**1. What is *Git* and why is it used?**

**Git** is a decentralized version control system (**DVCS**) that developers primarily use to manage source code across diverse development projects. It ensures better code quality, facilitates teamwork, and provides a robust backup mechanism.

**Benefits of Using Git**

* **History and Time Travel**: Git preserves every version of a project, making it possible to roll back changes or inspect code at any point.
* **Security and Integrity**: Each file in Git has a unique SHA-1 hash, which means unauthorized changes are easily detected.
* **Collaboration**: Git allows multiple developers to work on a project simultaneously without conflict.
* **Code Reviews**: Git simplifies the process of reviewing code before it's incorporated into the main codebase.

**Key Concepts**

**Commit**

A **commit** represents a saved state of your project. It is a snapshot of all the files in your project at a given point in time. Each commit has a commit message associated with it.

**Branch**

A **branch** is an independent line of development. It allows you to work on a feature or bug fix, isolate it from the rest of the codebase until it is ready, and then merge it back in.

**Master** is a default branch present in all Git repositories. You can create many other branches.

**Merge**

**Merging** in Git means taking the independent developments of two branches and combining them into a single branch.

After merging, the two branches from which you were merging are the same as the merged branch.

**Remote and Origin**

A **remote** is a common repository on a server or accessible memory that all team members use to exchange their changes.

An **origin** is a source of commit objects (store of all committed versions).

**The Command Line vs Visual Tools**

For everyday tasks, both CLI and GUI are powerful, but it's essential to understand **Git's** underlying mechanics so that you can troubleshoot and manage complex tasks efficiently.

**Beginner Tips**

* Make frequent commits.
* Write clear and concise commit messages.
* Push your code to a remote repository often, especially before taking a break or finishing a task.

**2. How does *Git* differ from other version control systems (*VCS*)?**

**Git** offers numerous advantages over traditional **Version Control Systems** (VCS), providing a distributed and efficient version control model.

**Unique Features**

* **Distributed Model**: Git decentralized nature empowers every set of files with a complete repository history and enables full-fledged operations without network connectivity. This stands in contrast to centralized systems that require a constant network connection and utilize a single centralized server for all project data.
* **Quick Operations**: Git is renowned for its fast and efficient operations. Unlike older VCS, which often performed slowly or required substantial server interactions, Git uses local operations, resulting in impressive speed.
* **Data Integrity**: Git ensures the cryptographic integrity of every piece of data, securing against file corruption, loss, or unauthorized alterations. The absence of this feature in some traditional VCS can pose data integrity risks.
* **Streamlined Branching**: While segments or branches in older VCS were often used for parallel development, Git branches are lightweight and designed for quick and frequent usage.
* **Customizable Workflows**: Git permits versatile workflows due to its distributed nature and the ability to manipulate the commit history before sharing changes with others.
* **Staging Area (Index)**: Before committing, developers use the "staging area" to selectively include specific changes in a commit rather than committing every altered file.
* **Powerful Library of Core Commands**: Git's robust set of indispensable commands enables a broad range of version control capabilities, from historical revision exploration to collaborative feature code management.

**Consider the User**

* **Simplicity and Flexibility**: Unlike some traditional VCS, Git is designed to be both user-friendly and capable of supporting complex tasks.
* **Efficiency and Performance**: Git's rapid and streamlined processes lead to optimized efficiency.

**Unexpected Change? No Problem!**

* **Snapshot vs File Tracking**: Git's snapshots record the current state of the complete repository, whereas some traditional VCS, such as CVS and Subversion, primarily track file changes.
* **Atomic Commits**: Every change is distinct and exclusive from others, enabling precise control over data synchronization.

**Git: Now and Hereafter**

* **Modern Landscape Readiness**: Git melodiously integrates with modern software development paradigms like Continuous Integration/Continuous Deployment (CI/CD).
* **Cross-Platform Compatibility**: Be it Windows, macOS, or Linux, Git operates identically.

**What Are Some Drawbacks of Other VCS?**

* **Risk of Single Point of Failure**: If the central server is compromised, it poses a threat to the entire project's data.
* **Version Numbering Limitations**: Users might be confined to a monotonically increasing number for versions, restricting granularity.
* **Long-Term Integrity Risks**: Without cryptographic validation, files can be altered unnoticed.
* **Data Loss Complications**: If a central server/repository is compromised or damaged, data loss is severe.
* **Limited Offline Functionality**: Users are hindered by the necessity of a steady network connection.
* **Stale Branches**: Centralized repositories can have stale branches, hindering collaboration.

**3. What is the difference between *Git* and *GitHub*?**

**Git** and **GitHub** are often confused due to their similar names. However, they serve different **version control** and development hosting purposes.

**Git**

**Git** is a **distributed version control system** that manages code and tracks its changes. It's mainly a local solution, only allowing developers to coordinate directly.

**Key Features**

* **Local Management**: Developers can work on their repositories without the need for a central server.
* **Speed**: Operations such as committing, branching, and merging are rapid, ideal for local development.
* **Staging Area**: Changes are first staged before getting committed for better control.
* **Content Tracking**: Rather than file-based, Git is content-based, ensuring efficient tracking.

**GitHub**

**GitHub**, on the other hand, is a **web-based platform** that provides hosting for **Git repositories**. It offers features that extend beyond Git's core functionality.

**Key Features**

* **Remote Repositories**: It allows developers to sync their local repositories with remote ones, enabling collaborative work.
* **Code Hosting**: Developers can store and manage their code from anywhere, not just their local machines.
* **Collaboration Tools**: GitHub provides tools for project management, issue tracking, and team collaboration.
* **Code Review**: Multiple users can review and discuss proposed changes before they're merged into the main codebase.
* **Access Control**: Different users can have varying levels of access to a repository or organization.

**4. What is a *repository* in *Git*?**

A **Git repository** is a location where all the **version-controlled** files and their history are stored. It serves as the core component for collaborative and version control workflows.

Every Git repository has two primary parts: the **working tree** and the hidden .git **directory**, where Git keeps all of its metadata for the repository.

**Key Features**

* **Distributed**: Each contributor to the project has their own copy of the repository, including its complete history and version-tracking capability. This enables offline work and serves as a backup for the project.
* **Efficient Storage**: Git optimizes storage using techniques such as file hashing and delta compression, emphasizing minimal redundancy and data redundancy elimination.
* **Content Integrity**: All files in a Git repository, as well as their versions, are checksummed and verified on every Git operation for data integrity.
* **Work Tracking**: Git allows for tracking changes made to the project over time, helping individuals or teams understand the evolution of the project, who made specific changes, and why.

**.git Directory**

The .git directory is the **control center** of a Git repository, housing everything Git needs to manage the repository and its history. It includes:

* The **object database** that hosts all the data about commits, trees, and blobs.
* The **references** directory maintaining pointers to specific commits (e.g., branches and tags).
* Configuration files detailing the repository's settings and attributes.

**Working Tree**

The working tree is a directory where **tracked files** are extracted for **viewing** and **editing**. Any changes made to files in the working tree can be staged and committed to the repository to capture the changes.

Untracked files, i.e., files not previously staged or committed, coexist with the working tree and reside outside the Git management.

**Code Example: Working Tree**

Here is the Python code:

# Let's define a variable to point to the .git directory

git\_directory = '.git'

# Now, we can simulate the construction of our working tree using a dictionary

working\_tree = {

'folder1': {

'file1.py': 'print("Hello, World!")'

},

'folder2': {

'file2.py': 'print(2 + 2)'

}

}

**5. Explain the concept of a *commit* in *Git*.**

In **Git**, a **commit** represents a snapshot of a project at a specific point in time. Each commit records the changes made, the author, and a unique identifier. It is organized in a **directed acyclic graph** (DAG), which aligns with its branching model.

**Key Elements of a Git Commit**

* **Tree**: Represents the state of the project's file system at the time of the commit. The tree object is a record of the location and content of each file in the repository.
* **Parent Commit(s)**: Identifies the commit(s) from which the current commit originated. In the case of the initial commit, there's no parent.
* **Metadata**: Includes details such as the commit's author, date, and a message summarizing the changes made.

**The Three-Tree Architecture**

In a typical Git repository, there is a **working directory**, **staging area**, and **commit history**. Each tracks the project's state at various stages.

* **Working Directory**: Refers to the current, live set of files and directories. This is the version of the project that you actively work on and modify.
* **Staging Area** (also known as the "index"): Serves as a buffer zone. Files in this area are marked for inclusion in the next commit. You control what content moves from the working directory to the staging area.
* **Commit History** (in the Git directory): Represents the timeline of commits in the form of a versioned archive. It also houses the full content of the project at relevant commit points.

**The Git Lifecycle**

The standard git-flow follows a lifecycle that incorporates the three trees mentioned above.

* **Creating Commits**: You start by making changes to your working directory. When you're ready to record these changes, you add specific files from the working directory to the staging area. With the desired changes in the staging area, the next command captures the staging area's content and creates a new commit in the commit history.
* **Browsing History**: Git's commit history is time-ordered, typically from the most recent commit backward. Each commit influences how the project appears at a given point in time.

**Viewing Commit History**

You can view the commit history using commands such as git log. This command provides a structured list of commits, detailing the author, commit date, and an associated commit message.

**Practical Applications**

* **Code Reversion**: You can reset a project to an earlier commit, effectively reversing changes.
* **Collaboration**: Commits facilitate collaboration by enabling team members to understand changes and when they were made.
* **Code Review**: Before incorporating changes into the primary branch, commits offer a structured way for team members to review each other's code. This process ensures that new code meets the repository's standards.

**Code Example: Commit Creation and Viewing**

Here is the Git console command to create "Meaingful Commit" with a message and then to view the Commit.

# To make a commit with a message

git commit -m "Meaningful Commit"

# To view the commit history

git log

**6. What is the difference between a *working directory*, *staging area*, and *repository* in *Git*?**

**Git** is characterized by a three-tiered architecture which includes the following layers:

* **Working Directory**
* **Staging Area**
* **Repository**

Each layer plays a distinct role in the Git architecture.

**Core Concepts**

**Working Directory**

The working directory is your playground. It's your space for experimenting, where you can **edit files**, merge different versions, and even create new ones.

**Staging Area (Index)**

The staging area (also known as the Index) is a kind of **middle-ground**. Here you can make fine adjustments to what changes will be included in your next commit. You have two types of files here:

* **MDS**: Files that have been modified since your last commit
* **SML**: Files that have been changed and prepared for your next commit

When you're happy with your changes in the staging area, you "stage" them. Staging in this context is like flagging certain parts of your project that you want to be saved in your next commit. Notice that this step is completely **optional.**

**Repository (History)**

The repository is the final tier, dealing with the preservation of your work. Think of it as the **local repository** that serves as the safe house for all your commits. It keeps track of all changes and commits you've made, ensuring that you can still retrieve them even after major updates.

The repository is also broken down into three apparent "trees" - the Working Directory, the Staging Area, and the last one, the HEAD. The HEAD points to your most recent commit. These three trees can seem confusing at first. But really, they're straightforward to understand - they represent the current status of your project, changes that you're planning to commit, and the commits that you've confirmed you want to save in the future.

**7. Define *branching* in *Git* and its importance.**

**Branching** in **Git** allows for the independent development of distinct features or sections of the codebase. It's a crucial version control strategy, particularly in collaborative projects.

**Key Concepts**

* **Master/Trunk/Branch**: The primary branch serving as the project source.
* **Feature Branches**: Additional branches dedicated to specific features or tasks, often derived from the Master branch.
* **Merging**: The process of integrating changes from one branch into another. For instance, integrating finished feature branches into the Master.
* **Conflict Resolution**: The act of resolving overlapping changes made to the codebase across different branches.

**The Importance of Branching**

1. **Code Isolation**: Each feature branch is an isolated environment for developing a specific feature. This avoids interference with the main codebase.
2. **Collaborative Development**: Allows multiple developers to work on the same codebase concurrently, minimizing conflicts.
3. **Risk Mitigation**: Changes are kept separate until they are thoroughly tested. If an experimental feature doesn't work as expected or introduces bugs, it won't affect the more stable Master branch.
4. **Staging Area**: A dedicated space for integration and testing before code is deployed or merged into the Master branch.
5. **Feature Toggles**: Streamlines the process of selectively deploying new features for testing.
6. **Code Reviews**: Assists in code quality checks before merging, ensuring the Master branch remains stable.
7. **Task Management**: Aligns with project management tools and methodologies, providing a way to structure tasks in a version-controllable manner.
8. **Release Management**: Facilitates the isolation and verification of changes earmarked for specific releases.

**8. What is a HEAD in *Git*?**

In **Git**, the concept of a **working directory**, the **staging area** (or **index**), and the **repository** (or **commit history**) are all managed by the HEAD.

**Role of HEAD in Git**

* **Point of Reference**: HEAD identifies the current snapshot/commit in the history tree. Any actions in the working directory or staging area are based on this commit.
* **Connector between Levels**: It links the most recent commit in a branch to both the staging area and the working directory. Each commit is precisely what HEAD points to.

**Code Example: Working with HEAD**

Here is the visual representation of **How Git Works**.

Here is the python code:

# Check out the "first-branch" branch

git checkout first-branch

# Make some modifications and stage them

git add modified-file-1.txt

# Commit the staged changes

git commit -m "Modified file 1 in first branch"

# Verify the new `HEAD` after the commit

git log --oneline --decorate

The **git log** command in the above code includes the --decorate option, displaying the most recent commit and any associated references.

**9. What does the 'clone' operation in *Git* do?**

**Cloning** in **Git** creates a **local copy** of the repository, allowing for version control and collaborative work. It establishes a link to a specified remote, permitting **fetch** and **push** operations.

**Key Attributes**

* **Two-Part Process**: Cloning involves duplicating the repository and setting up the local environment.
* **Full Version History**: The cloned repository typically includes all commits and file versions.
* **Remote Linkage**: A relationship is built with a designated remote for data synchronization.

**Terminology Overview**

* **Local Repository**: Storage on the user's machine where all project and version data reside.
* **Remote Repository**: An external, shared storage often on a server, accessible for collaborative work.
* **Working Directory**: The location on the user's file system where files are manipulated and changes are tracked for commits.

**Core Functionality**

* **Duplication**: The clone process reproduces the entire project, with all branches and commits, onto the user's machine. This offers a complete, standalone history.
* **Configuration**: The local environment is configured to maintain synchronization with a specific remote repository. By default, the repository from which the clone is made (origin) is set as the primary remote, but additional remotes can be added if required.
* **Integrity**: Git ensures the integrity of the copied data, such as commits and file snapshots, during the cloning process.
* **Selective History Contraints**: While it's common for a clone to receive the entire repository history, limited cloning is also possible, especially with large repositories. Technique like shallow cloning or specifying a certain commit for cloning can be used.

**Forming the Connection**

After a successful clone, the local repository is configured to interact with a designated remote repository, typically on a centralized server like Github or Bitbucket.

For GitHub, cloning often uses the HTTPS mode. Upon updates to the remote, the local repository can be synchronized using standard Git commands (e.g., **fetch**, **pull**, **push**).

Technically, clonality is not tied to a specific remote URL; it's more about aligning the histories. For instance, after cloning from GitHub using HTTPS, remotes can be reconfigured to use SSH.

**Detailed Process**

1. **Acquiring Data**: The clone command contacts the remote repository, retrieves its history and files, and then saves this snapshot locally.
2. **Repository Creation**: The git clone execution sets up a new repository, employing the data fetched from the target remote, and thus creates both the working directory and associated .git directory, which hosts the entire Git control structure.
3. **Linkage Configuration**: The clone operation automatically establishes a remote connection, naming it "**origin**" by default. Should you have multiple remotes or wish to use a different name, configuration adjustments may be required.
4. **Extraction**: The clone accomplishes the creation and association of the local repository, which users can then "extract" to their working directories for manipulation.
5. **Status Verification**: Confirming the successful execution of a clone can be done by taking a look at the current remotes via git remote -v, which should display the origin.

**Technical Commands**

* **Clone**: Initiate the cloning process.

git clone <remote-repo-url>

* **Remote**: Verify origin setup.

git remote -v

**10. How does *Git* store information?**

While **Git** does use a filesystem to store data, it isn't exclusively file-based. The three main components of the **Git** data structure are the **repository**, the **index/staging area**, and the **local working copy**.

**Git Storage Model**

1. **Local Working Copy**: Where you directly interact with your files.
2. **Staging Area (Index)**: Serves as a sort of "holding area" for changes you want to include in your next commit.
3. **Local Repository**: Where **Git** keeps track of the complete history and stores different versions of your files.

**Repository**

The **repository** houses the following key components:

* **Object Database**: Persistently stores snapshots of files and directories as objects, which can be further classified into:
  + **Blob** objects for individual file contents.
  + **Tree** objects that replicate file system hierarchy.
  + **Commit** objects linked to a specific state in the project's history.
  + **Tag** objects marking specific commits, often used for releases.
* **Index/Stage**: A dynamic workspace that mirrors the next commit's expected state.
* **Head Pointer**: A reference to the latest commit in your current working branch
* **Branches**: Human-readable names that reference specific commits, facilitating independent lines of development.

**Git Object Names**

1. **Blobs**: SHA-1 hash is based on the file's content.
2. **Trees**: SHA-1 hash combines the file and directory contents within—changes in any file or sub-directory would trigger a new hash.
3. **Commits**: SHA-1 hash incorporates the tree representing the project's directory structure, along with parent commit(s), commit message, and author information.

**Git Object Storage**

Git allows for a 'lazy write' mechanism for objects. This means that an object file is written upon creation and remains unchanged until the content it represents changes. When a change occurs, it gets updated and its SHA-1 is recalculated. This particular feature is known as 'immutable storage'.

Here's an illustration for better understanding:

* Let's assume a file's content changes and you add the updated file to the staging area.
* The new version of the file is then stored as a **Blob** object, and the associated **Tree** object also gets updated.
* During the following commit, a new **Commit** object is created, which references the updated **Tree** object.
* Simultaneously, the **Branch** you're working on moves forward to reference this new commit.

This systematic approach assures the integrity of your project's history. Since any alteration in content, directory structure, or commit details will trigger a change in the SHA-1 hash of the related object(s), a change can be immediately recognized.

**11. How do you initialize a new *Git* repository?**

Let's go through the step-by-step process of **creating a new Git repository** and then getting started by **adding files**, **making commits**, and **pushing** the repository to a remote location such as GitHub or Bitbucket.

**Initializing a New Git Repository**

**Using the Command Line**

1. Navigate to your project folder using your terminal or command prompt.
2. Run the following command to initialize a new Git repository:

git init

**Using Git GUI**

Most Git GUI applications, like Sourcetree or GitKraken, facilitate setting up repositories through an intuitive interface. Once you've navigated to your project folder, choose the "Initialize/Create New Repository" option.

**Getting Started: Adding Files, Making Commits, and Pushing to Remote**

**Adding Files to the Staging Area**

Use the following command to **add** files or directories to the staging area:

# To add a specific file

git add filename

# To add all files and directories

git add .

**Committing Changes**

Once files are in the staging area, use this command to **commit** these changes:

git commit -m "Your commit message goes here"

**Associating a Remote Repository**

If you plan to push your local repository to an online service like GitHub, first **link** your local repository to a remote one:

git remote add origin your-remote-repository-url

Replace "your-remote-repository-url" with the URL provided by your online repository.

**Pushing to Remote**

Finally, use this command to **push** your local repository to the remote one:

git push -u origin master

After the initial "upstream" link is established, subsequent push commands can be executed with a simple git push.

**Handling Sensitive Information**

To ensure that sensitive data, such as API keys or credentials, are not shared, employ the **.gitignore** file to list files and patterns that should be excluded from version control.

In particular, avoid using files with such sensitive information before creating a .gitignore.

Here is the .gitignore file:

# Ignored file - example

credentials.txt

**12. Explain the purpose of the git status command.**

**git status** is an essential command for **tracking changes** in a Git repository. It provides important details such as file states, current working branch, untracked files, and more.

**Key Output Information**

* **On Branch:** Indicates the active branch, allowing quick navigation through different branches.
* **Changes to be Committed:** Lists modified or staged files, providing a quick glance at what will be included in the next commit.
* **Changes not staged for commit:** Highlights modified files that were not yet staged for the next commit.
* **Untracked Files:** Recognizes files in the working directory that are not a part of the repository.
* **Branch is Up to Date:** Provides this affirmation when the working branch is up-to-date with its remote counterpart.
* **Your Branch is Ahead of 'origin/master' by 1 Commit:** Notifies when the current local branch outpaces the linked remote branch by a set number of commits. This is common in team-based workflows and signifies the necessity of a push.

**Note**: The remote branch in the example is 'origin/master', and the number of commits ahead is 1.

**Code Example: Terminal Output of git status**

Here is the code:

On branch main

Your branch is up to date with 'origin/main'.

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: app.js

Untracked files:

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

server.js

no changes added to commit (use "git add" and/or "git commit -a")

**Code Example: Visual Diff of Staged Changes using git diff --staged**

Here is the code:

diff --git a/readme.txt b/readme.txt

index ce01362..0602794 100644

--- a/readme.txt

+++ b/readme.txt

@@ -1 +1 @@

- Initial file version

+ Modified during stage

**Code Example: Multifile git status Output**

Here is the code:

On branch: master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: file1.cpp

modified: file2.cpp

Both file3.cpp and file4.cpp are untracked.

Your branch and 'origin/master' have diverged,

and have 4 and 2 different commits each, respectively.

**Best Practices**

* Regularly use git status to stay updated on the repository's state.
* Review and comprehend the provided information before proceeding with any actions, such as staging or committing.

**13. What does the git add command do?**

**git add** is the first step in tracking changes you make to existing files or adding new files to your Git repository. It prepares the changes for the next commit by moving modified or new files to the **staging area**.

When you invoke git add, three major actions take place:

1. **File Selection**: You choose which **specific file changes** to include in the upcoming commit.
2. **File Comparison**: Git analyzes the selected files and identifies the exact line-by-line or block-level modifications.
3. **Staging Area Update**: The selected file changes are prepared and categorized. Once you're ready, you can make a commit action to move all the changes in the staging area to the repository.

**Relation Manager**

The staging area acts as a bridge between your working directory and the repository (history).

* **Working Directory**: Contains the active files you're currently editing.
* **Staging Area (Index)**: A kind of "Sandbox" where you selectively group changes together for precise commits.
* **Repository**: The historical record of all your commits and their respective states.

**Key Actions**

* **Add All Modifications**: Restage modified files to the current state: git add .
* **Add Interactively**: Handpick changes from modified files: git add -p
* **Merge Conflicts**: After addressing conflicts, ready files for commit: git add <filename>

**Code Example: Staging vs Git History**

Here is the Python code:

# File: my\_file.py (in the working directory)

# Current content

print("This line is part of the initial version.")

# Post staging changes (not committed yet)

print("This line will be part of the next commit.")

In our Python example, consider "file.py" as the file. Initially, the file contains one line, which was the initial version. Then "This line will be part of the next commit." gets added (not committed yet), so that these changes occur between git add and git commit.

**14. How do you create a *commit* in *Git*?**

Let's look at the process of **creating a commit in Git**:

**1. Identifying Tracked and Untracked Files**

Before making a commit, the user should review any changes in the working directory and staging area.

**Untracked Files**: These are new files that Git does not track yet. To track untracked files and subsequently include them in a commit, they need to be staged using the git add command.

**Tracked Files**: Files that have previously been committed or added to the staging area can be included in the next commit.

**2. Ways to Select Files for Commit**

**git add Options**

* **Specific Files**: Directly add individual files or a group of files using their names or paths.

git add file1.txt path/to/file2.txt

* **By Pattern**: Use wildcards or named sets of files to add multiple files at once.

git add \*.txt

* **Interactively**: Launch a prompt where changes can be reviewed and selected for staging.

git add -i

**git rm and git mv**

These commands remove and rename files respectively. Changes are then staged and committed similarly to new additions.

**git reset**

Unstage files that were mistakenly added to the staging area.

git reset file1.txt

**git checkout**

To remove changes in the working directory, bring them back to the state of the HEAD commit.

git checkout file1.txt

**git commit**

This command is used to save changes to the local repository after they have been staged.

Parameters (optional):

**-m**

Use a one-line message to describe the commit.

git commit -m "Add new feature xyz"

**-a**

Stage all modified files and commit them.

This command omits the staging step, effectively combining git add and git commit. Since it bypasses the staging area completely, care should be taken when using it, to ensure only intended changes are included in the commit.

git commit -am "Update feature xyz"

When using -a without -m, Git will open the default text editor for you to compose the commit message in a separate file.

**4. Excluding Files from a Commit**

Some files might not need to be committed, for example:

* **Temporary Files**: This might include build artifacts or editor-specific temporary files.
* **Sensitive Information**: Such as passwords or private keys.

For such exclusions, you can define the file patterns or names in a .gitignore file.

To exclude files globally, create or edit the global .gitignore file.

git config --global core.excludesfile ~/.gitignore\_global

Then, ensure to add a .gitignore file in the root directory for the local repository if needed.

touch .gitignore

**CAUTION:**

* Avoid tracking, staging, or committing sensitive information.
* Keep .gitignore up to date, especially in collaborative projects.

**15. What information is contained in a *commit object* in *Git*?**

The **commit object** in Git encapsulates rich textual data that aids in tracking and managing changes in the repository.

**Core Elements in a Commit Object**

* **Author and Committer**: These metadata entries contain the individual's name and **email** address as well as the timestamp.
* **Commit Messages**: They provide a brief yet comprehensive understanding of the changes introduced in the commit, helping in maintaining clarity and context.
* **Parent Commits**: This represents a pointer to one or more parent commits. In the case of a merge commit, there can be more than one parent.
  + The HEAD of most branches points to the most recent commit, and this reference is what progresses as newer commits occur. The HEAD itself points to the most recent commit but operates **indirectly** through the branch reference.