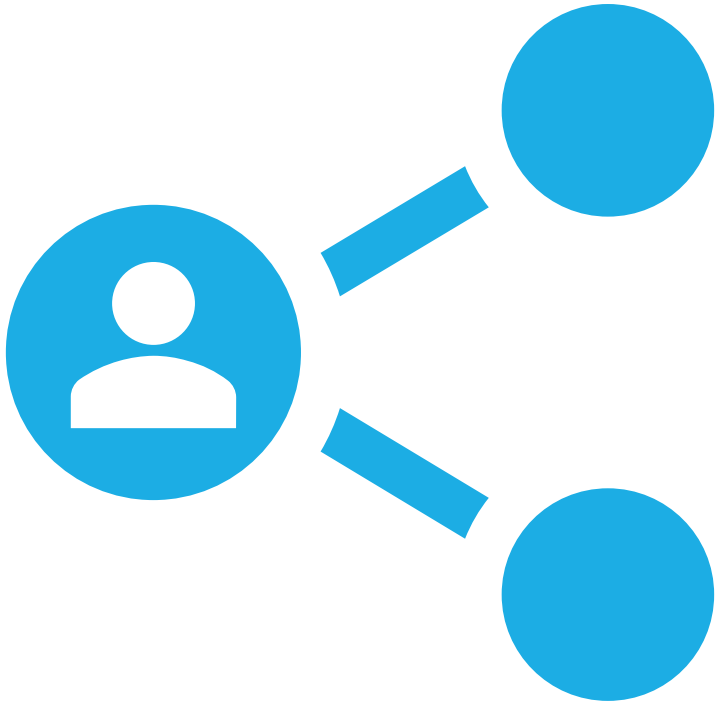
GIT

For when you inevitably break things



GIT

Git is a **source code management (SCM)** tool that we use as developers to track and maintain code bases. You might be asking "Why are we learning about a tool to track and maintain code when we haven't written any code?".

That's a good question that I have two answers for:

1. The longer you are exposed to this tool, the more you will understand it.
2. Learning the basics of Git will allow us to learn the basics of GitHub, which is a cloud based tool that will allow us to not only extend the functionality of Git, but also allow me to easily share code with all of you.

So with those (hopefully) satisfactory answers, let's get started.

GIT

Git is a free and open source project that dominates the **version control** market.

It was created by Linus Torvalds (a modern day computer science hero) because he needed a system to allow multiple people to work on the Linux kernel without paying for a commercial version control system.

Along with Git, there is GitHub. A service owned by Microsoft that allows developers to host their code in a cloud environment for free.

This means that anyone anywhere with an internet connection can stay up to date with a project that a team is working on, and contribute to it.

GIT

Before we get started with Git, I want to make something clear that many people get confused. **Git is not the same thing as GitHub**. People tend to use these terms interchangeably and this is wrong.

We will clear the difference up by the end of today, but please don't think of these things as one and the same.

In these slides we will only be covering the workflow that assumes you are:

- Working alone on a project
- Not using GitHub for cloud based backups and movement

This means that this slide deck wont go into topics like **branching** as you need to know your basics first!

THE PROBLEM

Before we get into Git, I want you to try and understand the following toy problem:

I have tasked you with writing an essay on your favorite topic. The only thing is I would like to see how over time your essay has progressed. Each time you feel like you have made progress, I want to be able to see that progress.

Chances are you would simply save the document and manually make a copy of it and store that copy somewhere so when it is time to submit, you can submit all the copies and I can see the changes you made over time.

The problem here is depending on how large this essay is I might end up having a large amount of copies when this really isn't needed and makes the process of moving these copies around much more annoying than it needs to be.

THE SOLUTION

What if instead of making copies each time, Word was smart enough to save your changes as you made them and allow you to go back when you wanted to?

Think about it like making a **checkpoint** in your progress which was saved into project itself. So when you made some progress you were happy with, you could create a checkpoint in the essay that would remember what the essay was like right at this point in time.

Now when you submit the document to me, I can simply see all the checkpoints that you made and go through them 1 by 1 to see how you made progress.

This also allows you to go back in time in case you ever make a change that you aren't happy with later!

THE GIT PROBLEM

Just like the essay problem, work you do on a code project continually overwrites any previous work! If you write 500 lines, save, come back the next day and change 200 of them, save, and come back the next day you have no way to go back to the original 500 lines. You overwrote them when you saved.

This is the problem you face when not using Git. If I write my code one day and then come and change some of it the next day, after I save I have no way of getting back to the first day's code. This is a massive problem if you break things on day two.

THE GIT SOLUTION

So what is the solution Git provides? Well it's much like the solution to the essay problem. Instead of simply saving the new code over and over again, Git stores **only the changes you make to your code.**

Now, just like the essay problem, you can ask Git to take you to any point of your project's history. This is great for us as developers because if we mess something up, as long as we followed the correct Git workflow, we can simply go back to the point in our project when everything was fine.

BASIC GIT WORKFLOW

Right now we don't need to get into the weeds and advanced features of Git. The simple stuff will let us build an understanding that as we progress through the course will build into a deeper understanding.

When working with just Git and not mixing in GitHub, the basic workflow is simple and can be split into two parts:

1. Initialization of the repository (only done once)
2. Add / Commit code changes (done repeatedly)

CONFIGURE GIT

Before you can take advantage of Git's power, you need to configure git on your system. This is pretty simple, we just need to give Git a username to attribute changes to when we make them.

Open a terminal and run the command:
git config --global user.name "*YourUsername*"

Replace YourUsername with one that you like! Later we will have to replace this with the username we actually get on GitHub. Once we have it set, we only ever have to run this when we want to change our configured username so almost never (unless we are changing our username or setting up a new computer).

INITIALIZING A LOCAL REPOSITORY

Now we can attach Git to our projects. In Git terms, we say **initialize a repository**.

Initializing a local Git repository is a two step process:

1. Navigate to the *root* directory of your project (the folder that contains all file and folders for the projects)
2. Run the command **git init**

You are now done, the repository has been initialized and you as you code you can take advantage of all the git features.

What really happens when you run **git init** in the terminal is a folder called ".git" is created. The git tool uses this folder to store information about your project.

STAGING & COMMITTING CODE CHANGES

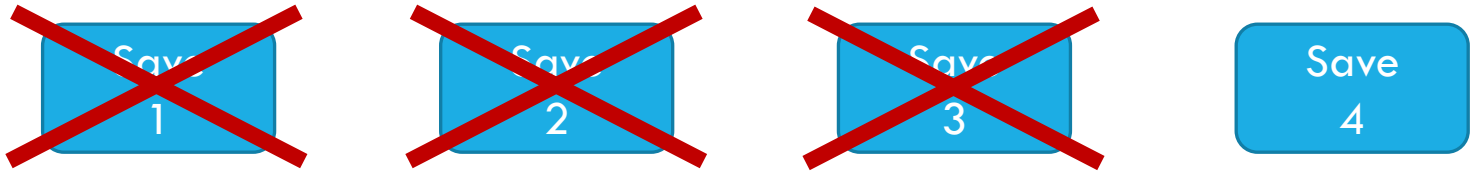
Now that your project is backed by Git, let's commit some code changes.

1. From inside the project run the command **git add –A**
   - This will add all of the changes you have made to the code to the **staging area.** In this stage, code can still be removed and not committed.
2. Run the command **git commit –m "Message that describes your changes"**
   - At this point, git has created a commit point in your project that you can navigate back to if needed. Think of it like a save point.

That's it! Git handles the rest for you. There is now a saved point that you can go back to at any moment if you break something.

VISUAL REPOSITORY

When working with git, we create “commit points” that we can navigate between. If we make a mistake in commit 4, we can always go back to commit 3. This is very different when compared to the way you usually do work on the computer which follows this model:



KNOWLEDGE CHECK

1. Create a folder inside your warmup folder called GitCheck and open this folder with vscode (using the command line)
2. Initialize this folder as a git repository
3. Create a file called index.html
4. Add and commit your changes with a good message (up to you what that is)
5. Open the index.html file, add in the basic starting point for a site and save
6. Add and commit your changes with a good message

BASIC GIT COMMANDS

Git has many commands that you should know about to make interacting with your repositories a breeze.

1. **git init**
   - This command will initialize a project as a git repository for you. This only needs to be run once per project and needs to be run from the **root** of the project.
2. **git status**
   - This will tell you some information about your current status in the repository. Is there code that has been modified but not committed or added to the staging area? It will also tell you what "branch" you are on, which is a topic for another time.
3. **git log**
   - This will show you all of the information of the previous commits in the repository. Things like the author, date of commit, and the commit message
4. **git add**
   - This command adds files to the staging area. You can either pass names of files directly, or add the –A option to add all changes to the staging area.
5. **git commit**
   - This command will take all the changes in the staging area and create a commit point. This commit points can be navigated between if you ever need to go back to a point in the repository.
6. **git checkout <id>**
   - This command will take you between commits in the repository. Remember you can only do this if all the current changes in your code are committed! This is also known as a **clean state**.

KNOWLEDGE CHECK

1. Using the GitCheck folder from before, check the status of your repository
2. Open index.html and make some changes. Save the document.
3. Check the status of your repository
    1. This is just to show info, it doesn't *do* anything
4. Add and commit your changes with a good message (up to you what that is)
5. Check the status of you repository
6. Check the logs of your repository
    1. This is just to show info, it doesn't *do* anything

Git is a **massively** important topic. It really only takes a couple projects to get used to it so make sure you are comfortable with it. There are more advanced topics we will cover as the course continues.

Here is a great post / cheat sheet that contains the most important git commands and what they do:

https://www.git-tower.com/blog/git-cheat-sheet/

The page also contains a decent post about version control in general.