**Activity Based Learning – Git and GitHub for DevOps Engineer**

**Initial Configuration for Git**

Before starting with the activities, let's ensure that Git is set up correctly on your local system.

**Steps:**

1. **Install Git:**
   * Download Git from <https://git-scm.com/downloads> and follow the installation instructions for your operating system (Windows, macOS, or Linux).
   * Verify the installation by opening your terminal/command prompt and running:

git --version

1. **Explanation:** This should return the version of Git installed, confirming that Git is correctly installed.
2. **Configure your username:**

git config --global user.name "Your Name"

**Explanation:** This sets your global Git username. This name will appear in your commits, making it easy to identify who made changes in a project.

1. **Configure your email:**

git config --global user.email "your.email@example.com"

**Explanation:** This sets your global email address. This email will be associated with your Git commits, which is useful for tracking who made each commit.

1. **Verify the configuration:**

git config --list

**Explanation:** This command lists the current configuration settings, including your username and email, ensuring everything is correctly configured.

1. **Set up your default editor (optional):** If you prefer to use a specific editor (e.g., VS Code, Sublime Text), set it up as the default editor for Git:

git config --global core.editor "code --wait"

**Explanation:** This sets Visual Studio Code (code) as the default editor. You can replace "code --wait" with your preferred editor’s command.

**What is GitHub and Why Do We Use It?**

**GitHub** is a cloud-based platform for hosting and managing Git repositories. It is widely used for version control and collaboration in software development. GitHub allows multiple people to work on a project at the same time without interfering with each other’s work.

**Why Use GitHub?**

* **Collaboration:** GitHub makes it easy to collaborate with others on code by providing tools like branches, pull requests, and merge capabilities.
* **Version Control:** It helps you track changes in your code over time, allowing you to revert to previous versions if necessary.
* **Remote Backup:** Your code is hosted on GitHub's servers, which acts as a backup, so you don't have to worry about losing it.
* **Open Source:** GitHub hosts open-source projects, allowing anyone to contribute to public repositories.

**Activity 1: Creating a GitHub Repository**

**Scenario:** You need to create a repository on GitHub to store your project remotely, share it with others, and collaborate.

**Steps:**

1. **Create a GitHub account:**
   * Go to <https://github.com> and sign up for an account if you don’t already have one.
2. **Create a new repository:**
   * After logging into GitHub, click on the **“+”** icon in the top right corner and select **New repository**.
   * **Repository name:** Choose a name for your repository, e.g., my-first-repo.
   * **Description (optional):** Provide a short description of your project.
   * **Visibility:** Choose whether the repository should be **Public** (anyone can view) or **Private** (only invited users can view).
   * Click **Create repository**.
3. **Copy the repository URL:** After creating the repository, GitHub will show a page with instructions on how to push an existing repository from the command line. Copy the repository URL, which should look something like this:

https://github.com/username/my-first-repo.git

**Explanation:** A GitHub repository is essentially a remote version of your project where you can push and pull changes. GitHub acts as a cloud-based backup and provides version control tools, so you can track the history of your project.

**Activity 2: Initialize a Local Git Repository and Connect to GitHub**

**Scenario:** You want to work locally on your project and then push your changes to GitHub for collaboration.

**Steps:**

1. **Initialize the repository locally:**

git init

**Explanation:** Initializes a new Git repository in the current directory, creating a .git folder. This marks the directory as a Git repository.

1. **Link the local repository to GitHub:**

git remote add origin https://github.com/username/my-first-repo.git

**Explanation:** This links your local repository to the GitHub repository you just created. The origin is the default name for your remote GitHub repository.

1. **Check the remote repository:**

git remote v

**Explanation:** Displays the remote URL associated with your project. This verifies that you are linked to the correct GitHub repository.

**Activity 3: Adding Files to Git**

**Scenario:** You created a new file index.html in your project, and now you need to track it with Git.

**Steps:**

1. **Add the file to Git:**

git add index.html

**Explanation:** Stages the index.html file for commit, meaning Git will track changes made to this file.

1. **Check the status:**

git status

**Explanation:** Shows the current state of your repository, including any untracked or staged files.

**Activity 4: Committing Changes**

**Scenario:** You made some updates to the index.html file and now want to commit those changes to Git.

**Steps:**

1. **Commit the changes:**

git commit -m "Added initial structure to index.html"

**Explanation:** This command commits the staged changes to your local repository, providing a message that describes what was modified.

1. **View commit history:**

git log

**Explanation:** Displays the commit history, showing the changes made in the project along with the commit IDs and messages.

**Activity 5: Pushing to GitHub**

**Scenario:** After completing the work locally, you need to push the changes to GitHub for sharing and collaboration.

**Steps:**

1. **Push the changes to GitHub:**

git push -u origin main

**Explanation:** Pushes the local commits to the remote GitHub repository. The -u flag establishes a connection between your local main branch and the remote main branch on GitHub, so subsequent pushes can be done with just git push.

**Activity 6: Cloning a GitHub Repository**

**Scenario:** You are collaborating with someone on a project, and you need to get a copy of the repository from GitHub.

**Steps:**

1. **Clone the repository:**

git clone https://github.com/username/my-first-repo.git

**Explanation:** This command creates a local copy of the repository on GitHub, including all files, branches, and commit history.

1. **Navigate to the repository:**

cd my-first-repo

**Explanation:** Changes the working directory to the cloned repository, so you can start working on it.

**Activity 7: Pulling Changes from GitHub**

**Scenario:** You are working on a project with a team, and changes have been made to the main branch. You want to pull the latest changes.

**Steps:**

1. **Pull the latest changes from GitHub:**

git pull origin main

**Explanation:** Fetches and merges the latest changes from the main branch of your GitHub repository into your local repository. This is useful for keeping your local repository up-to-date.

**Activity 8: Managing Branches on GitHub**

**Scenario:** You want to work on a new feature without affecting the main branch, so you need to create a new branch on GitHub.

**Steps:**

1. **Create a new branch on GitHub:**
   * Go to the repository page on GitHub.
   * Click on the **Branch** dropdown and enter a name for the new branch (e.g., feature-login), then click **Create branch**.
2. **Switch to the new branch locally:**

git checkout feature-login

**Explanation:** Switches to the newly created branch, where you can work on the new feature independently of the main branch.

1. **Push the new branch to GitHub:**

git push origin feature-login

**Explanation:** Pushes the feature-login branch to GitHub, making it available for collaboration or future pull requests.

**Activity 9: Creating a Branch Locally**

**Scenario:** You are working on a new feature and need to create a branch to isolate your work from the main branch.

**Steps:**

1. **Create a new branch locally:**

git branch feature-xyz

**Explanation:** This creates a new branch called feature-xyz without switching to it yet.

1. **Switch to the new branch:**

git checkout feature-xyz

**Explanation:** This switches to the newly created branch, so you can start making changes.

1. **Confirm you're on the correct branch:**

git branch

**Explanation:** Lists all branches and marks the current branch with a \*.

**Activity 10: Merging a Branch into Main**

**Scenario:** After completing work on the feature-xyz branch, you want to merge it into the main branch.

**Steps:**

1. **Switch to the main branch:**

git checkout main

**Explanation:** Switches to the main branch where you want to merge the feature branch.

1. **Merge feature-xyz into main:**

git merge feature-xyz

**Explanation:** Merges the feature-xyz branch into the main branch. This incorporates the changes you made in the feature branch into the main branch.

1. **Resolve merge conflicts (if any):**
   * If there are conflicts, Git will mark the conflicting files and you will need to manually resolve them.
   * After resolving conflicts, stage the changes:

git add <conflicted-file>

* + Commit the merge:

git commit -m "Resolved merge conflicts"

**Activity 11: Deleting a Branch Locally**

**Scenario:** After merging a branch, you want to clean up by deleting the feature branch.

**Steps:**

1. **Delete the local branch:**

git branch -d feature-xyz

**Explanation:** Deletes the branch feature-xyz locally, but only if it has been fully merged. If not, use -D (force delete).

1. **Confirm the branch is deleted:**

git branch

**Explanation:** Verify that the branch is no longer listed.

**Activity 12: Fetching Changes from a Remote Repository**

**Scenario:** You want to sync your local repository with the remote repository to get the latest updates.

**Steps:**

1. **Fetch the latest changes:**

git fetch origin

**Explanation:** Fetches the latest changes from the remote repository (origin), but it doesn't merge them into your local branch.

1. **View the changes:**

git log HEAD..origin/main

**Explanation:** Displays the commits that are in the remote repository but not yet in your local repository.

**Activity 13: Pulling Changes from GitHub**

**Scenario:** You want to update your local repository with changes that have been made on GitHub.

**Steps:**

1. **Pull the changes into your current branch:**

git pull origin main

**Explanation:** Fetches and merges changes from the main branch of the remote repository (origin) into your current local branch.

**Activity 14: Reverting Changes**

**Scenario:** You made a mistake in a commit and want to undo it by reverting the commit.

**Steps:**

1. **Revert the last commit:**

git revert HEAD

**Explanation:** This creates a new commit that undoes the changes made in the last commit.

1. **Undo changes before committing (soft reset):**

git reset --soft HEAD~1

**Explanation:** This moves the HEAD pointer back by one commit but keeps the changes staged (ready to commit again).

1. **Completely remove the last commit (hard reset):**

git reset --hard HEAD~1

**Explanation:** This completely removes the last commit, including any changes made in that commit. Use with caution.

**Activity 15: Viewing Differences**

**Scenario:** You want to see what changes have been made to files before committing.

**Steps:**

1. **View differences in unstaged changes:**

git diff

**Explanation:** Shows the differences between your working directory and the staged changes.

1. **View differences between staged changes and the last commit:**

git diff --cached

**Explanation:** Displays the differences between the staged changes and the last commit.

**Activity 16: Stashing Changes**

**Scenario:** You’re in the middle of work but need to switch to another task. You want to save your current changes without committing them.

**Steps:**

1. **Stash your changes:**

git stash

**Explanation:** Saves your uncommitted changes temporarily and restores the working directory to the state of the last commit.

1. **List all stashed changes:**

git stash list

**Explanation:** Lists all the stashed changes, showing stash references like stash@{0}, stash@{1}, etc.

1. **Apply the latest stashed changes:**

git stash apply

**Explanation:** Applies the latest stash to your working directory.

1. **Pop the latest stash:**

git stash pop

**Explanation:** Applies the latest stash and removes it from the stash list.

**Activity 17: Tagging a Commit**

**Scenario:** You want to create a tag for a specific commit to mark a release or significant point in your project.

**Steps:**

1. **Create a lightweight tag:**

git tag v1.0

**Explanation:** Creates a tag named v1.0 for the current commit.

1. **Create an annotated tag:**

git tag -a v1.0 -m "Release version 1.0"

**Explanation:** Creates an annotated tag with a message. Annotated tags are stored as full objects in Git and contain metadata such as the tagger's name, email, and date.

1. **Push tags to the remote repository:**

git push origin v1.0

**Explanation:** Pushes the tag v1.0 to the remote GitHub repository.

**Activity 18: Viewing the Commit History with Graph**

**Scenario:** You want to see a visual representation of your commit history, including branches and merges.

**Steps:**

1. **View commit history as a graph:**

git log --graph --oneline --decorate --all

**Explanation:** Displays the commit history in a graphical format, showing branches and merges in a more visual way.

**Activity 19: Resolving Merge Conflicts**

**Scenario:** You attempt to merge branches, but Git reports a conflict. You need to resolve it.

**Steps:**

1. **Start a merge that causes a conflict:**

git merge feature-xyz

**Explanation:** If there are conflicting changes between your current branch and feature-xyz, Git will mark the conflicts.

1. **Resolve the conflicts manually:**
   * Open the conflicting files and choose the changes you want to keep.
   * Git marks conflicts with markers like <<<<<<<, =======, and >>>>>>>.
2. **Mark the conflicts as resolved:**

git add <conflicted-file>

**Explanation:** After resolving conflicts, stage the files to mark them as resolved.

1. **Complete the merge:**

git commit

**Explanation:** Finalizes the merge after conflicts have been resolved.

**Activity 20: Rebase Instead of Merge**

**Scenario:** You want to incorporate changes from another branch without creating a merge commit.

**Steps:**

1. **Rebase your branch onto main:**

git checkout feature-xyz

git rebase main

**Explanation:** Re-applies your feature-xyz commits on top of the latest commits from the main branch. This avoids creating a merge commit.

1. **Resolve any conflicts during rebase:**
   * If conflicts occur, resolve them in the same way as a regular merge.
   * After resolving, use:

git rebase --continue

1. **Explanation:** This resumes the rebase process after resolving any conflicts.

**Activity 21: Cloning a Remote Repository**

**Scenario:** You want to create a copy of an existing GitHub repository on your local system to start working on it.

**Steps:**

1. **Clone a repository from GitHub:**

git clone https://github.com/username/repository.git

**Explanation:** This command clones the entire repository from the specified URL to your local machine. It also creates a .git directory to track the project.

1. **Navigate to the cloned repository:**

cd repository

**Explanation:** Changes your directory to the cloned repository, so you can start working with it.

**Activity 22: Checking the Status of Your Repository**

**Scenario:** You want to check the current status of your local repository, including staged, unstaged, and untracked files.

**Steps:**

1. **Check repository status:**

git status

**Explanation:** Shows the current status of the repository, including changes that are staged, not staged, and untracked files. It helps identify what needs to be committed.

**Activity 23: Undoing Local Changes**

**Scenario:** You made changes to a file but want to discard them and revert to the last committed state.

**Steps:**

1. **Discard changes in a specific file:**

git checkout -- <file>

**Explanation:** Discards any local modifications in the specified file and restores it to the state of the last commit.

1. **Discard all local changes in the repository:**

git reset --hard

**Explanation:** Resets all files in the working directory to the last commit, discarding any local changes.

**Activity 24: Working with Remote Repositories**

**Scenario:** You need to connect a local repository to a remote repository and push changes.

**Steps:**

1. **Add a remote repository:**

git remote add origin https://github.com/username/repository.git

**Explanation:** Adds a remote repository URL (e.g., GitHub) as the origin for your local repository.

1. **Push changes to the remote repository:**

git push -u origin main

**Explanation:** Pushes the local changes to the main branch of the remote repository and sets the upstream (-u).

**Activity 25: Creating and Switching to a New Branch**

**Scenario:** You want to create a new branch and switch to it in a single command.

**Steps:**

1. **Create and switch to a new branch in one step:**

git checkout -b feature-abc

**Explanation:** This command creates a new branch named feature-abc and switches to it immediately.

**Activity 26: Staging Partial Changes**

**Scenario:** You only want to stage specific parts of the changes in a file, rather than staging the entire file.

**Steps:**

1. **Stage part of a file's changes interactively:**

git add -p <file>

**Explanation:** This command allows you to interactively stage parts of a file by selecting chunks of the changes to be included in the commit.

**Activity 27: Viewing Differences in Staged vs. Unstaged Changes**

**Scenario:** You want to compare the changes you've made in your working directory with the staged version of the file.

**Steps:**

1. **View unstaged changes (differences in working directory):**

git diff

**Explanation:** Shows differences between your working directory and the index (staged changes).

1. **View staged changes (differences between index and last commit):**

git diff --cached

**Explanation:** Displays the differences between the staged changes and the last commit, showing what will be included in the next commit.

**Activity 28: Creating an Alias for Git Commands**

**Scenario:** You want to simplify frequently used Git commands by creating aliases for them.

**Steps:**

1. **Create an alias for a common Git command:**

git config --global alias.co checkout

**Explanation:** Creates an alias so that git co is equivalent to git checkout, making it quicker to use.

1. **Verify the alias:**

git co

**Explanation:** Test the alias by using git co instead of the full git checkout command.

**Activity 29: Viewing Commit Logs with Custom Format**

**Scenario:** You want to view commit history in a customized format to make it easier to read or search.

**Steps:**

1. **View commit history with custom format:**

git log --oneline --author="Author Name" --since="1 week ago"

**Explanation:** Displays the commit history in a one-line format, filtering by author and limiting to commits made in the last week.

1. **View detailed commit history:**

git log --stat

**Explanation:** Shows detailed commit history, including which files were modified and the number of lines added or removed.

**Activity 30: Cherry-Picking a Commit**

**Scenario:** You want to apply a specific commit from one branch onto your current branch without merging the entire branch.

**Steps:**

1. **Identify the commit hash:**

git log

**Explanation:** Find the commit hash you want to cherry-pick (e.g., abc123).

1. **Cherry-pick the commit onto your current branch:**

git cherry-pick abc123

**Explanation:** Applies the changes from the specified commit onto your current branch.

**Activity 31: Configuring Git Global Settings**

**Scenario:** You want to set up your Git environment with your user information so that commits have your name and email associated with them.

**Steps:**

1. **Set your global username:**

git config --global user.name "Your Name"

**Explanation:** Sets your Git username globally, so it's used for all repositories.

1. **Set your global email:**

git config --global user.email "youremail@example.com"

**Explanation:** Sets your email address globally, so it’s associated with your commits.

**Activity 32: Viewing Git Configurations**

**Scenario:** You want to verify the current Git configurations that have been set.

**Steps:**

1. **View your global Git configuration:**

git config --list

**Explanation:** Displays all the configurations currently set, including user name, email, and other settings.

1. **View a specific Git configuration:**

git config user.name

**Explanation:** Displays the configured username.

**Activity 33: Viewing the Difference Between Two Commits**

**Scenario:** You want to compare the changes between two specific commits in your history.

**Steps:**

1. **View the difference between two commits:**

git diff commit1 commit2

**Explanation:** Displays the changes between the two specified commits.

1. **View a diff with commit hash references:**

git diff abc123 def456

**Explanation:** Compares the changes between the commits abc123 and def456.

**Activity 34: Using Git's Blame Feature**

**Scenario:** You want to find out who last modified a specific line in a file and when it happened.

**Steps:**

1. **Blame a file to see line-by-line changes:**

git blame <file>

**Explanation:** Shows each line of a file along with the commit hash and the author of that specific change.

**Activity 35: Using Stash with a Message**

**Scenario:** You are working on something, but need to switch to another branch without losing your work. You want to stash your changes with a descriptive message.

**Steps:**

1. **Stash your changes with a message:**

git stash push -m "WIP on feature-xyz"

**Explanation:** Stashes your current changes with a message so you can easily identify the stash later.

**Activity 36: Creating a Remote Branch**

**Scenario:** You need to create a branch on GitHub (remote repository) and push your local branch to it.

**Steps:**

1. **Create a new branch locally and push it to remote:**

git checkout -b feature-abc

git push origin feature-abc

**Explanation:** This creates a new branch locally and pushes it to the remote GitHub repository.

1. **Verify the branch exists remotely:**

git branch -r

**Explanation:** Lists all remote branches, allowing you to verify that your newly pushed branch exists on the remote.

**Activity 37: Using Git Submodules**

**Scenario:** You want to include another Git repository as a submodule within your project.

**Steps:**

1. **Add a submodule to your repository:**

git submodule add https://github.com/username/submodule-repo.git

**Explanation:** Adds an external repository as a submodule to your project.

git submodule update --init --recursive

**Explanation:** Initializes and updates all submodules in your project.

**Activity 38: Ignoring Files with .gitignore**

**Scenario:** You have files in your project that you don’t want to track with Git (e.g., build artifacts, IDE settings, etc.).

**Steps:**

1. **Create a .gitignore file:**
   * Create a .gitignore file in your project root and add patterns for files you don’t want to track.

\*.log

node\_modules/

\*.class

1. **Explanation:** Tells Git which files or directories to ignore when tracking changes.
2. **Check which files are ignored:**

git status --ignored

**Explanation:** Displays a list of ignored files.

**Activity 39: Reverting a Git Repository to a Previous State**

**Scenario:** You want to undo all changes and revert your repository to a previous commit.

**Steps:**

1. **Reset the repository to a previous commit:**

git reset --hard <commit-hash>

**Explanation:** Resets the working directory and index to the state of the specified commit, discarding all changes after that commit.

1. **Confirm that your repository has been reverted:**

git log

**Explanation:** Verifies that the commit history has been reverted to the specified commit.

**Activity 40: Viewing the Commit Graph**

**Scenario:** You want to visually view the commit history in a graphical format that shows the branching structure.

**Steps:**

1. **View commit history as a graph:**

git log --graph --oneline --decorate --all

**Explanation:** Displays the commit history as a graph with a simplified format, showing all branches and merges in the repository.

**Activity 41: Rewriting Commit History with Interactive Rebase**

**Scenario:** You want to modify your commit history, such as combining multiple commits into one, or editing previous commit messages.

**Steps:**

1. **Start an interactive rebase for the last N commits:**

git rebase -i HEAD~3

**Explanation:** Starts an interactive rebase for the last 3 commits. In the editor, you can choose actions like pick, edit, or squash to modify the commit history.

1. **Combine multiple commits into one (squash commits):**
   * In the rebase editor, change the command from pick to squash for the commits you want to combine.

**Explanation:** This combines multiple commits into one. After saving and closing the editor, Git will prompt you to edit the commit message.

**Activity 42: Working with Git Hooks**

**Scenario:** You want to automate tasks before committing changes, such as running tests or checking for correct formatting.

**Steps:**

1. **Set up a pre-commit hook:**
   * Create a file in .git/hooks/pre-commit and add the commands you want to run before a commit, such as:

#!/bin/sh

npm run lint

* + Make the script executable:

chmod +x .git/hooks/pre-commit

1. **Explanation:** This hook will automatically run npm run lint every time you try to commit, ensuring code quality.