|  |  |  |
| --- | --- | --- |
| SCHOOL OF INFORMATION AND TECHNOLOGY | | |
| NAME: Zhaun Gabrielle F. Pasion | DATE PERFORMED: 11/21/24 |  |
| Section: Idc2 | DATE SUBMITTED: 11/21/24 |

# SYSADM1 – Git Basics

Answer the following research questions about Git, GitLab desktop and GitHub.

1. What is Git, and why is it important in software development?

Git is a version control system that tracks changes to files, allowing developers to maintain a history of their work and collaborate effectively. It is crucial in software development because it enables multiple developers to work on the same project simultaneously, making it easier to track revisions, manage different versions of code, and merge changes without losing work. Git helps maintain a clear record of changes, supports branching and merging, and allows developers to revert to earlier versions when necessary, making collaboration smoother and more efficient.

1. How does Git track changes in a project?

Git keeps track changes in project by maintaining a hidden directory called .git in the root of the project. This directory is created when the user initialize a Git repository using the git init command. The .git directory contains all the necessary information that Git uses to track the changes in the project, including:

* **Commit history**: Git stores snapshots of the project at each commit, allowing usersto track and revert to previous versions.
* **Staging area**: Git tracks which changes are staged (ready to be committed) and which are not.
* **Configuration files**: Git saves configuration settings specific to the repository

Once the repository is initialized, Git begins tracking files in the working directory. Files are categorized into **tracked** (files that Git knows about) and **untracked** (new files Git hasn’t yet seen). As the user make changes to tracked files, Git detects modifications and stores these changes in the .git directory, allowing the user to commit, stage, and revert changes as needed.

1. What is the difference between a local repository and a remote repository in Git?

The main difference between a **local repository** and a **remote repository** in Git lies in their location and purpose. A **local repository** is stored on an individual’s local machine and contains a complete copy of the project’s files and version history. It allows the developer to work offline and commit changes locally. The changes made in a local repository can later be synchronized with a remote repository by pushing or pulling updates. On the other hand, a **remote repository** is hosted on a remote server, such as GitHub, GitLab, or an off-site server, and serves as a central location for collaboration. Multiple team members can access the remote repository to push their changes and pull updates from others. It provides a shared space where the project’s files and version history are stored and accessible to all collaborators, ensuring the project stays up to date across different team members and locations. In summary, the local repository is for individual use, while the remote repository is for team collaboration and centralized storage.

1. What are the basic Git commands?

**git init**: Initializes a new Git repository in your project. This is the first step to start using Git for version control.

**git add**: Moves changes from the working directory to the staging area. This allows you to prepare a snapshot before committing the changes to the repository.

**git commit**: Takes the staged changes and commits them to the project history. It creates a snapshot of the changes and adds them to the version history.

**git status**: Displays the state of the working directory and staging area, showing which files are tracked, untracked, modified, or staged for the next commit.

**git log**: Lets you view the project’s commit history. It displays previous revisions, along with details about each commit, such as author, date, and commit message.

**git push**: Pushes local changes to a remote repository. It sends commits from your local branch to a remote repository, making them available to others.

**git pull**: Combines git fetch and git merge to download changes from a remote repository and merge them into your local repository.

**git fetch**: Downloads branches and their associated commits from a remote repository, but does not automatically merge them into your local repository. It allows you to inspect changes before merging.

**git clone**: Creates a copy of an existing Git repository. It’s typically used to get a working copy of a remote repository.

**git branch**: Manages branches in the repository. You can use it to create, list, or delete branches, enabling you to work on different features or versions of your project.

**git checkout**: Allows you to switch between branches or check out previous versions of files or commits.

**git reset**: Undoes changes in the working directory or staging area, allowing you to remove changes before committing them.

**git revert**: Reverts a commit, undoing the changes made in a previous commit. It creates a new commit that removes the effects of the faulty commit.

**git merge**: Integrates changes from different branches, combining the work done in parallel.

1. How do you check the status of a Git repository?

To check the status of a Git repository, the git status command is used. This command provides an overview of the current state of the working directory and staging area, displaying which files have been modified, which are staged for commit, and which are untracked. It also shows whether the local branch is up to date with the remote repository. By running git status, it's possible to quickly assess what changes need to be added, committed, or pushed.

1. What is the purpose of branches in Git, and how do you create and switch between them?

In Git, branches serve as independent lines of development, allowing multiple features or bug fixes to be worked on simultaneously without affecting the main codebase. Each branch acts as a separate workspace, making it easy to manage different tasks, experiment with new ideas, or address urgent issues without disrupting the primary project.

To create and switch between branches in Git, the git switch command is used. To create a new branch and switch to it, the command is git switch -c <new-branch-name>, where <new-branch-name> is the name of the new branch. To switch to an existing branch, the command is git switch <branch-name>, replacing <branch-name> with the desired branch name. If switching to a branch from a remote repository, the git switch -c <local-branch-name> origin/<remote-branch-name> command can be used to create a local version of the remote branch. For switching back to the previous branch, Git offers a shorthand command git switch -.

1. What are GitLab Desktop and GitHub, and how are they different from Git?

GitLab and GitHub are web-based Git repositories that enable developers to store, manage, and collaborate on code. While they are both built on Git, they are distinct platforms with unique features and functionalities.

**Git** is an open-source version control system (VCS) that allows developers to track changes in code, collaborate on projects, and maintain multiple versions of code simultaneously. Git manages the local version control, allowing users to save "snapshots" of their work, revert to previous versions, and work on various branches.

**GitHub** is a cloud-based platform built around Git that facilitates collaboration, version control, and project management. It focuses on building a strong community of developers and is particularly known for its massive user base and social features like repositories, pull requests, and issue tracking. GitHub is more DIY in its approach, requiring users to integrate third-party tools for continuous integration (CI) and continuous delivery (CD).

**GitLab** is similar to GitHub but offers a more comprehensive set of built-in features, particularly for DevOps and CI/CD processes. Unlike GitHub, GitLab has native CI/CD pipelines, issue tracking, and project management tools, making it a more all-in-one platform for developers looking for integrated workflows. GitLab is also open-source, allowing greater flexibility for customization and deployment.

In summary, **Git** is the core version control system, while **GitHub** and **GitLab** are platforms that extend Git’s functionality with additional tools for collaboration and project management. The key difference between GitHub and GitLab lies in their feature sets, with GitLab offering more built-in DevOps capabilities and GitHub focusing more on community-driven collaboration.

1. How do you connect a local Git repository to a GitLab or GitHub repository?

To connect a local Git repository to GitHub or GitLab:

**Step 1: Create a repository** on GitHub or GitLab:

* + **GitHub**: Create a new repository.
  + **GitLab**: Create a new project.

**Step 2: Initialize Git locally** if the project isn't already a Git repository:



**Step 3: Add and commit files** (if not already done):



**Step 4: Add the remote repository URL**:

**GitHub**:

****

**GitLab**:



**Step 5: Push the code**:



**Step 6: Verify** the files are uploaded to GitHub or GitLab.

1. What are the steps to collaborate with others using GitLab or GitHub?

Step 1: **Create or Fork a Repository**

* On GitHub: Create a new repository or fork an existing one.
* On GitLab: Create a new project or fork an existing repository.

Step 2: **Clone the Repository**

* Copy the repository URL from GitHub or GitLab.
* Clone it to the local machine using the Git software.

Step 3: **Create a Branch**

* Create a new branch to work on, ensuring changes are made separately from the main branch.

Step 4: **Make Changes Locally**

* Edit the files, add new features, or fix bugs in the project.
* Save and commit the changes locally.

Step 5: **Push Changes to Remote**

* Push the changes from the local branch to the remote repository on GitHub or GitLab.

Step 6: **Open a Pull/Merge Request**

* On GitHub: Open a Pull Request (PR) to merge the branch into the main branch.
* On GitLab: Open a Merge Request (MR) to merge your branch with the main branch.

Step 7: **Review and Discuss**

* Collaborators will review the PR/MR, discuss the changes, and suggest any modifications.

Step 8: **Merge Changes**

* After review and approval, the repository owner or collaborators with the appropriate permissions will merge the changes into the main branch.

Step 9: **Pull the Latest Changes**

* Regularly pull the latest changes from the main branch to stay up-to-date with the project.

1. How do you resolve merge conflicts in Git?

Step 1: The easiest way to resolve a conflicted file is to open it and make any necessary changes.

Step 2: After editing the file, we can use the git add a command to stage the new merged content.

Step 3: The final step is to create a new commit with the help of the git commit command.

Step 4: Git will create a new merge commit to finalize the merge.

Also using Git commands to resolve conflicts using the following codes.

**git log --merge**: Produces a list of commits causing the conflict.

**git diff**: Identifies the differences between the state of repositories or files.

**git checkout**: Undoes changes made to a file or switches branches.

**git reset --mixed**: Undoes changes to the working directory and staging area.

**git merge --abort**: Exits the merge process and returns to the state before the merge started.

**git reset**: Resets the conflicted files to their original state during a merge conflict.

1. What is a pull request, and why is it used in GitHub?

A **pull request (PR)** in GitHub is a feature that allows a developer to propose changes to a project by submitting their branch for review before merging it into the main codebase. It facilitates collaboration by enabling team members to review, discuss, and suggest improvements to the proposed changes. Pull requests also serve as a way to track and manage changes, ensuring that code is integrated in an organized and controlled manner while minimizing conflicts. This process is essential for maintaining code quality and ensuring that all changes are thoroughly reviewed before being incorporated into the project.

1. What are some best practices for writing commit messages?

* **Be clear and concise**: The commit message should clearly describe the changes made and the reason for them. Avoid vague messages like "Fix stuff" or "Update project."
* **Use an imperative tone**: Write commit messages in the imperative mood (e.g., "Fix bug" rather than "Fixed bug"), as it aligns with Git's default commit messages.
* **Keep it short and focused**: The message should ideally be no more than 50 characters for the summary line. If necessary, provide more detail in the body of the commit.
* **Explain the "why," not just the "what"**: Describe why a change was made, not just what was changed. This helps others (and your future self) understand the reasoning behind the change.
* **Follow a consistent format**: Consider using a standardized format, such as the Conventional Commits style, which categorizes changes (e.g., "feat:", "fix:", "docs:") and includes a brief description.
* **Group related changes**: A commit should represent a single, logical change. Avoid mixing unrelated changes in one commit to make it easier to understand and review.
* **Test before committing**: Ensure the code is tested and verified before committing to avoid breaking the build for others.
* **Use branches effectively**: Work on feature-specific or bug-specific branches and ensure that commit messages accurately reflect the purpose of the branch.

**References:**

Desktop, N. (2023, July 24). What is Git and Why Should You Use it? *What is Git and Why Should You Use it?* https://www.nobledesktop.com/learn/git/what-is-git

*Git - Recording changes to the repository*. (n.d.). https://git-scm.com/book/ms/v2/Git-Basics-Recording-Changes-to-the-Repository#:~:text=As%20you%20edit%20files%2C%20Git,changes%2C%20and%20the%20cycle%20repeats.

Designveloper. (2024, October 20). What is Git and how does Git track our project files? *Medium*. https://dsvgroup.medium.com/what-is-git-and-how-does-git-track-our-project-files-8f38c93f3bc2

*Repositories | Git tutorial | Nulab*. (n.d.). Nulab. https://nulab.com/learn/software-development/git-tutorial/git-basics/repositories/#:~:text=A%20remote%20repository%20is%20hosted,machine%20for%20an%20individual%20user.

Atlassian. (n.d.). *Basic Git Commands | Atlassian Git Tutorial*. https://www.atlassian.com/git/glossary#commands

Aubry, F. (2024, September 11). *Git Switch Branch: A Guide With Practical Examples*. Datacamp. https://www.datacamp.com/tutorial/git-switch-branch

*Git, GitHub, and GitLab: What’s the difference? | BairesDev*. (2022, October 26). BairesDev. https://www.bairesdev.com/blog/git-github-and-gitlab-whats-the-difference/

McKenzie, C. (2024, March 1). How to git push an existing project to GitLab. *How to git push an existing project to GitLab*. https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/How-to-add-and-push-an-existing-project-to-GitLab

*Software Collaboration with GitHub and GitLab - Research Computing Documentation*. (n.d.). https://docs.rc.uab.edu/workflow\_solutions/git\_collaboration/

Afreen, S. (2024, July 15). *How to resolve merge conflicts in Git?* Simplilearn.com. https://www.simplilearn.com/tutorials/git-tutorial/merge-conflicts-in-git

Manzoor, S. (2024, July 19). *Good Commit ✔ VS. Bad Commit ❌: Best Practices for Git*. DEV Community. https://dev.to/sheraz4194/good-commit-vs-bad-commit-best-practices-for-git-1plc#:~:text=Best%20Practices%20for%20Good%20Commits&text=Never%20push%20unrelated%20changes%20in,%2C%20bug%20fixes%2C%20and%20experiments.