

Module code: SWDVC301

Module Title: CONDUCT VERSION CONTROL

Learning unit 1: Setup repository

1.1 Git introduction based on version control

1.1.1 Definition of general key terms

Version Control

Version control is a system that allows developers to track and manage changes made to a project's source code over time. It helps teams collaborate on code, keep track of changes, and maintain a history of all modifications.

Git

Git is a popular version control system that allows developers to track and manage changes made to a project's source code over time. It was created by Linus Torvalds in 2005 and is widely used in the software development industry. Git provides features like branching and merging, which make it easier for developers to work on multiple versions of the code simultaneously.

GitHub

GitHub is a web-based platform that provides hosting for Git repositories. It allows developers to store their Git repositories in the cloud and provides a range of collaboration tools like pull requests, issues, and code reviews. GitHub also provides features like project management tools, wikis, and access controls to help teams work more effectively.

Terminal

A terminal is a text-based interface used to interact with a computer's operating system. It allows users to execute commands and run scripts directly from the command line. In the context of version control, the terminal is often used to interact with Git and GitHub repositories using command-line tools like git, git-lfs, and git-annex.

1.1.2. Introduction to version control

Version control is a system that allows developers to track and manage changes made to a project's source code over time.

With version control, developers can work on multiple versions of the code simultaneously and can easily revert back to an earlier version if needed.

It is a critical tool in software development because it helps teams collaborate on code, keep track of changes, and maintain a history of all modifications.

With version control, developers can work on multiple versions of the code simultaneously, and can easily revert back to an earlier version if needed. This makes it easier to experiment with new ideas, develop new features, and fix bugs without fear of losing previous work.

In addition to **keeping track of code changes**, version control also provides a range of other benefits, including:

Collaboration: With version control, developers can work on the same codebase simultaneously, and merge their changes together seamlessly. This enables teams to work more effectively and efficiently, and ensures that everyone is working on the latest version of the code.

Backup and Disaster Recovery: Version control systems store all versions of the code, making it easy to recover from accidental deletions, hardware failures, or other disasters.

Code Reviews: Version control systems provide tools for code review, allowing teams to collaborate on code changes, and ensuring that code meets the team's standards for quality and security.

Continuous Integration and Deployment: Version control systems can be integrated with other tools like continuous integration and deployment systems to automate the software development process and ensure that code changes are tested and deployed quickly and reliably.

Overall, version control is a critical tool for modern software development, enabling teams to work collaboratively, maintain code quality, and deliver high-quality software quickly and efficiently.

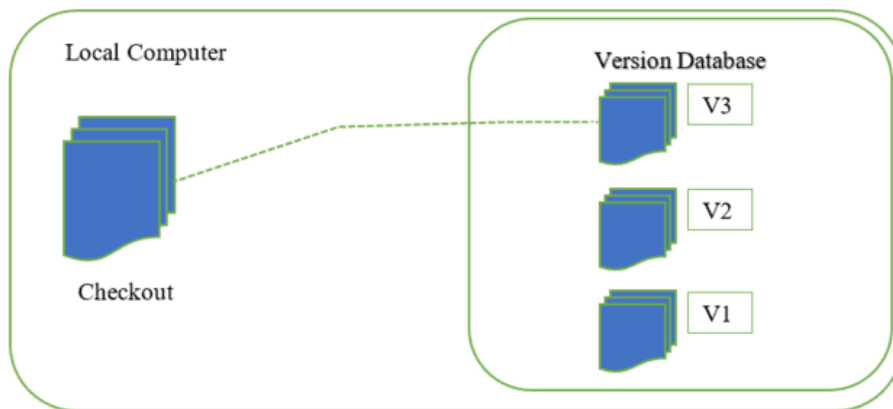
✚ Types of version control

Local Version Control

Local version control is the simplest form of version control, where changes made to a project's source code are tracked and managed on a developer's local machine. In this approach, developers create a copy of the code repository on their local machine and make changes to it. The version control system keeps track of changes made to the local repository, and developers can revert to an earlier version of the code if needed.

A local version control system is a local database located on your local computer, in which every file change is stored as a patch. Every patch set contains only the changes made to the file since its last version. In order to see what the file looked like at any given moment, it is necessary to add up all the relevant patches to the file in order until that given moment.

The main problem with this is that everything is stored locally. If anything were to happen to the local database, all the patches would be lost. If anything were to happen to a single version, all the changes made after that version would be lost. Also, collaborating with other developers or a team is very hard or nearly impossible.



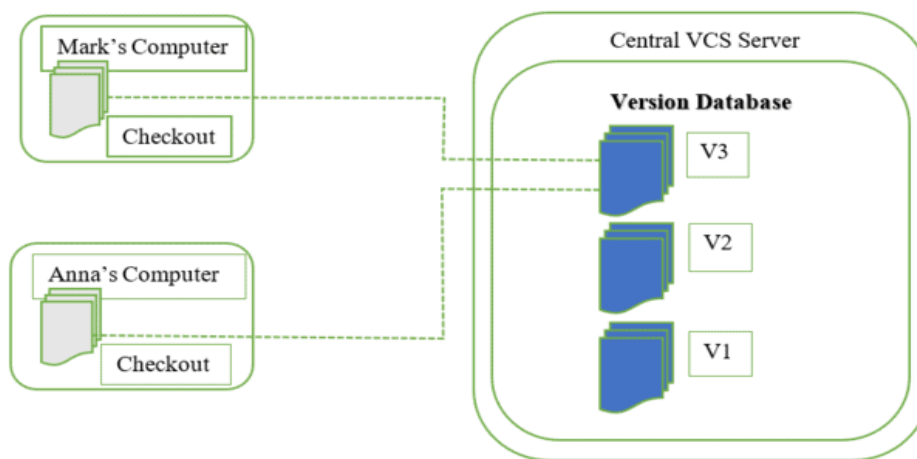
The disadvantage of local version control is that it does not provide collaboration features that enable multiple developers to work on the same codebase simultaneously. Moreover, local version control systems do not provide any backup or disaster recovery mechanism.

Centralized Version Control System

Centralized version control system (CVCS) is a version control system in which developers work on a shared repository hosted on a central server. In this approach, developers check out a copy of the code from the central server, make changes to it, and then commit their changes back to the central server.

The advantage of CVCS is that it provides collaboration features that enable multiple developers to work on the same codebase simultaneously. Moreover, it provides backup and disaster recovery mechanisms, as the central server can be backed up regularly.

The disadvantage of CVCS is that it is vulnerable to a single point of failure, as the central server can be a bottleneck for the entire team's workflow. Moreover, if the central server goes down, developers cannot access the code repository or collaborate with other team members until the server is restored.



A centralized version control system has a single server that contains all the file versions. This enables multiple clients to simultaneously access files on the server, pull them to their local computer or push them onto the server from their local computer. This way, everyone usually knows what everyone else on the project is doing. Administrators have control over who can do what.

This allows for easy collaboration with other developers or a team.

The biggest issue with this structure is that everything is stored on the centralized server. If something were to happen to that server, nobody can save their versioned changes, pull files or collaborate at all. Similar to Local Version Control, if the central database became corrupted, and backups haven't been kept, you lose the entire history of the project except whatever single snapshots people happen to have on their local machines.

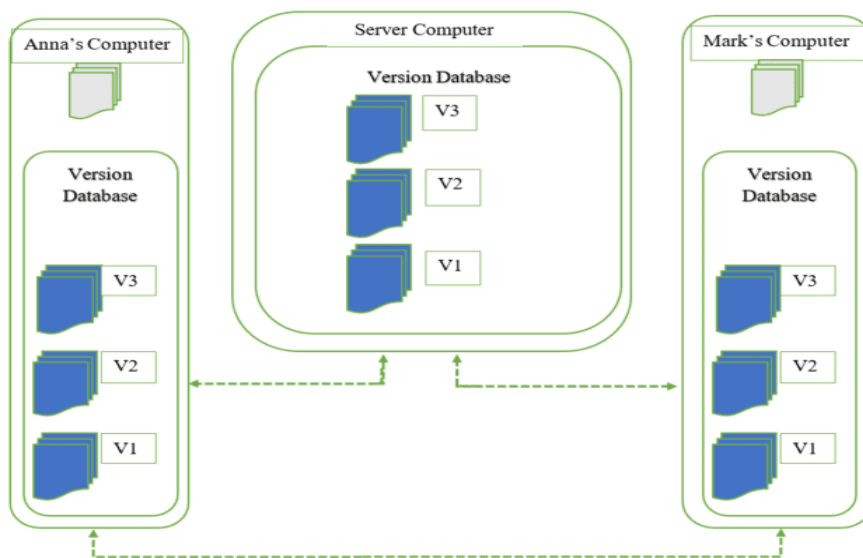
The most well-known examples of centralized version control systems are Microsoft Team Foundation Server (TFS) and SVN.

Distributed Version Control

Distributed version control system (DVCS) is a version control system that allows developers to create local repositories that mirror the remote repository's contents. In this approach, each developer has a complete copy of the code repository on their local machine, and they can work on it independently. Developers can commit their changes to their local repository and can also pull changes from other team members' local repositories.

The advantage of DVCS is that it provides robust collaboration features that enable developers to work on the same codebase simultaneously and independently. DVCS also provides a backup and disaster recovery mechanism, as each developer has a complete copy of the repository on their local machine.

The disadvantage of DVCS is that it can be complex to manage, and developers need to be careful when merging changes from different branches to avoid conflicts. Moreover, DVCS requires more disk space than other version control systems, as each developer has a complete copy of the repository on their local machine.



With distributed version control systems, clients don't just check out the latest snapshot of the files from the server, they fully mirror the repository, including its full history. Thus, everyone collaborating on a project owns a local copy of the whole project, i.e. owns their own local database with their own complete history. With this model, if the server becomes unavailable or dies, any of the client repositories can send a copy of the project's version to any other client or

back onto the server when it becomes available. It is enough that one client contains a correct copy which can then easily be further distributed.

Git is the most well-known example of distributed version control systems.

Well Known version control system

Git

Git is a distributed version control system that was created by Linus Torvalds, the creator of the Linux operating system. It is widely used in the software development industry and is known for its speed, efficiency, and flexibility. Git is designed to handle everything from small personal projects to large enterprise software development projects.

Git uses a branching model, which enables developers to create multiple branches of the codebase and work on them independently. It also provides tools for merging changes from different branches and resolving conflicts. Git is open-source software and is available for free.

CVS (Concurrent Version System)

CVS is a centralized version control system that was developed in the 1980s. It was one of the first version control systems to gain widespread adoption in the software development industry. CVS allows multiple developers to work on the same codebase simultaneously by checking out a copy of the code from a central repository.

CVS provides tools for managing multiple versions of files, merging changes from different developers, and tracking changes over time. However, CVS is an older system that lacks some of the advanced features of newer version control systems.

Mercurial

Mercurial is a distributed version control system that is similar to Git in many respects. It was created by Matt Mackall in 2005 and is known for its ease of use and flexibility. Like Git, Mercurial uses a branching model, which enables developers to work on different versions of the codebase simultaneously.

Mercurial provides tools for merging changes from different branches, resolving conflicts, and tracking changes over time. It is open-source software and is available for free.

SVN (Subversion)

SVN, or Subversion, is a centralized version control system that was created in 2000. It was designed to be a successor to CVS and provides similar

functionality. SVN allows multiple developers to work on the same codebase simultaneously by checking out a copy of the code from a central repository.

SVN provides tools for managing multiple versions of files, merging changes from different developers, and tracking changes over time. However, like CVS, SVN is an older system that lacks some of the advanced features of newer version control systems.

Overall, each of these version control systems has its own strengths and weaknesses. Git and Mercurial are both popular distributed version control systems, while CVS and SVN are centralized version control systems that have been widely used in the software development industry for many years.

Benefits of version control

Version control is a crucial tool in software development that allows developers to track changes made to their codebase over time. Here are some of the benefits of using version control:

History Tracking: Version control systems keep a detailed history of all changes made to a codebase, including who made the changes, what changes were made, and when they were made. This makes it easy to track down issues and revert to previous versions if necessary.

Collaboration: Version control systems allow multiple developers to work on the same codebase at the same time, without fear of overwriting each other's changes. This enables developers to work together more efficiently and effectively.

Branching and Merging: Version control systems allow developers to create separate branches of a codebase, which can be used for testing, bug fixing, or implementing new features. When changes made in one branch are ready to be incorporated into the main codebase, they can be merged back in.

Accountability: Version control systems help to create a culture of accountability in software development teams. Each change made to the codebase is tracked, and it's clear who made the change and when. This can help to improve code quality and reduce errors.

Backup and Restore: Version control systems provide a reliable backup of a codebase. If a local copy of the codebase is lost, it can be restored from the version control system.

Overall, version control is a vital tool for any software development team. It helps to improve collaboration, accountability, and code quality, and it provides a reliable way to track changes made to a codebase over time.

Application of version control

Version control has many applications in software development, and it is a critical tool for managing codebases, especially in collaborative development environments. Here are some of the common applications of version control:

Source Code Management: Version control is used to manage source code and related assets, such as configuration files, build scripts, and documentation. This includes tracking changes to code, creating and merging branches, and managing code releases.

Bug Tracking and Issue Management: Version control systems can be integrated with bug tracking and issue management systems to keep track of reported issues, track their progress, and monitor fixes.

Continuous Integration and Deployment: Version control is often used as part of a continuous integration and deployment process, where changes are automatically built, tested, and deployed to production systems.

Documentation Management: Version control systems can also be used to manage documentation and other non-code assets, such as design documents, technical specifications, and user manuals.

Research Collaboration: Version control systems can be used in research environments to manage datasets, code, and research notes, allowing researchers to track changes, collaborate, and reproduce experiments.

Content Management: Version control can also be used to manage content, such as web pages, blog posts, and marketing materials. This allows content creators to track changes and manage multiple versions of content.

Overall, version control is a versatile tool that can be used in many different contexts to manage a wide range of assets. Its flexibility and ability to track changes over time make it an essential tool for software development, research collaboration, and content management.

1.1.3. Description of git

Git is a popular version control system that was developed by Linus Torvalds, the creator of the Linux operating system. It is a distributed version control system, which means that each user has a complete copy of the codebase, rather than relying on a centralized server. Git was designed to be fast, flexible, and easy to use, and it has become one of the most widely used version control systems in software development.

Git is based on the concept of a repository, which is a directory or folder that contains all the files and code for a project. When a user makes changes to the code, Git tracks those changes and creates a snapshot of the code at that point in time. These snapshots are stored in the repository, along with metadata that tracks who made the changes, when they were made, and what changes were made.

One of the key features of Git is branching, which allows users to create a separate copy of the codebase for testing or development purposes. This makes it easy to experiment with new features or fixes without affecting the main codebase. Git also has powerful merging capabilities, which allow users to merge changes from one branch to another or to the main codebase.

Git has a command-line interface that can be used to perform all the basic version control tasks, such as creating a new repository, adding files to the repository, making changes, committing changes, creating and merging branches, and pushing changes to a remote repository. Git also has a graphical user interface (GUI) that makes it easy to perform these tasks without using the command line.

One of the key advantages of Git is its distributed nature, which means that users can work offline and still have access to a complete copy of the codebase. Git also makes it easy to collaborate with other developers, as changes can be pushed to a remote repository and then pulled by other users.

In summary, Git is a powerful and flexible version control system that is widely used in software development. Its distributed nature, branching and merging capabilities, and easy-to-use interface make it a popular choice for teams of all sizes

Git Basic concept

Git is a distributed version control system that allows developers to track changes to their code over time. Here are some of the basic concepts of Git:

Repository: A repository is a collection of files and code for a project. Git stores all the changes made to the codebase in the repository, along with metadata that tracks who made the changes, when they were made, and what changes were made.

Commit: A commit is a snapshot of the codebase at a particular point in time. When a developer makes changes to the code, they create a new commit, which is then added to the repository. Each commit has a unique identifier, which can be used to track changes over time.

Branch: A branch is a separate copy of the codebase that can be used for testing or development purposes. Developers can create a new branch from the main codebase, make changes to the branch, and then merge those changes back into the main codebase when they are ready.

Merge: Merging is the process of combining changes from one branch into another branch or the main codebase. Git has powerful merging capabilities, which make it easy to merge changes from different branches while minimizing conflicts.

Remote: A remote is a copy of the repository that is stored on a remote server, such as GitHub or GitLab. Developers can push changes to the remote repository, and then pull those changes onto their local machine.

Pull: Pulling is the process of downloading changes from a remote repository and merging them into the local codebase. This allows developers to collaborate with other developers and keep their codebase up-to-date with the latest changes.

Overall, Git is a powerful and flexible version control system that allows developers to track changes to their code, collaborate with other developers, and manage complex codebases with ease. Understanding the basic concepts of Git is essential for using it effectively in software development projects.

Git architecture

Git is a distributed version control system that is designed to be fast, flexible, and easy to use. Its architecture consists of three main components:

Working Directory: The working directory is the directory on a developer's local machine where they edit and modify the files in the codebase. It contains the current version of the codebase, along with any changes that the developer has made.

Staging Area: The staging area is an intermediate area where developers can review and prepare changes before committing them to the repository. Developers can choose which changes to include in a commit by staging them in the staging area.

Repository: The repository is the central database that stores all the changes made to the codebase over time. Each commit is stored in the repository, along with metadata that tracks who made the changes, when they were made, and what changes were made.

Git's architecture is based on the concept of snapshots, which are taken of the codebase at different points in time. Each snapshot is stored as a commit, along with metadata that describes the changes made to the code. When a developer makes changes to the code, Git creates a new snapshot of the code and stores it in the repository.

Git's distributed architecture means that each developer has a complete copy of the repository on their local machine, which makes it easy to work offline or collaborate with other developers. Developers can make changes to their local copy of the repository, and then push those changes to a remote repository, such as GitHub or GitLab.

Overall, Git's architecture is designed to be fast, flexible, and easy to use, and it provides developers with a powerful tool for managing complex codebases and collaborating with other developers

Git workflow

Description of Git workflow

Git is a version control system used by software developers to track changes in their codebase. It allows teams to collaborate on code and track changes made to the code over time. Git provides a variety of features to support workflow, including branching, merging, and rebasing. Here is a general overview of a typical Git workflow:

1. **Create a new branch:** Before making any changes to the code, it's generally a good idea to create a new branch. A branch is a copy of the codebase at a specific point in time. By creating a branch, you can work on changes without affecting the main codebase.
2. **Make changes:** Once you've created a new branch, you can make changes to the code. You can add, remove, or modify code as necessary.
3. **Commit changes:** Once you've made changes, you'll need to commit them to your branch. A commit is a snapshot of the changes you've made to the code. Each commit should have a descriptive message that explains the changes you've made.
4. **Push changes:** Once you've committed your changes, you'll need to push them to a remote repository. A remote repository is a shared location where all members of a team can access the code. By pushing your changes, you make them available to other team members.
5. **Create a pull request:** If you're working on a team, it's a good idea to create a pull request before merging your changes into the main codebase. A pull request is a request to merge your changes into the main codebase. It allows other team members to review your changes and provide feedback.

6. **Review changes:** Once you've created a pull request, other team members can review your changes. They can leave comments and suggest changes if necessary.
7. **Merge changes:** Once your changes have been reviewed and approved, you can merge them into the main codebase. Merging combines the changes in your branch with the main codebase.
8. **Update local codebase:** After merging your changes, you'll need to update your local codebase to reflect the changes in the main codebase. You can do this by pulling the latest changes from the remote repository.

This is a basic overview of a Git workflow, and there are many variations depending on the needs of your team. However, this general process should give you an idea of how Git works and how it can be used to collaborate on code.

Types of Git workflows

Git is a popular version control system used by software developers to track changes in code and collaborate on projects. There are several different Git workflows that teams can use to manage their code changes and collaborate effectively. Here are some of the most common Git workflows:

Centralized Workflow:

This workflow is the simplest and most commonly used in many organizations. It involves a central repository where developers push their changes and pull changes from. The workflow is centralized because only one branch, usually the master branch, is used to store all changes.

Feature Branch Workflow:

This workflow is ideal for larger teams where many developers are working on the same codebase. Each feature or task is assigned a separate branch, which developers use to work on that specific feature. Changes made to the feature branch are then merged back into the main branch after the feature is completed.

Gitflow Workflow:

Gitflow is a branching model that provides a more formal approach to managing code changes. It involves two long-lived branches - a develop branch and a master branch. Developers create feature branches from the develop branch and merge their changes back into the develop branch. When the code in the develop branch is stable, it is merged into the master branch, which is used to create releases.

Forking Workflow:

This workflow is commonly used in open-source projects. Instead of having one central repository, each developer creates a fork of the main repository. Developers then create feature branches in their fork, make changes, and create pull requests to merge their changes back into the main repository.

Pull Request Workflow:

This workflow is a variation of the feature branch workflow. It involves creating a separate branch for each feature or task, making changes, and then creating a pull request to merge the changes back into the main branch. The pull request allows other developers to review the changes before they are merged, ensuring that the code is of high quality.

These are just a few examples of Git workflows that teams can use to manage their code changes. The right workflow depends on the team size, the complexity of the project, and the development process.

Initialization of Git

Step 1: Install Git

First, you need to install Git on your computer. You can download Git from the official Git website (<https://git-scm.com/downloads>) and install it according to the instructions provided for your operating system.

Step 2: Create a Git repository

Once Git is installed on your computer, you need to create a new Git repository. Navigate to the directory where you want to create your repository, and use the following command in your terminal or command prompt:

```
git init
```

This command initializes a new Git repository in the current directory.

To navigate to the directory where you want to create your Git repository, you can use the command line interface (CLI) of your operating system. Here are the steps:

Open your command prompt (Windows) or terminal (Mac/Linux).

Use the `cd` command (which stands for "change directory") followed by the path of the directory you want to navigate to. For example, if you want to navigate to a directory named "my-project" located in your home directory, you can use the following command:

```
cd ~/my-project
```

This command navigates to the "my-project" directory in your home directory on a Unix-based system. If you're on a Windows system, you can use the following command instead:

```
cd C:\Users\YourUsername\my-project
```

This command navigates to the "my-project" directory located in the "YourUsername" folder on your C: drive.

Once you're in the directory where you want to create your Git repository, you can use the `git init` command to initialize a new Git repository in that directory.

Step 3: **Add files to the repository**

Next, you need to add files to your repository. You can do this by creating new files in the repository directory, or by copying existing files into the repository directory.

Once you have added files to your repository, you need to tell Git to track them. To do this, use the following command:

```
git add <file-name>
```

Replace `<file-name>` with the name of the file you want to track. You can also use `git add .` to track all files in the repository directory.

Step 4: **Make a commit**

After adding files to the repository, you need to make a commit to save the changes. A commit is a snapshot of the changes you have made to your files, along with a message describing the changes.

To make a commit, use the following command:

```
git commit -m "Commit message"
```

Replace "Commit message" with a short description of the changes you have made.

Step 5: **Connect to a remote repository (optional)**

If you want to collaborate with others on your project, you may want to connect your local repository to a remote repository. A remote repository is a Git repository hosted on a server, such as GitHub or Bitbucket.

To connect to a remote repository, use the following command:

```
git remote add origin <remote-url>
```

Replace <remote-url> with the URL of the remote repository. You can find this URL on the website of the remote repository provider.

Step 6: **Push changes to the remote repository (optional)**

If you have connected your local repository to a remote repository, you can push your changes to the remote repository to share them with others.

To push your changes to the remote repository, use the following command:

```
git push origin <branch-name>
```

Replace <branch-name> with the name of the branch you want to push to the remote repository. By default, the branch name is master.

And that's it! You have successfully initialized Git and created a new repository. From here, you can use Git to track changes to your files, collaborate with others on your project, and more.

Terminal Basic commands in Git

git init: Initializes a new Git repository in the current directory.

git clone: Copies an existing Git repository to your local machine.

git add: Adds files or changes to the staging area.

git commit: Commits changes to the local repository with a message describing the changes made.

git push: Pushes committed changes to a remote repository, usually on a Git hosting service like GitHub or GitLab.

git pull: Pulls changes from a remote repository to your local repository.

git status: Shows the current status of the repository, including any changes that are staged or not staged.

git log: Shows a history of commits in the local repository.

git branch: Shows a list of branches in the repository or creates a new branch.

git merge: Merges changes from one branch into another branch.

Installation of Git Setup

To install Git on your machine, you can follow these steps:

For Windows:

Download the latest Git for Windows installer from the Git website: <https://git-scm.com/download/win>

Double-click the downloaded file to start the installation process.

Follow the prompts in the installer. The default options should be sufficient for most users.

Once the installation is complete, open the command prompt or Git Bash and run **git --version** to verify that Git is installed and working correctly.

For Mac:

If you have Homebrew installed, you can install Git by running **brew install git** in the terminal. If you don't have Homebrew, you can download the Git for Mac installer from the Git website: <https://git-scm.com/download/mac>

Double-click the downloaded file to start the installation process.

Follow the prompts in the installer. The default options should be sufficient for most users.

Once the installation is complete, open the terminal and run **git --version** to verify that Git is installed and working correctly.

For Linux:

You can install Git using your distribution's package manager. For example, on Ubuntu, you can run **sudo apt-get install git** in the terminal.

Alternatively, you can download the Git source code from the Git website: <https://git-scm.com/download/linux>

Extract the downloaded archive to a directory of your choice.

In a terminal window, navigate to the directory where you extracted the Git source code and run the command **make**.

Once the compilation process is complete, run **sudo make install** to install Git on your system.

Run **git --version** in the terminal to verify that Git is installed and working correctly.

Configure Git

Git **init** command:

The git init command is used to create a new Git repository in your project directory. Follow these steps to use it:

```
$ cd your_project_directory # Navigate to your project directory
```

```
$ git init # Initialize a new Git repository
```

This will create a new .git directory in your project, which is where Git will store all the necessary repository data.

Git **config** command:

The git config command is used to configure Git settings. There are two levels of configuration: global and local. Global settings apply to your user account across all repositories, while local settings are specific to a particular repository. Here's how to use the git config command:

To set global configuration:

```
$ git config --global user.name "Your Name" # Set your name
```

```
$ git config --global user.email "your@email.com" # Set your email
```

To set local configuration (specific to a repository), navigate to your project directory and run the same commands without the --global flag. These settings will override the global settings for that repository.

Git version command:

The git **--version** command is used to check the installed version of Git on your system. Run the following command to view the Git version:

```
$ git --version
```

This will display the installed Git version information, such as "git version 2.30.1".

Configure .git ignore file

The .gitignore file is a plain text file used by the version control system Git to specify intentionally untracked files that Git should ignore. It is typically placed in the root directory of a Git repository.

The purpose of the .gitignore file is to tell Git which files and directories should not be tracked or included in the version control system. This is useful for excluding files that are generated by the build process, contain sensitive

information (such as passwords or API keys), or are simply not relevant to the development or collaboration process.

The `.gitignore` file uses simple pattern matching rules to specify the files and directories that should be ignored. The patterns can include wildcards and specific file or directory names. For example, `*.log` would match all files with the `.log` extension, and `secret.txt` would match a file named `secret.txt`.

Here are a few key points about `.gitignore` files:

Rules: Each line in the `.gitignore` file represents a rule. Rules can be specific to files, directories, or patterns.

Wildcards: Wildcards, such as `*`, `?`, and `[]`, can be used to represent patterns in file or directory names. For example, `.txt` matches all files with the `.txt` extension.

Negation: A rule starting with an exclamation mark (`!`) negates the pattern and tells Git to include the matched files, even if they would have been ignored by previous patterns.

Comments: Lines starting with a hash symbol (`#`) are treated as comments and are ignored by Git. They can be used to add explanatory or descriptive text.

Recursive Matching: By default, the `.gitignore` file applies to the current directory and its subdirectories. To exclude specific files or directories only in the current directory, you can use a leading slash (`/`) before the pattern.

Multiple `.gitignore` Files: Git allows multiple `.gitignore` files within a repository. Rules in these files are cumulative, with more specific rules overriding more general rules.

It's important to note that once a file or directory is already tracked by Git, adding it to the `.gitignore` file will not remove it from the repository. It will only prevent untracked files or directories from being added in the future.

By utilizing the `.gitignore` file, developers can avoid cluttering their Git repositories with unnecessary files, focus on tracking and managing relevant code and assets, and ensure that sensitive information is not accidentally committed.

If you're using Git for version control, creating and maintaining a well-defined `.gitignore` file is recommended practice for effective repository management.

To configure a `.gitignore` file, you need to create a file named `.gitignore` in the root directory of your Git repository. This file contains a list of patterns that specify which files and directories should be ignored by Git. Here's how you can configure a `.gitignore` file:

1. Create the .gitignore file: Open a text editor and create a new file. Save it as .gitignore (note the leading dot) in the root directory of your Git repository.
2. Specify the patterns: In the .gitignore file, you can specify patterns to exclude files or directories from being tracked by Git. Each pattern should be on a new line. Here are some examples:

Ignore a specific file: To ignore a file named "example.txt," add the following line to the .gitignore file:

```
example.txt
```

Ignore all files with a specific extension: To ignore all files with the .log extension, add the following line:

```
*.log
```

Ignore a directory: To ignore a directory named "logs," add the following line:

```
logs/*.txt
```

Ignore all files and directories within a directory: To ignore everything within a directory named "temp," including subdirectories, add the following line:

```
temp/
```

3. Save the .gitignore file: Save the .gitignore file after adding the desired patterns.
4. Commit the .gitignore file: Use Git to commit the .gitignore file to your repository. Run the following commands in the terminal or command prompt:
git add .gitignore
git commit -m "Add .gitignore file"

This will stage and commit the .gitignore file to your Git repository.

After configuring the .gitignore file, Git will exclude the specified files and directories from being tracked and considered for commits. Make sure to review and update the .gitignore file as needed when adding new files or directories to your project.

Note that the .gitignore file only affects untracked files. If a file is already tracked by Git, you need to use `git rm --cached <file>` to remove it from version control.

1.1.3. Use of GitHub repository

Description of GitHub

GitHub is an Internet hosting service for software development and version control using Git. It provides the distributed version control of Git plus access control, bug tracking, software feature requests, task management, continuous integration, and wikis for every project.

GitHub is a web-based platform that provides hosting services for Git repositories. It offers a wide range of features and tools to facilitate collaboration, version control, and software development workflows. Here's a description of GitHub key features:

1. **Version Control:** GitHub is built on Git, a distributed version control system. It allows developers to track changes to their code, manage different versions of files, and collaborate with others on the same project. Git enables developers to work offline and merge changes easily.
2. **Hosting Service:** GitHub provides a cloud-based hosting service for Git repositories. It allows developers to store their code and related project files on remote servers, making it accessible from anywhere with an internet connection. This eliminates the need for setting up and maintaining local servers for code hosting.
3. **Collaboration:** GitHub is designed to support collaborative software development. It offers features like pull requests, which enable developers to propose changes to a project and review and discuss them with others. It also provides features for managing issues, assigning tasks, and tracking project progress.
4. **Social Coding:** GitHub fosters a social coding environment. Developers can follow other users, star repositories they find interesting, and contribute to open-source projects. This allows for community-driven development, collaboration, and knowledge sharing.
5. **Web Interface:** GitHub provides a user-friendly web interface that allows developers to interact with their repositories. They can create and manage repositories, browse code, review changes, and perform basic Git operations directly through the web interface. The web interface makes it easy to navigate code, view commit history, and access project-related information.

6. **Pull Requests:** Pull requests are a key feature of GitHub that enable developers to propose changes to a project. They facilitate code review and discussion among team members before merging the changes into the main codebase. Pull requests provide a structured workflow for collaboration and maintaining code quality.
7. **Issue Tracking:** GitHub includes an issue tracking system that allows developers to create and manage issues related to their projects. Issues can be used to report bugs, suggest enhancements, or discuss project-related topics. Issue tracking helps teams stay organized and manage project tasks effectively.
8. **Continuous Integration and Deployment:** GitHub integrates with various continuous integration and deployment tools. This allows developers to automate build processes, run tests, and deploy applications based on triggers such as code commits or pull requests. Continuous integration and deployment help maintain code quality and streamline the development workflow.
9. **Documentation and Wikis:** GitHub provides features for creating and hosting project documentation, including README files, wikis, and project websites. This allows developers to document their projects, provide usage instructions, and share additional resources with users and contributors.
10. **Integration with Third-Party Tools:** GitHub integrates with a wide range of third-party tools and services, such as project management tools, code editors, testing frameworks, and deployment services. These integrations enhance the development workflow by connecting GitHub with other essential tools used in software development.

Create account on GitHub

Here's a step-by-step guide on how to create an account on GitHub:

1. **Open the GitHub website:** Go to the GitHub homepage by typing "github.com" in your web browser's address bar or by clicking [here](#).
2. **Sign up:** On the GitHub homepage, you will see a "Sign up" button. Click on it to proceed.
3. **Provide your information:** GitHub will ask you to provide the following information to create your account:
4. **Username:** Choose a unique username for your GitHub account. This will be part of your GitHub profile URL (e.g., [github.com/username](#)).

5. Email address: Enter a valid email address that you want to associate with your GitHub account. GitHub will use this email address for notifications and account-related communication.
6. Password: Create a strong password to secure your GitHub account. Make sure it meets the specified requirements (e.g., minimum length, combination of uppercase and lowercase letters, numbers, and special characters).
7. Choose a plan: GitHub offers different plans, including free and paid options. Select the plan that suits your needs. If you're just getting started, the free plan should be sufficient.
8. Complete the CAPTCHA: To verify that you're a human and not a bot, you may need to complete a CAPTCHA or a similar verification process. Follow the instructions on the screen to proceed.
9. Read and accept the terms: Review GitHub's Terms of Service and Privacy Policy. If you agree to the terms, check the box to indicate your acceptance.
10. Customize your experience (if prompted) Optional: GitHub may ask you to provide additional information, such as your experience level and interests. You can choose to fill out these details or skip them for now. You can always update your profile information later.
11. Verify your email address (if required): Depending on your sign-up method, GitHub may send a verification email to the email address you provided. Open the email and click on the verification link to verify your email address.
12. Set up your profile: Once you've created your GitHub account and verified your email address, you can proceed to set up your profile. You can add a profile picture, provide a brief bio, and customize other details to personalize your GitHub profile.

Remember to keep your GitHub account credentials secure and consider enabling two-factor authentication (2FA) for an extra layer of security.

Create new remote repository

Here's a step-by-step guide on how to create a new remote repository on GitHub:

1. Sign in to your GitHub account: Go to the GitHub homepage (github.com) and sign in with your username and password.
2. Access the repository creation page: Once you're signed in, click on the "+" icon in the top-right corner of the GitHub page. From the dropdown menu, select "New repository." Alternatively, you

can directly visit the repository creation page by going to github.com/new.

3. Provide repository details:
4. Repository name: Enter a unique name for your repository. Choose a descriptive name that reflects the purpose or content of your project.
5. Description (optional): Add a brief description to provide more information about your repository (e.g., its purpose, features, etc.).
6. Visibility: Choose whether you want the repository to be public (visible to everyone) or private (accessible only to you and collaborators you invite). Note that private repositories require a paid GitHub plan.
7. Initialize this repository with a README: If you want to create an initial README file, which is commonly used to provide documentation or project information, check this option. It's recommended to check this option to have a starting point for your repository.
8. Choose repository options (optional):
9. Add a .gitignore: If your project requires ignoring specific files or directories, you can select a .gitignore template suitable for your project's programming language or framework. This helps Git ignore unnecessary files when tracking changes.
10. Choose a license: If you want to include a license file in your repository to specify how others can use and distribute your code, you can choose a license template. GitHub offers various open-source licenses for different purposes.
11. Create the repository: Once you've provided the required information and selected any desired options, click on the "Create repository" button. GitHub will now create your new remote repository.
12. Repository setup: After creating the repository, you'll be redirected to the repository page. Here, you can further configure and manage your repository, such as adding files, creating branches, setting up project boards, and inviting collaborators.

Apply git commands related to repository

Git clone: The "**git clone**" command is used to create a local copy of a remote repository. It allows you to download all the files and commit history from a remote Git repository to your local machine. Here is how you can use the "git clone" command:

- Open your terminal or command prompt: Launch the terminal or command prompt application on your local machine.
- Navigate to the directory where you want to clone the repository: Use the "cd" command to change to the desired directory. For example, if you want to clone the repository to your "Documents" folder, you can use the following command:

cd Documents

- Clone the repository: To clone a remote repository, you need the URL of the repository. The URL can be found on the repository's GitHub page. Use the following command to clone the repository:

```
git clone <repository-url>
```

Replace "<repository-url>" with the URL of the repository you want to clone. For example:

```
git clone https://github.com/username/repository-name.git
```

The repository will be cloned into a new directory with the same name as the repository. If you want to specify a different directory name, you can add it as an additional argument at the end of the command.

- Authenticate (if required): If the repository is private and requires authentication, Git will prompt you to provide your GitHub username and password or a personal access token. Enter the credentials to proceed with the cloning process.
- Wait for the cloning process to complete: Git will download all the files and commit history from the remote repository to your local machine. The time taken for cloning depends on the size of the repository and your internet connection speed.
- Repository cloned successfully: Once the cloning process is complete, you will see a message indicating that the repository has been cloned successfully.

You now have a local copy of the remote repository on your machine. You can navigate into the cloned directory using the "cd" command and start working with the files, making changes, and pushing your own commits to the remote repository.

Remember to regularly pull changes from the remote repository to keep your local copy up to date with the latest changes made by other contributors.

Git remote

The "git remote" command in Git is used to manage the remote repositories associated with your local repository. It allows you to view, add, rename, or remove remote repositories. Here are some common use cases of the "git remote" command:

View remote repositories:

```
git remote
```

This command will list the names of all remote repositories associated with your local repository. By default, the primary remote repository is named "origin."

Show detailed information about a specific remote repository:

```
git remote show <remote-name>
```

Replace "<remote-name>" with the name of the remote repository. This command displays information such as the remote URL, the branches tracked by the remote repository, and more.

Add a remote repository:

```
git remote add <remote-name> <repository-url>
```

Replace "<remote-name>" with a name of your choice to identify the remote repository and "<repository-url>" with the URL of the remote repository you want to add. This allows you to connect your local repository with the remote repository.

Rename a remote repository:

```
git remote rename <old-name> <new-name>
```

Replace "<old-name>" with the current name of the remote repository and "<new-name>" with the new name you want to assign to it.

Remove a remote repository:

```
git remote remove <remote-name>
```

Replace "<remote-name>" with the name of the remote repository you want to remove. This command disconnects your local repository from the remote repository.

Update the URL of a remote repository:

```
git remote set-url <remote-name> <new-url>
```

Replace "<remote-name>" with the name of the remote repository and "<new-url>" with the new URL you want to set for the remote repository. This can be useful if the remote repository's URL has changed.

These are some common commands for managing remote repositories using "git remote." There are additional options and functionalities available for more specific use cases. You can find more information about these commands and their options in the Git documentation.

Learning unit 2: Manipulate files

2.2. Definition of general key terms

Status: The "status" in Git refers to the current state of your working directory and staging area. When you run the "git status" command, Git provides information about the modified, added, or deleted files in your repository. It helps you track the changes made to your files and see which files are ready to be committed.

Branch: A "branch" in Git is a parallel line of development. It represents an independent line of work within a repository. Each branch can have its own commits, allowing multiple people to work on different features or bug fixes simultaneously. The main branch, usually named "master" or "main," represents the default and most stable version of the project. Creating and switching between branches allows you to work on different features or experiments without affecting the main branch.

Commit: A "commit" in Git represents a snapshot of your repository at a specific point in time. It captures the changes made to files and includes a unique identifier called a commit hash. When you make a commit, you are saving your changes permanently in the Git history. Each commit contains a message describing the changes made, providing a concise summary of the modifications. Commits are essential for tracking the history of a project, allowing you to review changes, revert to previous versions, and collaborate with others.

🚦 Add file change to git staging area

✓ Operation on git status command

Check the status of your repository: Run the following command to see the current status of your repository and any file changes:

git status

• View new untracked file

View new untracked files: The "git status" command will show any untracked files in your working directory. These are files that Git is not currently tracking. To add a new untracked file to the staging area, you can use the following command:

git add <file-name>

Replace "<file-name>" with the name of the untracked file you want to add. This command stages the file, making it ready to be committed.

```
user@MyPeOpLe MINGW64 ~/desktop/testsite (master)
$ ls
img/  img1.PNG  index/  scripts/  server/  styles/

user@MyPeOpLe MINGW64 ~/desktop/testsite (master)
$ git add index

user@MyPeOpLe MINGW64 ~/desktop/testsite (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   index/index.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    img1.PNG
    scripts/
    server/
    styles/
```

• View modified file

The "git status" command also displays modified files in your working directory. These are files that have been changed since the last commit. To add a modified file to the staging area, use the same "git add" command mentioned above.

- **View deleted file**

If you have deleted a file from your working directory, the "git status" command will show it as a deleted file. To stage the deletion, use the "git add" command as well:

```
git add <file-name>
```

Git will now consider the file changes you added using the "git add" command as part of the staging area.

- **Verify the staging area:** To check the status of the staging area and see which files are staged for the next commit, you can use the "git status" command again. It will display the files that have been added to the staging area and are ready to be committed.

Remember, adding file changes to the staging area is an important step before committing your changes. The staging area allows you to selectively include specific changes in each commit, providing a more granular and organized approach to version control.

✓ **Operation on git add command**

The "git add" command in Git is used to add file changes to the staging area. It allows you to specify which files or folders you want to include in the next commit. Here are the different operations you can perform with the "git add" command:

- **Stage all files:**

```
git add .
```

The period (".") is used to represent the current directory. This command stages all modified and new files in the current directory and its subdirectories.

- **Stage a file:**

```
git add <file-name>
```

Replace "<file-name>" with the name of the specific file you want to stage. This command adds the specified file to the staging area.

- **Stage a folder:**

```
git add <folder-name>
```

Replace "<folder-name>" with the name of the specific folder you want to stage. This command adds all the files within the folder and its subdirectories to the staging area.

It's important to note that the "git add" command only adds the changes to the staging area and does not commit them. After using "git add," the files are ready to be included in the next commit. To commit the changes, you need to use the "git commit" command.

Here are a few additional options you can use with the "git add" command:

-p or --patch: This option allows you to interactively select and stage specific changes within a file.

--all or -A: This option stages all modified, deleted, and new files, including files in subdirectories.

--ignore-removal: This option stages modified and new files but ignores deleted files.

--update or -u: This option stages modified and deleted files but does not include new files.

These options provide flexibility in selecting and staging specific changes based on your needs.

Remember to use the "git status" command to verify the changes staged in the staging area before committing them.

Operations on git reset command for:

• Unstage a file

operations on the git reset command for unstaging a file and deleting/staging a file or folder.

➤ To unstage a file using git reset, you can follow these steps:

Open your command line interface or terminal.

Navigate to the root directory of your Git repository.

Execute the following command to unstage the file:

git reset HEAD <file>

Replace <file> with the path to the file you want to unstage. For example, if you want to unstage a file named "example.txt" located in the root directory of your repository, the command would be:

git reset HEAD example.txt

This command removes the file from the staging area, but it will still remain in your working directory.

• Deleting and staging file/folder

- To delete a file or folder and stage the deletion using git reset, follow these steps:

Open your command line interface or terminal.

Navigate to the root directory of your Git repository.

Execute the following command to delete the file or folder and stage the deletion:

git rm <file/folder>

Replace <file/folder> with the path to the file or folder you want to delete. For example, to delete a file named "example.txt" located in the root directory, the command would be:

git rm example.txt

If you want to delete an entire folder, you can use the -r (recursive) flag:

git rm -r folder/

After executing the git rm command, Git stages the deletion of the file or folder. To permanently remove the file or folder from your repository, you need to commit the changes using git commit.

Remember to be cautious when using the git reset and git rm commands, as they modify your repository's history. Always double-check the changes you're making and ensure they align with your intentions.

Commit File changes to git local repository

✓ Best practice of creating a commit message

operations on the git commit command like committing a file and editing the commit message, Here are the steps to commit file changes to a Git local repository:

Open your command line interface or terminal.

Navigate to the root directory of your Git repository.

Use the following command to view the status of your repository and see the changes you've made:

git status

You will see a list of modified files or untracked files. To commit a file, you need to add it to the staging area using the git add command. For example, to stage a file named "example.txt" located in the root directory, use the following command:

git add example.txt

You can also use the git add command with wildcards or add multiple files at once.

✓ Operation on git commit command

• Commit a file

Once your files are staged, you can create a commit by using the git commit command. The commit message should be descriptive and concise, summarizing the changes you're making. It's considered a best practice to follow these guidelines:

Start the message with a capitalized verb in the imperative mood (e.g., "Add," "Fix," "Update").

Keep the subject line (first line) to a maximum of 50-72 characters.

Add a more detailed description in the body of the commit message, if necessary, using additional lines.

To commit your changes with a message, execute the following command:

git commit -m "Your commit message here"

Replace "Your commit message here" with your actual commit message.

• Edit commit message

If you need to edit the commit message after committing, you can use the git commit --amend command. This command allows you to modify the most recent commit message, including the subject line and the body. Execute the command, and a text editor will open, allowing you to make your changes.

git commit --amend

Once you're done editing, save and close the text editor to finalize the changes.

It's important to note that commit messages are essential for maintaining a clear history of your repository and understanding the changes made at a glance. Aim to create meaningful and informative commit messages to facilitate collaboration and future reference.

🔧 Operations on git log command

• To see simplified list of commit

git log command to view a simplified list of commits and a more detailed list of commits. The **git log** command is used to display the commit history of a Git repository.

To see a simplified list of commits using git log, follow these steps:

Open your command line interface or terminal.

Navigate to the root directory of your Git repository.

Execute the following command:

git log

This will display a simplified list of commits in your repository, showing the commit hash (SHA-1), author, date, and commit message.

You can navigate through the commit log using the arrow keys or by scrolling. Press q to exit the log view and return to the command line.

• To see a list of commits with more detail

To see a more detailed list of commits, including the changes made in each commit, you can use additional options with the git log command. Here are a few commonly used options:

git log -p: This option displays the commit history with the actual changes made in each commit, showing the diff for each commit.

git log --stat: This option provides a concise summary of the changes made in each commit, including the number of files changed and the number of lines added or removed.

git log --graph: This option visualizes the commit history as a text-based graph, showing the branching and merging of commits.

You can combine these options as needed to get the level of detail you require. For example, **git log -p --stat** would show both the actual changes and a summary of each commit.

Additionally, there are many more options and flags available with the git log command, such as filtering commits by author or date range. You can refer to the official Git documentation or use **git log --help** for more information on these options.

Manage branches

A branch is a separate copy of the codebase that can be used for testing or development purposes. Developers can create a new branch from the main codebase, make changes to the branch, and then merge those changes back into the main codebase when they are ready.

✓ Operations on branches

Create a branch:

To create a new branch in Git, you can use the following command:

git branch <branch_name>

This will create a new branch with the specified name based on the current branch you are on.

List branches:

To list all the branches in your Git repository, you can use the following command:

git branch

This will display a list of all the local branches in your repository. The currently active branch will be marked with an asterisk (*).

Delete local branch:

To delete a local branch in Git, you can use the following command:

git branch -d <branch_name>

Replace <branch_name> with the name of the branch you want to delete. Note that you cannot delete the branch you are currently on. If you want to delete the current branch, you need to switch to another branch first.

Delete remote branch:

To delete a remote branch in Git, you can use the following command:

git push origin --delete <branch_name>

Replace <branch_name> with the name of the branch you want to delete on the remote repository. This command will remove the specified branch from the remote repository.

Switch branch:

To switch to a different branch in Git, you can use the following command:

`git checkout <branch_name>`

Replace <branch_name> with the name of the branch you want to switch to. This command will update your working directory to the specified branch.

Rename branch:

To rename a branch in Git, you can use the following command:

`git branch -m <old_branch_name> <new_branch_name>`

Replace <old_branch_name> with the current name of the branch you want to rename, and <new_branch_name> with the desired new name for the branch.

Learning unit 3: Ship codes

3.1. Definition of general key terms

Pull:

Pull is a Git command used to update your local repository with the latest changes from a remote repository. It combines the "fetch" operation (retrieving changes from the remote repository) and the "merge" operation (applying those changes to your local branch).

Fetch:

Fetch is a Git command that retrieves the latest changes from a remote repository without modifying your local branches. It updates the remote-tracking branches, allowing you to see the changes made by others.

Push:

Push is a Git command used to upload your local commits to a remote repository. It sends your changes to the remote repository, updating the branches and making your work available to others.

Pull request:

A pull request (PR) is a feature found in many version control systems, including Git. It is a way to propose changes to a codebase hosted in a remote repository. When you create a pull request, you are requesting the repository maintainers to review and potentially merge your changes into the main branch.

Merge:

Merge is a Git operation that combines changes from different branches. It integrates the changes made in one branch (source branch) into another branch (target branch). This brings the changes from the source branch into the target branch, allowing you to have a consolidated version of the code.

3.2. Fetch file from GitHub repository

3.2.1. Operations on 'git fetch' command

🔧 Fetch the remote repository:

To fetch the latest changes from the remote repository, you can use the following command:

git fetch

This command will retrieve all branches from the remote repository and update your remote-tracking branches, allowing you to see the latest changes made by others.

🔧 Fetch a specific branch:

If you only want to fetch changes for a specific branch, you can specify the branch name after the remote repository's name. Here's an example:

git fetch origin <branch_name>

Replace <branch_name> with the name of the branch you want to fetch from the remote repository (e.g., master, develop, etc.). This command will fetch the latest changes for that specific branch only.

🔧 Fetch all branches simultaneously:

By default, the git fetch command fetches all branches from the remote repository. However, if you have multiple remote repositories and want to fetch all branches from all of them simultaneously, you can use the --all flag. Here's an example:

git fetch --all

This command will fetch all branches from all remote repositories configured in your local repository.

🔧 Synchronize the local repository:

The git fetch command is often used to synchronize the local repository with the remote repository, without modifying your local branches. By fetching the

latest changes, you can see what others have been working on and decide how to integrate those changes into your own branches later.

Remember that after fetching, you can use other Git commands like 'git merge' or 'git rebase' to incorporate the fetched changes into your local branches as per your requirements.

3.2.2. Operations on 'git pull' command

✚ Default git pull:

The default usage of git pull combines the **git fetch** and **git merge** commands in one step. It fetches the latest changes from the remote repository and then merges them into your current branch. Here's the command:

git pull

This command is equivalent to running **git fetch** followed by **git merge** to integrate the fetched changes into your current branch.

✚ Git pull remote branch:

To pull changes from a specific remote branch, you can use the following command:

git pull <remote_name> <branch_name>

Replace <remote_name> with the name of the remote repository (e.g., origin) and <branch_name> with the name of the branch you want to pull changes from. This command will fetch the latest changes from the specified remote branch and merge them into your current branch.

✚ Git force pull:

In some cases, you may want to forcefully overwrite your local changes with the remote changes when pulling. To do this, you can use the **--force** or **-f** flag with the git pull command. Here's an example:

git pull --force

Caution: Forcefully pulling can result in the loss of your local changes, so use this option carefully and ensure you have a backup of any important modifications.

✚ Git pull origin master:

To specifically pull changes from the master branch of the origin remote repository, you can use the following command:

git pull origin master


This command fetches the latest changes from the master branch of the origin remote repository and merges them into your current branch.

Remember to always review the changes being pulled before merging them into your branch to ensure they align with your expectations and do not introduce any conflicts or unexpected behaviors.

3.2.3. Push files to remote branch

✓ Tags used on git push command


In addition to pushing files to a remote branch, you can also use tags with the git push command to push specific versions or milestones of your code. Here's how you can do it:

 Push files to a remote branch:

To push your local changes to a remote branch, you can use the following command:

git push <remote_name> <local_branch>:<remote_branch>

Replace <remote_name> with the name of the remote repository (e.g., origin), <local_branch> with the name of the local branch you want to push, and <remote_branch> with the name of the branch on the remote repository where you want to push your changes.

 Push tags:

To push tags to the remote repository, you can use the git push command with the --tags option. Here's an example:

git push <remote_name> --tags

Replace <remote_name> with the name of the remote repository where you want to push the tags. This command will push all the tags from your local repository to the specified remote repository.

If you want to push a specific tag, you can use the following command:

git push <remote_name> <tag_name>

Replace <remote_name> with the name of the remote repository and <tag_name> with the name of the specific tag you want to push.

Remember to ensure that you have the necessary permissions and the remote repository has been set up as a remote in your local repository before attempting to push your changes or tags.

3.2.4. Operations on git push

Here are the operations you can perform using the git push command:

✚ Push to origin master:

To push your local branch to the master branch on the origin remote repository, you can use the following command:

git push origin <local_branch>:master

Replace <local_branch> with the name of your local branch. This command will push the changes from your local branch to the master branch on the origin remote repository.

✚ Git push force:

In certain situations, you may need to force push your changes to a remote repository, overwriting any conflicting changes that may exist. Use the --force or -f flag with the git push command, like this:

git push --force

Caution: Force pushing can lead to the loss of other contributors' work or cause conflicts. Use this option with caution and only when necessary.

✚ Git push verbose:

If you want to see more detailed information about the push operation, you can use the --verbose or -v flag. It provides additional output, such as the names of the branches and the objects being pushed. Here's an example:

git push --verbose

This command will display detailed information about the push operation, including the progress and any error messages if they occur.

✚ Delete a remote branch:

To delete a remote branch, you can use the git push command with the --delete option. Here's an example:

git push origin --delete <branch_name>

Replace <branch_name> with the name of the branch you want to delete on the remote repository. This command will remove the specified branch from the remote repository.

Remember to use caution when performing operations like force pushing or deleting remote branches, as they can have permanent consequences and affect the work of other collaborators.

3.2.5. Merge branches on remote repository

Merge branches on a remote repository:

To merge branches on a remote repository, you typically follow these steps:

a. Fetch the latest changes from the remote repository:

git fetch

b. Switch to the target branch where you want to merge changes:

git checkout <target_branch>

c. Merge the source branch into the target branch:

git merge <source_branch>

d. Push the merged changes to the remote repository:

git push origin <target_branch>

Replace <target_branch> with the name of the branch you want to merge changes into, and <source_branch> with the name of the branch containing the changes you want to merge.

✓ Operation on git rebase command

The git rebase command allows you to modify the commit history of a branch, typically by moving, combining, or altering commits. Some common operations include:

a. Rebase a branch onto another branch:

git rebase <base_branch>

This command replays the commits of the current branch on top of the <base_branch>, integrating the changes from the <base_branch> into your current branch.

b. Interactive rebase:

git rebase -i <commit>

This command allows you to interactively modify commits, such as squashing, reordering, or editing commit messages. It opens an editor where you can specify the changes you want to make.

✓ create pull request

To create a pull request, you typically follow these steps:

- a. Commit your changes to a branch in your local repository.
- b. Push the branch to the remote repository:


git push origin <branch_name>

- c. Go to the remote repository's hosting service (e.g., GitHub, GitLab) and navigate to your branch.
- d. Click on the "Create Pull Request" or similar button.
- e. Fill in the necessary details for the pull request, such as the source branch and the target branch.
- f. Submit the pull request.

The pull request allows others to review your changes and merge them into the target branch after reviewing and approving the code.

Remember to review any potential conflicts, communicate with your team, and ensure you have the necessary permissions to push changes or create pull requests on the remote repository.

✓ Operations on git merge

 Merge the specified commit to the current active branch:

If you want to merge a specific commit into your current active branch, you can use the following command:

git merge <commit>

Replace <commit> with the commit hash or reference (such as a branch name) of the commit you want to merge. This command will merge the changes introduced by that commit into your current branch.

 Merge commits into the master branch:

To merge commits from another branch into the master branch, you can follow these steps:

- a. Checkout the master branch:

git checkout master

- b. Merge the desired branch into the master branch:

git merge <branch_name>

Replace <branch_name> with the name of the branch containing the commits you want to merge into master. This command will integrate the changes from the specified branch into the master branch.

🔧 Git merge branch:

To merge changes from one branch into another branch, you can use the following command:

git merge <source_branch>

Replace <source_branch> with the name of the branch you want to merge into the current active branch. This command will merge the changes from the source branch into the current branch.

Remember to resolve any conflicts that may arise during the merge process. After merging, it's good practice to thoroughly test the merged code to ensure it functions as expected.

Integrated/Summative assessment

Integrated Situation

SEZERANO ALPHA Crispin is a Senior Developer of Innovate company Ltd located at Huye District, he assigned a project to 5 developers to design web application that contains different forms such as: login form, Student Registration form, Entertainment form, courses registration form and book registration form, using html language,css and PHP(for CRUD operations), but due to Covid-19 pandemic developers were not able to work together on the given task and become difficult to control them. Senior developer decided to assign tasks to each developer respectively and remotely. and he recommended them to work individually on a given task. He told them to create their own repository related to a given task, As a one of 5 developers you are hired to choose one task from above, work on it and create Pull Request of your task to be merged on the main branch, use GitHub as version control platform. Submit Pull Request link to the senior developer.

Tools, material and equipment are provided.

This integrated situation is inclusiveness, The time allowed to accomplish this task is 4 hours.

Database NAME: students_db

Tables:

- students(stud_ID(auto_increment,Primary_key) first_name,last_name, DOB, address, username, password)
- entertainment(ent_ID(auto_increment,primary_key),
crub_name,stud_ID(foreign_key),created_date(current_TIMESTAMP), mentor_Name,mentor_Phone)
- courses(course_ID(auto_increment,primary_key), course_Title,course_Code,stud_ID(foreign_key),max_point)
- book(book_ID(auto_increment,primary_key), book_Title,book_Author,published_Date)