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.

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.

Benefits of version control

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.

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.

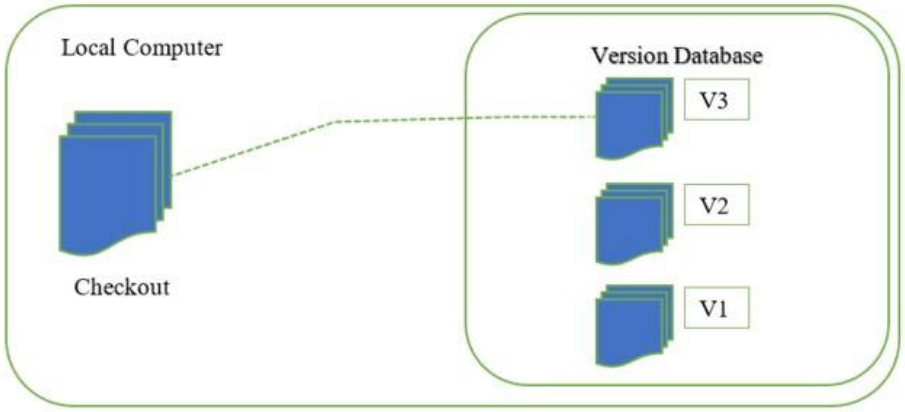
Types of version control

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.



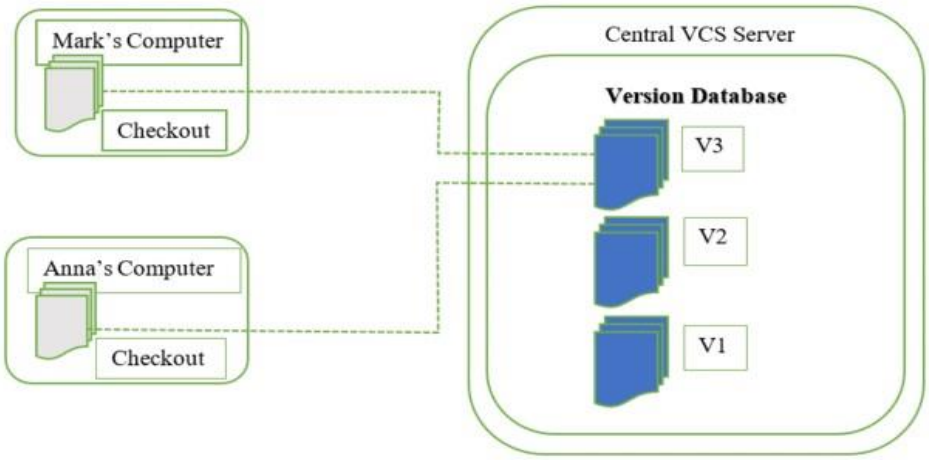
The disadvantage of local version control is that it does not provide collaboration features that enable multiple developers to work on the same codebase simultaneously. Moreover, local version control systems do not provide any backup or disaster recovery mechanism.

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.

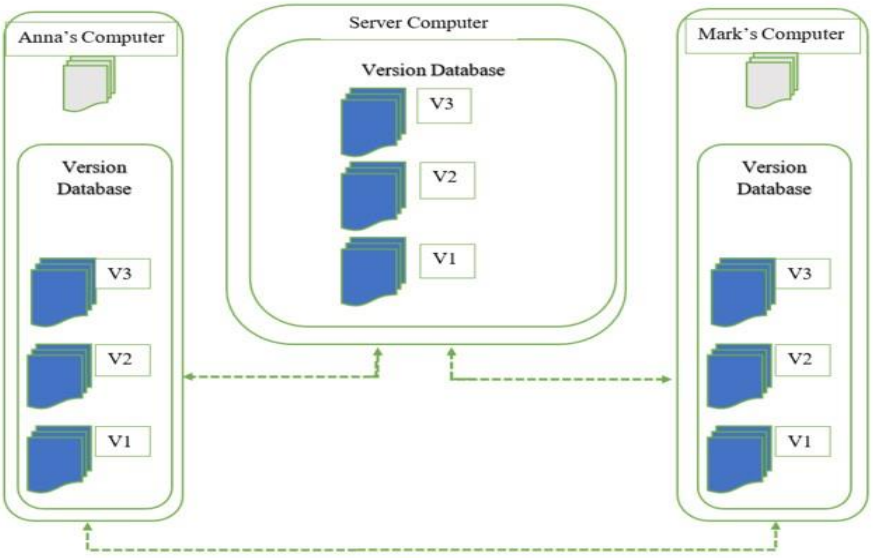
The most well-known examples of centralized version control systems are Microsoft Team Foundation Server (TFS) and SVN.

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 (Global Information Tracker.)

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.

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://gitscm.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.