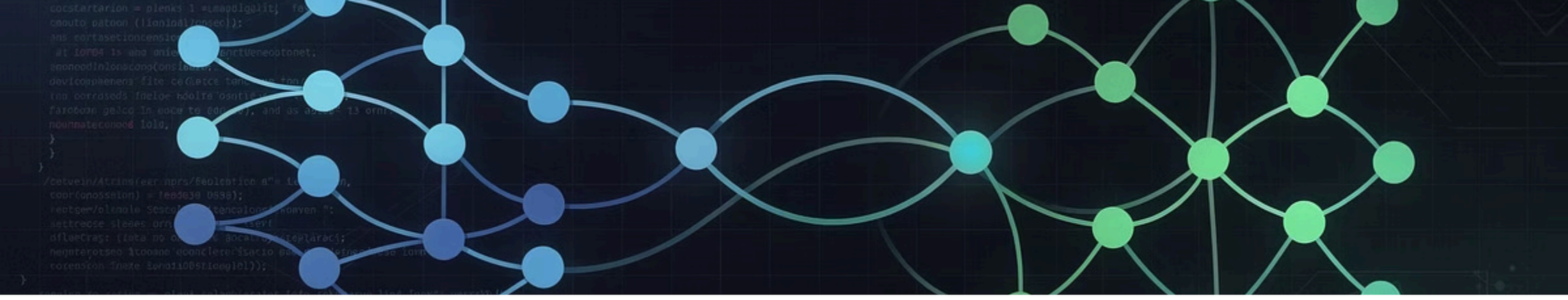


GIT FUNDAMENTALS

VERSION CONTROL FOR MODERN DEVOPS WORKFLOWS

CORE CONCEPTS

Understand the essentials of Git and version control.

COLLABORATION

Learn how to work effectively with others using Git.

DEVOPS WORKFLOWS

Apply Git principles to enhance your development process.

GIT FUNDAMENTALS AND BASIC OPERATIONS

This session will introduce the basics of Git for robust version control, with a strong hands-on focus to ensure practical understanding.

EVER WISH YOU COULD UNDO CODE CHANGES?

CODE MANAGEMENT

Efficiently track, save, and manage your code's evolution.

VERSION HISTORY

Revert to previous states and explore past changes with ease.

SESSION AGENDA & KEY TAKEAWAYS

AGENDA

INTRODUCTION & SETUP

LOCAL GIT WORKFLOW

REMOTE GITHUB WORKFLOW

BASIC COLLABORATION

WRAP-UP

KEY TAKEAWAYS

- Understand Git as a version control tool
- Learn essential Git commands
- Use Git locally and with GitHub
- Build confidence through hands-on practice

WHAT IS GIT?

Git is a **distributed version control system** created by Linus Torvalds, the creator of Linux.

WHAT DOES THIS MEAN?

VERSION HISTORY

Records every modification, creating a full project timeline.

MISTAKE REVERSAL

Easily revert to any past state to fix errors without losing work.

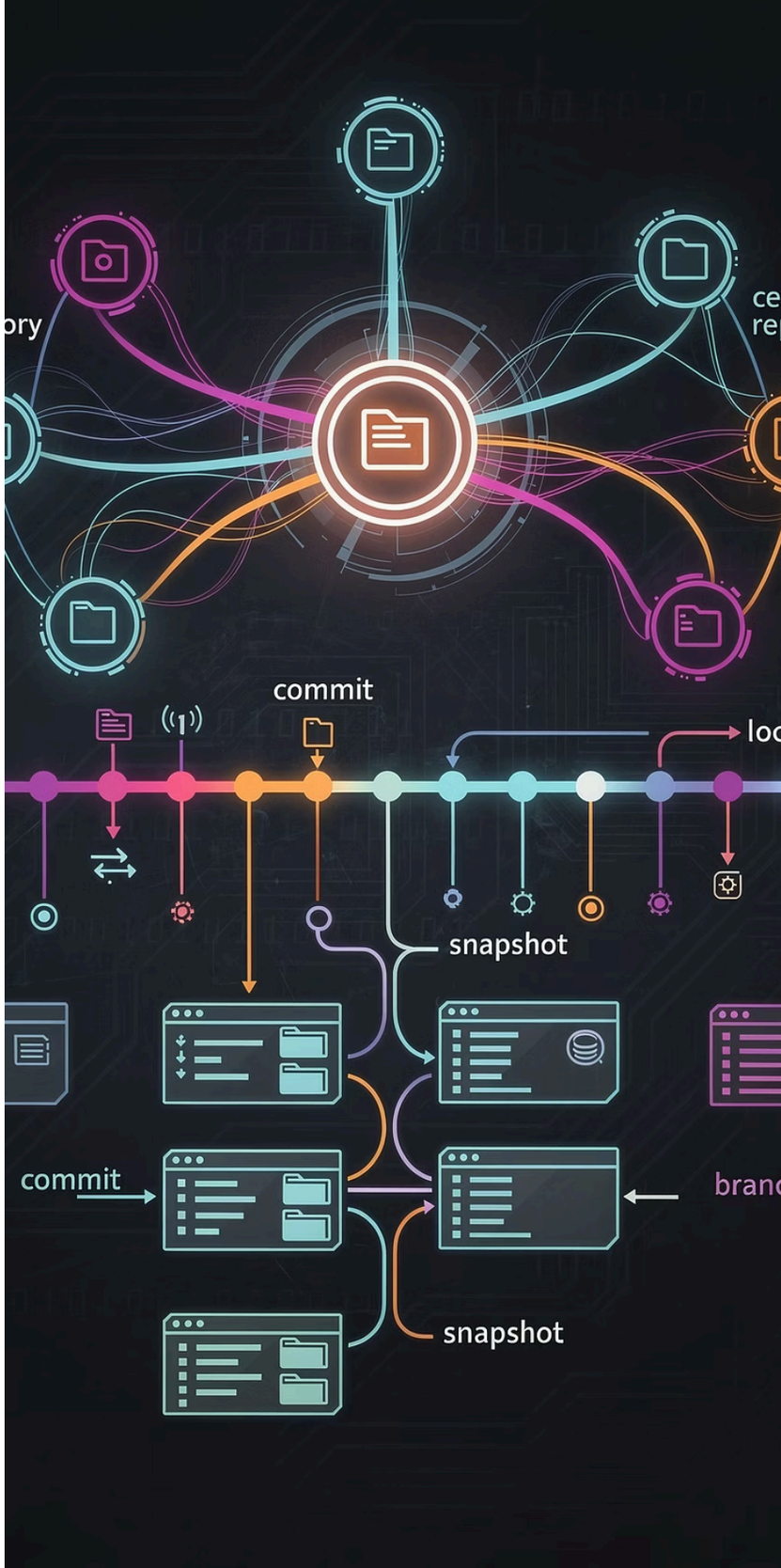
TEAM COLLABORATION

Enables multiple developers to work efficiently on the same project.

DISTRIBUTED COPIES

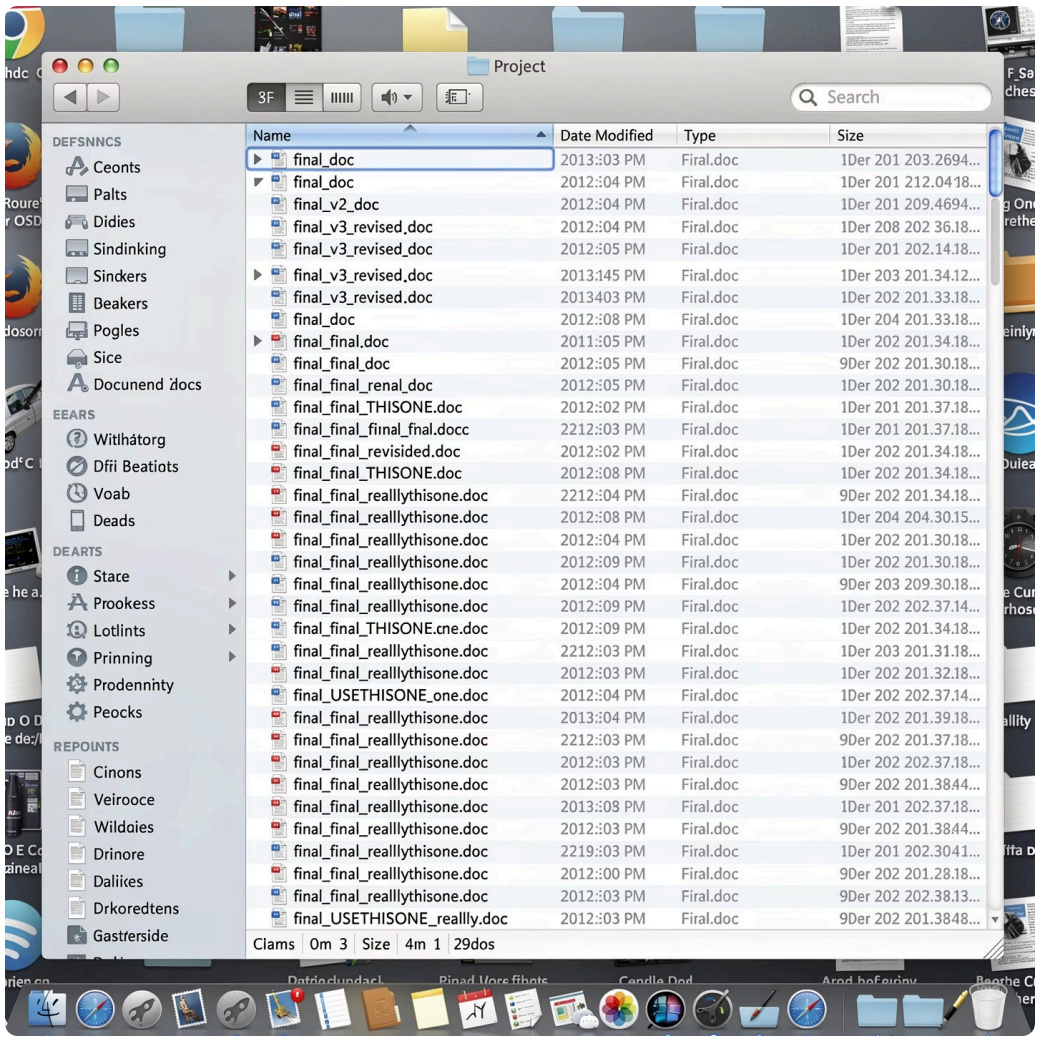
Every developer has a complete local copy for resilience and offline work.

> “Git is a time machine for your code.”



WHY USE GIT?

THE CHAOS WITHOUT GIT



Before Git, version control was often manual and error-prone:

- `project_final.doc`
- `project_final_v2.doc`
- `project_final_really_this_one.doc`
- Messy, confusing, no clear tracking.

THE ORDER WITH GIT

Git transforms this chaos into a structured workflow:

- Structured history of all changes.
- Safe experimentation with branches.
- Easy collaboration among team members.
- Automatic backups and recovery (especially with platforms like GitHub).

KEY GIT CONCEPTS

REPOSITORY (REPO)

Your project's central storage, containing all files, changes, and history.

COMMITS

Immutable snapshots of your project at a specific point in time, each with a unique ID.

STAGING AREA

A "waiting room" where you prepare changes for your next commit, selecting what to include.

THE GIT WORKFLOW: A THREE-STEP PROCESS

SETUP & INSTALLATION

Installing Git is the first step to leveraging its powerful version control capabilities. Follow these simple instructions tailored to your operating system to get Git up and running.

INSTALL GIT

MACOS

For macOS users, the easiest way to install Git is via **Homebrew**. Open your terminal and run:

```
brew install git
```

LINUX (UBUNTU/DEBIAN)

On Debian-based Linux distributions like Ubuntu, you can install Git using your package manager:

```
sudo apt install git -y
```

WINDOWS

Windows users should download the official installer from the Git website. Follow the installation wizard, accepting the default options.

git-scm.com/download/win

VERIFY INSTALLATION

Once the installation is complete, confirm Git is successfully installed and accessible from your command line by executing:

```
git --version
```

You should see output similar to `git version 2.39.2 (Apple Git-143)`, indicating a successful setup.

CREATE YOUR GITHUB ACCOUNT

Setting up your GitHub account is a straightforward process, opening the door to powerful collaboration and version control features.

ACCOUNT SETUP STEPS

VISIT GITHUB.COM

Navigate to the official GitHub website to begin the registration process.

SIGN UP

Choose a unique username, provide your email address, and set a strong password.

VERIFY EMAIL

Check your inbox for a verification email from GitHub and click the confirmation link.

LOG IN & EXPLORE

Once verified, log in to your new account and explore your personal profile and dashboard.

WHY GITHUB IS ESSENTIAL

CLOUD BACKUP

Securely store your code projects in the cloud, protecting against local data loss.

TEAM COLLABORATION

Seamlessly work with others on shared projects, tracking contributions and changes.

PULL REQUESTS

Facilitate code reviews and integrate new features with a structured merging process.

CI/CD INTEGRATION

Automate testing, building, and deployment of your code with integrated tools.

LOCAL GIT WORKFLOW: `git init`

The foundation of any Git project is initializing a repository. This command transforms a standard directory into a Git-managed project, ready for robust version control.

UNDERSTANDING `git init`

```
git init
```

Purpose

Creates a new, empty Git repository in the current directory, setting up the necessary internal Git structures to begin tracking changes.

Example

```
mkdir my-new-project
cd my-new-project
git init
```

Output

Upon successful execution, you will see a message similar to: `Initialized empty Git repository in /path/to/your/project/.git/`

Tip

This command creates a hidden `.git` folder. This folder contains all the information Git needs for version control; never delete or modify its contents manually!

LOCAL GIT WORKFLOW: `git status`

After making changes, the `git status` command is your essential tool for understanding the current state of your working directory and staging area. It provides a snapshot of what Git knows about your project.

```
git status
```

PURPOSE

Performs a quick "health check" of your repository, showing which files are modified, staged, or untracked, and suggesting next actions.

OUTPUT DETAILS

The output clearly indicates:

- Untracked files (new files Git hasn't seen yet).
- Changes not staged for commit (modified files not added to the staging area).
- Changes to be committed (files in the staging area, ready for your next commit).
- Your current branch name.
- Helpful suggestions for common Git commands.

TIP

Run `git status` **OFTEN!** It's your best friend for keeping track of your changes and understanding where you are in the Git workflow.

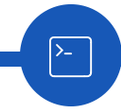
LOCAL GIT WORKFLOW: `git add`

The `git add` command is crucial for preparing your changes to be recorded in the repository. It moves modifications from your working directory to the staging area, allowing you to curate what goes into your next commit.

```
git add <file>
git add .
```

PURPOSE

This command "stages" changes, meaning it adds a snapshot of the current state of a file or set of files to the staging area. It's like putting selected items into a shopping cart before you "checkout" with a commit.



EXAMPLE

To stage a new file named `index.html`:

```
echo "<h1>Hello Git</h1>" > index.html
git add index.html
```

Or to stage all changes in the current directory:

```
git add .
```

TIP

Using `git add .` is a quick way to stage all modified and untracked files in your current directory and its subdirectories. Be cautious with it; always run `git status` beforehand to ensure you're only staging intended changes.

LOCAL GIT WORKFLOW: `git commit`

The `git commit` command is the definitive step in your local Git workflow. It transforms your staged changes into a permanent, unchangeable record within your project's history.

```
git commit -m "Your descriptive commit message here"
```

PURPOSE

The `git commit` command creates a permanent snapshot of your staged changes in the repository's history, along with a descriptive message. It's the point where your work is officially saved.

OUTPUT DETAILS

Upon successful commit, Git provides a unique **Commit ID** (a SHA-1 hash), a summary of the **changes made** (insertions/deletions), and a list of **files updated** within that commit. It also indicates your current branch.

TIP

Always write **clear and concise commit messages**. A good commit message explains *what* was changed and *why*, making your project history easy to understand and navigate for yourself and your collaborators.

Each commit acts as a checkpoint, allowing you to easily revisit or revert to previous states of your project.

LOCAL GIT WORKFLOW: `git log`

The `git log` command is your window into the history of your repository. It allows you to review past commits, understand changes over time, and trace the evolution of your project.

```
git log
git log --oneline
```

PURPOSE

Displays a detailed list of all commits in the current branch, showing the commit hash, author, date, and commit message. It's essential for understanding who changed what and when.

`--oneline`

A convenient option to condense each commit into a single line, displaying only the first few characters of the commit hash and the commit message. Ideal for a quick, high-level overview.

NAVIGATING HISTORY

Beyond simple viewing, `git log` supports various flags to filter by author, date, or specific files, allowing you to pinpoint exact changes or contributors.

HANDS-ON EXERCISE: TRACING HISTORY

Let's create a simple project, make some changes, and then use `git log` to review its history.

INITIALIZE PROJECT

Create a new directory named `my-history-repo` and initialize a Git repository within it using `git init`.

FIRST COMMIT

Create a file (e.g., `index.html`), add some content, stage it with `git add .`, and commit it with a descriptive message like "Initial commit".

SECOND COMMIT

Modify `index.html`, add more content, stage the changes, and commit again with a message like "Added header section".

VIEW HISTORY

Now, run `git log` and `git log --oneline` in your terminal to observe the commit history you've created.

REMOTE REPOSITORIES WITH GITHUB

A remote repository is a version of your Git project hosted on the internet or a network, such as GitHub. It acts as a central hub for collaboration and backup.

WHY USE A REMOTE REPOSITORY?

BACKUP

Safeguards your project history against local data loss.

TEAM COLLABORATION

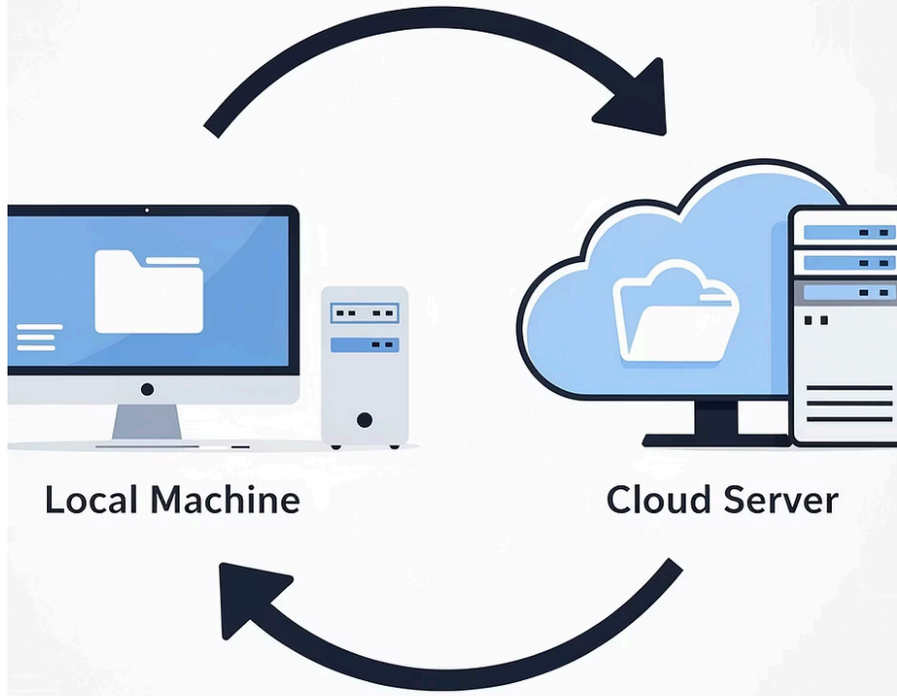
Allows multiple developers to work on the same project simultaneously.

CI/CD PIPELINE

Integrates with automated build, test, and deployment systems.

OPEN-SOURCE CONTRIBUTIONS

Facilitates sharing and contributing to public projects.



Local Git Workflow: Connecting to Remote & Pushing

After committing changes locally, you'll want to share them with a remote repository, typically hosted on platforms like GitHub. This involves two main steps: adding the remote and then pushing your local commits.

```
git remote add origin <repository_url>
git push -u origin main
```

Create Remote First

Before using `git remote add`, ensure your repository is already created on GitHub or your chosen remote hosting service. This provides the URL destination for your local project.

Simplify Future Pushes

The `-u` (or `--set-upstream`) flag in `git push -u origin main` establishes a link between your local `main` branch and the remote `main` branch. After this, subsequent pushes can be simplified to just `git push`.

LOCAL GIT WORKFLOW: `git clone` & `git pull`

These commands are essential for interacting with remote repositories, enabling you to get a copy of a project and keep your local work updated with the latest changes.

```
git clone <repository_url>
git pull
```

`git clone`

Downloads an entire repository from a remote server (like GitHub) to your local machine, creating a new local Git repository.

`git pull`

Fetches and integrates new changes from the remote repository into your current local branch, keeping your project up-to-date.

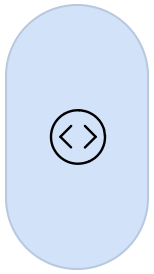
BEST PRACTICES

- **Commit often:** Save your progress regularly.
- **Use `.gitignore`:** Exclude unwanted files.
- **Pull before push:** Always update your local repo before pushing your own changes.

BASIC COLLABORATION: PULL REQUESTS (PRS)

Pull Requests (PRs) are a fundamental mechanism in Git-based workflows, especially on platforms like GitHub. They are used to propose changes to a codebase and request a review from collaborators before those changes are merged into the main branch.

WHY PRS MATTER



CODE QUALITY

PRs enforce code review, where peers examine proposed changes for bugs, style consistency, and adherence to best practices, significantly improving the overall quality of the codebase.

TEAM VISIBILITY

They provide transparency within the development team, allowing everyone to see what changes are being proposed and understand their impact on the project.

SAFER MERGING

By requiring review and approval, PRs act as a safeguard, preventing unintended errors or regressions from being introduced directly into critical branches.

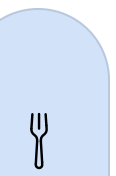
DISCUSSION & COLLABORATION

PRs serve as a dedicated forum for discussion around specific changes, fostering collaboration and enabling collective decision-making before any code is officially integrated.

FORKING & CREATING PULL REQUESTS (PRS)

Forking is a fundamental GitHub operation that allows you to create your own personal copy of someone else's repository. This copy lives on your GitHub account, independent of the original project. It's especially useful for contributing to open-source projects or experimenting without directly affecting the upstream repository.

PULL REQUEST WORKFLOW STEPS



FORK THE REPOSITORY

Start by creating a copy of the original (upstream) repository on your own GitHub account. This gives you a personal playground for your changes.

CLONE YOUR FORK

Download your forked repository from GitHub to your local development machine using `git clone`. This creates a local working copy.

MAKE YOUR CHANGES

Work on the code locally, implementing new features, fixing bugs, or making improvements as needed for your contribution.

COMMIT & PUSH

Stage and commit your changes locally, then push these commits from your local machine to your forked repository on GitHub.

CREATE A PULL REQUEST

From your forked repository on GitHub, open a Pull Request to the original (upstream) repository, proposing your changes for review and eventual merging.

Common Git Pitfalls and Troubleshooting

Navigating Git can sometimes lead to unexpected errors. Understanding these common pitfalls and their solutions will help you troubleshoot effectively and maintain a smooth workflow.

`fatal: not a git repository`

This error indicates that you are trying to run Git commands in a directory that is not a Git repository. Ensure you are inside a folder that has been initialized with `git init` or cloned from a remote repository.

`nothing to commit, working tree clean`

This message means there are no changes in your working directory that have been staged for a commit. Use `git status` to see pending modifications, and `git add .` or `git add <file>` to stage them.

`detached HEAD state`

This occurs when you are directly on a commit hash instead of a branch. While useful for inspecting past states, it's generally recommended to create a new branch (`git switch -c <new-branch>`) if you intend to make new changes.

`Permission denied (publickey)`

This often indicates issues with SSH key authentication when trying to interact with a remote repository. Verify your SSH keys are correctly set up on your machine and added to your Git hosting service (e.g., GitHub).

TROUBLESHOOTING TIPS

HELPFUL COMMANDS

`git status`

Your first step to understand the current state of your repository and identify modified, staged, or untracked files.

`git remote -v`

Lists the remote repositories associated with your local repo, showing their URLs. Useful for verifying connections.

`git reset`

A powerful command to undo changes, ranging from unstaging files (`--soft`) to reverting history (`--hard`). Use with caution!

`git checkout -- <file>`

Discards changes in your working directory for a specific file, reverting it to the last committed state.

`git commit --amend`

Allows you to modify the last commit, such as changing the commit message or adding/removing files. Avoid amending pushed commits.

GENERAL ADVICE

READ ERROR MESSAGES CAREFULLY

Git error messages often provide precise instructions or hints about what went wrong and how to fix it.

SEARCH GITHUB ISSUES/STACKOVERFLOW

Chances are, someone else has encountered the same issue. Leverage online communities for solutions and best practices.

MISTAKES ARE NORMAL

Learning Git involves making and fixing errors. Don't be afraid to experiment and undo changes; that's what version control is for!

WRAP-UP & HOMEWORK

CORE GIT FLOW RECAP

`git add`

Stage changes

`git commit`

Save snapshot

`git push`

Upload to remote

`git pull`

Download updates

YOUR HOMEWORK

Practice makes perfect! Apply what you've learned by completing the following tasks:

- **Clone a public repository:** Find an interesting open-source project on GitHub and clone it to your local machine.
- **Add a new file:** Create a simple new file (e.g., a `README.md` or a text file) within your cloned repository.
- **Commit & Push:** Stage your new file, commit it with a meaningful message, and push your changes to your local clone.

NEXT SESSION: BRANCHING STRATEGIES

In our next session, we'll dive into the critical concepts of Git branching, including creating feature branches, understanding different merge strategies, and resolving conflicts gracefully.

WHY SSH INSTEAD OF HTTPS?

When interacting with remote Git repositories, especially on platforms like GitHub, SSH (Secure Shell Protocol) offers a more secure and streamlined authentication method compared to HTTPS. Here's why it's often preferred:

ENHANCED SECURITY

SSH provides robust, cryptographic authentication, allowing secure communication between your local machine and GitHub without ever transmitting your password over the network.

PASSWORD-LESS ACCESS

Once your SSH key is set up and added to your GitHub account, you no longer need to enter your username and password for every Git operation, significantly speeding up your workflow.

PROFESSIONAL STANDARD

SSH is the standard authentication method in many professional and enterprise development environments, offering a more robust and scalable solution for team collaboration.

RELIABILITY FOR DEVOPS

Its stateless nature and built-in security features make SSH a more reliable and secure choice for automated scripts and continuous integration/delivery (CI/CD) pipelines in DevOps.

> SSH provides a secure, password-less gateway for seamless interaction with your GitHub repositories.

WHAT IS SSH?

SSH stands for **Secure Shell**. It's a cryptographic network protocol that enables secure communication between two networked machines.

WHY DO WE USE IT WITH GIT?

- **Secure Communication:** Encrypts all traffic between your local machine and remote Git repositories, protecting your data.
- **Password-less Authentication:** Uses public-key cryptography to verify your identity without needing to enter a password for every interaction.
- **DevOps Standard:** Widely adopted in DevOps environments for servers, Git operations, and automated CI/CD pipelines due to its security and efficiency.

SIMPLIFIED AUTHENTICATION MECHANISM

PUBLIC KEY

This key is uploaded to your GitHub account (or other Git service). It's publicly available and used to encrypt messages or verify signatures.

PRIVATE KEY

This key remains securely on your local computer. It decrypts messages encrypted by the public key and creates digital signatures.

IDENTITY VERIFICATION

When you connect to GitHub, they use your public key to challenge your local machine, which then responds using its private key to prove your identity.

SESSION RECAP – KEY TAKEAWAYS

We've covered a lot of ground in Git fundamentals. Here's a quick summary of the most important concepts to remember:

1. GIT AS DISTRIBUTED VERSION CONTROL
2. CORE GIT WORKFLOW
3. UNDERSTANDING REPOSITORIES & COMMITS
4. WORKING WITH GITHUB
5. SECURE SSH AUTHENTICATION

QUESTIONS & DISCUSSION

Let's open the floor for some questions and discussion to consolidate our learning and address any lingering uncertainties. Your feedback helps everyone!

- **What was unclear today?**
- **Which Git command feels most confusing right now?**
- **Where do you think Git will help you the most in your projects?**