Version Control using Git and CLI

This transcript covers the foundational concepts of **Git** and **version control**. Since you're a beginner, I'll break it down step by step in simple terms.

---

1. Navigating the Command Line

- **Opening Terminal**: The session starts with using the **Terminal (Command Line Interface - CLI)**, which is a tool for interacting with your computer using text commands.
- **Changing Directories**: The command `cd Desktop` is used to navigate to the Desktop.
- **Creating a Directory**: `mkdir Story` creates a new folder (called a directory) named **Story**.
- **Navigating into a Directory**: `cd Story` moves inside the **Story** directory.

---

2. Creating and Editing Files

- `touch chapter1.txt` creates an empty text file called **chapter1.txt**.
- `open chapter1.txt` opens the file for editing.
- After writing content, the file is **saved**.

---

3. Initializing a Git Repository

- **What is Git?**
  Git is a version control system that helps track changes in your files and allows you to revert to previous versions if needed.
- **Initializing Git**
  - Running `git init` inside the **Story** folder turns it into a **Git repository**.
  - This creates a hidden folder called `.git` that stores all Git-related information.
  - You can see hidden files using `ls -a`.

---

4. Understanding Git's Key Areas

1. **Working Directory**: This is where you create and edit files.
2. **Staging Area**: A middle ground where files are prepared before saving.
3. **Local Repository**: Where committed changes are stored as different versions (save points).

---

5. Checking Git Status

- `git status` shows which files are **untracked** (i.e., Git is not watching them yet).

- Newly created files appear in **red**, meaning they are not in Git's staging area.

---

6. Adding Files to the Staging Area

- `git add chapter1.txt` moves the file from the working directory to the **staging area**.
- Running `git status` again will show the file in **green**, meaning it is now staged and ready to be committed.

---

7. Committing Changes

- A **commit** is like saving a version of your files.
- `git commit -m "Complete chapter 1"` saves the staged files with a message.
- The commit message should always be **clear and descriptive**.

---

8. Viewing Commit History

- `git log` shows a history of all commits.
- Each commit has:
  - A **unique identifier (hash)**.
  - The **author** of the commit.
  - A **timestamp**.
  - The **commit message**.

---

9. Adding and Committing Multiple Files

- Creating two new files: `touch chapter2.txt chapter3.txt`
- Running `git status` will show them as **untracked**.
- Instead of adding them one by one, you can use:
  - `git add .` (adds all new and modified files at once).
- Then commit with `git commit -m "Complete chapter 2 and 3"`.

---

10. Rolling Back Changes

- If you accidentally mess up a file, Git allows you to revert back to a previous version.
- `git checkout` is used to **restore** an older version of a file.
- This ensures that even if you make mistakes, you don’t lose your work.

Why Use Git?

1. **Version Control**: Keep track of changes and roll back if needed.
2. **Collaboration**: Multiple people can work on the same project without conflicts.
3. **Backup**: Your work is saved in different stages, preventing data loss.

Remote GitHub Repositories

This transcript introduces several fundamental **Git and GitHub** concepts. Let me break them down in a beginner-friendly way:

---

1. Local vs. Remote Repositories

- **Local Repository:** A Git repository that exists only on your computer.
- **Remote Repository:** A repository hosted on a platform like **GitHub**, allowing others to access it.

👉 The lesson shifts focus from **local version control** to **pushing code to a remote repository on GitHub**.

---

2. Creating a GitHub Repository

- **Sign up/Login:** You need a **GitHub account** to host remote repositories.
- **Creating a Repository:** On GitHub, click the **"+" button** → **New repository**.
- **Public vs. Private:**
  - **Public repositories** are visible to everyone.
  - **Private repositories** are restricted to you (unless you share access).

🔹 **Example:** Creating a repo called `"Story"` with a description `"My masterpiece."`

---

3. Understanding Open Source

- **Public repositories on GitHub allow you to learn from others' code.**
- You can explore repositories like `"Flappy Bird"` to see how others implement things.
- **Contributing to Open Source:** Once you gain experience, you can help improve existing projects.

---

4. Linking Local and Remote Repositories

After creating a GitHub repository, we need to connect our **local repository** to this **remote repository**.

Steps to push an existing local repository to GitHub:

1️⃣ **Navigate to the project folder on your computer:**

cd Story

(This ensures you're inside your project directory.)

2️⃣ **Check commit history:**

git log

(This shows previously saved versions of the project.)

3️⃣ **Set up a Remote Repository (Link GitHub to your local repository):**

git remote add origin <GitHub repository URL>

- `"origin"` is the **default name** for the remote repo.
- `<GitHub repository URL>` is the **web address of your GitHub repo**.

🚀 **Why use `origin`?**
While you can name your remote anything (even `"bacon"`), `"origin"` is the convention, making it easier for others to understand your project.

---

5. Pushing Local Changes to GitHub

Once the remote repository is linked, push your local files to GitHub:

git push -u origin main

Breaking it down:

- `"git push"` → Sends local files to GitHub.
- `-u` → Links the local **main branch** with the remote **main branch**.
- `"origin"` → The remote repository on GitHub.
- `"main"` → The branch where the code is stored (previously called `"master"`).

👉 **After this command, all your commits (saved changes) are uploaded to GitHub.**

---

6. Checking if the Upload Was Successful

- After pushing, refresh your **GitHub repository page**.
- Your files should now be visible on GitHub.
- You can see past **commit messages** and changes.

---

7. Understanding Git's Workflow

Git tracks project changes through **commits and branches**:

Key Git Areas:

📁 **Working Directory:** Where your files exist before being added to Git.
📂 **Staging Area:** Where files go after using `"git add"` (prepares them for commit).
📜 **Local Repository:** Where commits are stored on your computer.
🌐 **Remote Repository:** The GitHub version of your project.

---

8. What's Next?

The transcript hints at the next lesson, covering:
✅ `.gitignore`: A special file that tells Git to **ignore certain files** (like passwords or API keys).

Summary

- **Git tracks changes locally.**
- **GitHub stores code remotely for collaboration.**
- **Linking local and remote repositories lets you push changes online.**
- **You can explore public GitHub projects to learn from real-world code.**

Would you like me to clarify any specific part? 😊

.gitignore

This transcript teaches about `.gitignore`, which is a special file used in Git to prevent certain files from being committed to a Git repository. Let's break down the concepts in a **simple way** step by step.

---

1. What is a `.gitignore` file?

A `.gitignore` file is used to specify which files or folders should be ignored by Git when committing changes. This is useful for:

- **Sensitive files** (like passwords, API keys)
- **User-specific files** (like system configuration files)
- **Unnecessary files** (such as logs or temporary files)

For example, you may have a file called `secrets.txt` that contains passwords. You don't want it to be uploaded to **GitHub**, so you add it to `.gitignore`.

---

2. How to Create a `.gitignore` File

To create a `.gitignore` file in a project:

Open a **terminal** and go to your project folder:
cd Desktop
mkdir Project
cd Project

1.

Create a few test files:
touch file1.txt file2.txt file3.txt secrets.txt

2.

Create the `.gitignore` file:
touch .gitignore

3.

This file is **hidden**, but you can see it by running:

ls -a

---

3. How to Use `.gitignore`

Open the `.gitignore` file in a text editor (like **VS Code**) and list the files to be ignored:

```
# Ignore the secrets file
secrets.txt

# Ignore MacOS system files
.DS_Store
```

Now, Git will **ignore** these files when committing.

---

4. How `.gitignore` Works in Git

Let's see what happens **before using `.gitignore`**:

```
git init  # Initialize Git in the project
git add . # Add all files
git status # Show the files to be committed
```

This will show **all files**, including **`secrets.txt`** and **`.DS_Store`**.

Now, to **remove unwanted files from Git tracking**:

```
git rm --cached -r .
```

Then, add the **`.gitignore`** file and commit only necessary files:

```
git add .
git status # Now, only required files are staged
git commit -m "Initial Commit"
```

---

5. Using Wildcards in `.gitignore`

You can use **wildcards** to ignore multiple files of a certain type:

- `*.log` → Ignore all log files
- `*.tmp` → Ignore all temporary files
- `node_modules/` → Ignore the entire `node_modules` folder

Example `.gitignore` for a **Node.js project**:

```
# Ignore dependencies
node_modules/
```

```
# Ignore logs
*.log

# Ignore system files
.DS_Store
```

GitHub provides **pre-made templates** for `.gitignore` files. You can find them here.

---

6. Why is `.gitignore` Important?

- **Security**: Prevents sensitive data from being uploaded.
- **Clean Repo**: Avoids unnecessary files in your repository.
- **Collaboration**: Helps teammates avoid downloading useless files.

Summary

1. **`.gitignore` prevents Git from tracking certain files.**
2. **Create a `.gitignore` file and list files to be ignored.**
3. **Use `git rm --cached -r .` to remove unwanted files from staging.**
4. **Use wildcards (`*`) to ignore multiple files of the same type.**
5. **GitHub has ready-made `.gitignore` templates for different projects.**

Would you like a **practical exercise** to practice `.gitignore`? 🚀

Cloning

This transcript introduces fundamental **Git** and **GitHub** concepts, particularly **cloning repositories** and why it's useful. Since you're a beginner, I'll break it down into **simple terms** with examples to help you understand better.

Key Concepts Covered in the Lesson:

1. Pushing vs. Cloning

- In the previous lesson (as mentioned in the transcript), you learned about **pushing** a local repository to GitHub.
- Now, you are learning about **cloning**, which is the reverse: **downloading a remote repository** from GitHub to your computer.

---

2. What is Cloning?

- **Cloning** means making a **copy** of an existing GitHub repository onto your computer.
- This allows you to have a **local version** of the project, including **all its versions and commits**.

The command to do this is:
git clone <repository_url>
Example:
git clone https://github.com/example/repository.git

-
- **Why clone?**
    - To use someone else's open-source project and customize it.
    - To **contribute** to open-source projects (fix bugs, add features).
    - To learn by **reading and modifying existing code**.

---

3. Example of Cloning a Repository

- The instructor demonstrates cloning **QuakeJS**, a JavaScript-based browser game.
- Steps:
    1. **Find the repository URL** (on GitHub).

Run:
git clone <quakejs_url>

2.

Change into the project directory:
cd quakejs

3.

Install dependencies (packages required for the project):
npm install

4.

Set up the required environment:
<command given in documentation>

5.

Start the server:
node <server_command>

6.
7. **Run the game in a browser.**

---

4. Example: Cloning and Running Wordle

- Wordle is a famous word-guessing game that was acquired by *The New York Times*.
- Someone created an **open-source version** of Wordle.
- You can **clone** and **modify** it to:
    1. Play unlimited words per day.
    2. Change the colors or features.
    3. Add new words.
- Steps:

Clone the repository:
git clone <wordle_url>

1.

Move inside the project:
cd word-mastermind

2.

Install required packages:
npm install

3.

Start the project:
npm start

4.

5. Open the browser and **play the game**.

---

5. Understanding Open-Source Development

- **Why read and modify open-source code?**
    - It improves your coding skills.
    - You can contribute and **collaborate** with others.
    - You get hands-on experience with **real-world projects**.
- **How to find beginner-friendly projects?**
    - The lesson mentions **"awesome-for-beginners"**, a curated list of beginner-friendly open-source projects.
    - You can find it on GitHub and choose a project that interests you.

---

6. Next Steps: Learning Pull Requests and Branching

- Cloning is the first step in working with GitHub projects.
- The **next lessons** will teach:
    - **Branches** → Making changes separately without affecting the main project.
    - **Pull Requests** → Requesting project owners to accept your changes.
    - **Merging** → Combining different changes into one.

Summary

1. **Cloning (`git clone`) allows you to copy a GitHub repository to your local system.**
2. **It lets you run and modify open-source projects like QuakeJS and Wordle.**
3. **You can contribute to open-source projects and practice coding by reading and modifying existing code.**
4. **Beginner-friendly projects are available on GitHub, such as "awesome-for-beginners".**
5. **Next, you'll learn about Pull Requests, Branching, and Merging.**

Would you like a hands-on **Git/GitHub tutorial** to practice cloning and modifying a repository? 😊

Branching and Merging

This transcript explains **Git branching and merging**, which are fundamental concepts in **version control**. Since you're new to Git and GitHub, I'll break everything down step by step in a **simple way**.

---

1. What is Git and Version Control?

Before jumping into branches, let's understand what Git does:

- **Git** is a tool that helps **track changes** in your code, allowing you to go back to previous versions if needed.
- **Version Control** helps manage different versions of your project efficiently.

---

2. What is a Branch in Git?

Imagine you are writing a book. You have the **main storyline** (called the **main branch**). Now, you get an idea to **add a new plot** but don't want to change the main storyline yet.

What can you do?

- You create a **new branch** and work on your idea separately.
- If you like the changes, you can **merge** them back into the main storyline later.
- If you don't like the changes, you can simply delete the branch without affecting the main book.

In Git, the **main storyline** is called the **main branch** (previously called **master** in older versions), and **new ideas/changes** are done in **new branches**.

---

3. Creating and Switching Branches in Git

Let's see how you can **create and switch branches** in Git.

Step 1: Create a New Branch

To create a new branch:

git branch alien-plot

This creates a branch named **alien-plot**.

Step 2: View Available Branches

To check the branches you have:

git branch

The current branch you're on will be marked with an **asterisk (*)**.

Step 3: Switch to the New Branch

To move to the new branch:

git checkout alien-plot

Now, you are working inside **alien-plot** instead of the **main** branch.

---

4. Making Changes in a Branch

Now that you're inside the **alien-plot** branch, you can make changes to your files.

For example:

1. Modify **chapter1.txt** and **chapter2.txt** by adding an alien storyline.
2. Save the files.

Step 4: Stage and Commit Changes

Now, you need to **tell Git to track these changes**:

git add .

Then, commit the changes with a message:

git commit -m "Modify chapter 1 and 2 to have an alien theme"

---

5. Switching Back to the Main Branch

At any time, you can go back to the **main** branch:

git checkout main

Your changes in **alien-plot** will **not be lost**. You can switch back and forth between branches.

---

6. Adding New Changes in the Main Branch

While the **alien-plot** branch exists, you can still work on the **main branch**.

For example, if you add a **new file** (`chapter4.txt`), you commit it as usual:

git add .
git commit -m "Add chapter 4"

Now, the **main branch** has **chapter4.txt**, but the **alien-plot** branch does not.

---

7. Merging a Branch into the Main Branch

After testing the **alien-plot** branch, you decide to merge it into the main branch.

Step 1: Switch to the Main Branch
git checkout main

Step 2: Merge the Branch
git merge alien-plot

This brings all changes from **alien-plot** into **main**.

Step 3: Resolve Conflicts (If Any)

Sometimes, Git will ask you to manually fix conflicts if the same lines were changed in different branches.

Step 4: Push the Merged Code to GitHub
git push origin main

Now, your main branch on **GitHub** is updated with the changes.

---

8. Checking Branch History (Git Log & Network Graph)

To see the history of commits:

git log

This shows which commits were made on which branches.

On **GitHub**, you can see a graphical representation of branches and merges under:
📌 **Repository > Insights > Network**

---

9. Summary

✅ **Branches allow you to work on new ideas without affecting the main project.**
✅ **You can create, switch, and merge branches using simple Git commands.**
✅ **Merging helps bring tested changes back into the main branch safely.**
✅ **Version control ensures that even if something goes wrong, you can always revert back.**

---

10. Next Steps for You

1️⃣ **Try creating your own Git repository** (on your PC and GitHub).
2️⃣ **Create a branch, make changes, and merge it** into the main branch.
3️⃣ **Use `git log` and `git branch`** to explore your repository structure.

Would you like a **step-by-step practice task** to help you understand better? 😊

Forking and Pull Requests

This transcript explains the concepts of **GitHub collaboration**, specifically how developers can **contribute to open-source projects** or work as a team using Git and GitHub. Below is a **detailed breakdown** of the concepts explained in simple terms:

---

1. Local and Remote Repositories

- Git works with **four main areas**:

    1. **Working Directory** – This is where you make changes to your files.
    2. **Staging Area** – A place where changes are prepared before committing.
    3. **Local Repository** – A version-controlled copy of your project stored on your computer.
    4. **Remote Repository** – A version-controlled copy stored on **GitHub** (or another Git server), used for collaboration.
- When you create a repository on your computer and then upload (push) it to GitHub, it becomes a **remote repository**.

---

2. Forking a Repository

- If someone else wants to contribute to your project but **does not have direct write access**, they need to **fork** your repository.
- **Forking** means creating a personal copy of another user's GitHub repository under their own account.
- **Example from the transcript:**
    - Angela owns a repository.
    - "Bangela" (her evil twin) wants to make changes but doesn't have permission.
    - So, Bangela **forks** the repository, making an independent copy in her GitHub account.

---

3. Cloning vs. Forking

- **git clone**: Downloads an exact copy of a repository to your local computer so you can work on it.
- **Forking**: Creates a duplicate repository on GitHub under your account, allowing you to make changes without affecting the original repository.

---

4. Making Changes in the Forked Repository

- After forking, Bangela now has her own copy.
- She **clones** the forked repository to her local machine and makes changes (such as adding features or fixing bugs).

- She **commits** these changes to her local repository and then **pushes** them to her forked repository on GitHub.

---

5. Pull Requests (PRs)

- Since Bangela does not have permission to directly update Angela's original repository, she **creates a Pull Request (PR)**.
- A **Pull Request** is a way of **suggesting changes** to the original repository.
- **Why is it called a "Pull" request?** Because Angela (the original owner) needs to **pull** the changes into her repository; Bangela **cannot push** directly.
- In the Pull Request, Bangela can:
    - Explain **what changes** she made.
    - Justify **why the changes** are beneficial.
    - Add **comments** or **notes** for Angela.

---

6. Reviewing a Pull Request

- Angela, the owner of the original repository, receives the PR.
- She can:
    1. **Review the changes** – Compare the old and new versions.
    2. **Approve or reject** – If the changes are useful, she can **merge** them into the original repository.
    3. **Comment on the PR** – Suggest improvements if needed.

---

7. Merging a Pull Request

- If Angela likes the changes, she **merges** them.
- Merging creates a new commit in the original repository.
- Now, Bangela's changes become **part of the official project**.

---

8. Open-Source Collaboration

- This system allows **open-source projects** to function efficiently.
- Instead of letting **everyone modify the original repository**, only **trusted contributors** get direct access.
- Others can **fork, modify, and request approval** via Pull Requests.

Key Takeaways

✅ **Forking** = Making a personal copy of a repository.
✅ **Cloning** = Downloading a repository to work on locally.

✅ **Pull Requests** = Propose changes to a repository you don’t own.
✅ **Merging** = The repository owner accepts changes and adds them.

Would you like me to explain any specific Git/GitHub commands in detail? 🚀