Tasks Assigned for today

06-03-2025
Suneetha.V

1. **Clear explanation of Git 2 Types(CVCS, DVCS)**
2. **Git bash, github**
3. **Protocols(HTTPS, SSH)**
4. **Real Time Environment(Development, QA, UAT, Production)**
5. **Git Commands**

1. Centralized Version Control System (SVN)

In a **Centralized Version Control System (CVCS)** like **SVN**, there is a **central repository** where all project data is stored. Developers **check out** the code to their local machines, make changes, and then **commit** the changes back to the central repository. Here's how it works:

- **Single Source of Truth**: The code repository is centralized, and there is only one version of the project.
- **Checkout and Update**: Developers checkout the latest version of the code, make changes, and then update the central repository after committing their changes.
- **Real-Time Collaboration**: SVN allows multiple developers to work on the same codebase at the same time, but it requires frequent updates and commits to avoid conflicts.
- **Version History**: The history of changes is stored in the central repository, allowing developers to access previous versions.

Advantages of CVCS:

1. **Simplicity**:
    - Easier to understand and use for small teams or less complex projects.
    - The central repository structure is straightforward and familiar to many.
2. **Centralized Control**:
    - Administrators have more control over the repository. You can manage permissions, user access, and ensure that developers only modify what they are authorized to.
    - Easier to enforce code review processes since everything is managed in a central location.
3. **Simpler Backup and Recovery**:
    - As there is a single central repository, it's easier to back up, restore, and manage the version history from one location.
4. **Easier to Manage Large Files**:
    - SVN and other CVCSs may handle large binary files more effectively compared to some DVCSs.
5. **Consistency**:
    - Since all team members are working off of the same central repository, there is always a single version of the truth, avoiding issues with inconsistencies across multiple copies.

Disadvantages of CVCS:

1. **Single Point of Failure**:
    - If the central server goes down, no one can commit new changes or retrieve data. This can halt development for the entire team.

2. **Limited Offline Capability**:
    - Developers cannot work offline. If they want to commit their changes, they must be connected to the central server. Local changes can't be committed until they reconnect.
3. **Performance Bottlenecks**:
    - For large teams, having a single central server can lead to performance issues. Each action, like committing changes, checking out files, or updating, requires communication with the server.
4. **Scalability Issues**:
    - As the project and team grow, managing and scaling a centralized repository can become challenging, especially for large open-source projects.
5. **No Local History**:
    - Developers only have access to the current version and can’t access the full history of changes locally without connecting to the central server.

2. Distributed Version Control System (DVCS) - Git

In a **Distributed Version Control System (DVCS)** like **Git**, every developer has a full copy of the project repository, including its history, on their local machine. They can make changes locally, and later push those changes to a shared repository (such as GitHub, GitLab, or Bitbucket). Here’s how it works:

- **Multiple Copies of Repository**: Each developer has a full copy of the entire repository, including all branches, commits, and history.

- **Local Commits**: Developers can commit changes locally and work offline without needing to access the central repository.
- **Branching and Merging**: Git makes branching and merging easy, allowing developers to create feature branches, experiment, and then merge back into the main codebase.
- **Pull Requests**: Changes are pushed to a remote repository, and other team members review and approve changes before they are merged into the main branch.

Advantages of DVCS:

1. **Offline Work**:
    - Developers can commit, branch, and work on their code even when they are not connected to the central repository, offering much more flexibility in terms of remote work.
2. **No Single Point of Failure**:
    - Since every developer has a complete copy of the repository, there is no dependency on a central server. If the remote repository goes down, developers can still work on their local copies and push updates when it becomes available again.
3. **Faster Operations**:
    - Since operations (commits, branching, merges) are done locally on the developer's machine, they tend to be faster than in CVCS. There's no need to wait for server communication, which can be especially beneficial for large projects.
4. **Branching and Merging**:

- Git and other DVCSs are designed to make branching and merging easy and efficient. Developers can create feature branches, experiment, and then merge changes back into the main project without disrupting the main codebase.
- Excellent for parallel development, supporting workflows like **feature branching**, **GitFlow**, and **pull requests**.

5. **Redundancy and Safety**:
   - Since every user has the full project history, the repository is inherently backed up across multiple systems, reducing the risk of losing data. Even if the central server crashes, any developer can restore the repository.
6. **Better Collaboration**:
   - Git is excellent for teams, especially distributed teams, because changes are synchronized with a remote repository, and collaboration is done using mechanisms like pull requests, forks, and reviews.
7. **Efficient Handling of Large Projects**:
   - Git is optimized for handling large codebases and repositories with lots of branches. It scales better in terms of performance as compared to centralized systems like SVN.

Disadvantages of DVCS:

1. **Complexity**:
   - Git is much more complex to learn and use than a CVCS. It has many commands, options, and workflows that can be difficult for beginners to master.

- While Git offers powerful features, it also comes with a steeper learning curve, especially for users unfamiliar with distributed systems.

2. **Storage Overhead**:
   - Since every user has a full copy of the entire repository (including history), this can lead to larger storage requirements on each user's machine, especially with large projects. In some cases, this overhead may be undesirable, especially when working with binary files.
3. **Merge Conflicts**:
   - While Git handles merges well, complex branches and changes can lead to merge conflicts. Resolving these conflicts can be time-consuming and requires careful attention to detail.
4. **Distributed Workflow Management**:
   - In large teams, managing multiple branches and workflows can become complex. Issues like maintaining consistency across repositories and ensuring everyone is on the same page can sometimes be challenging without a clear workflow (e.g., GitFlow).
5. **Backup Management**:
   - While the distributed nature is an advantage for redundancy, it also means that backups are not centralized, which can make managing and restoring backups more complicated, especially for large teams and repositories.
6. **Confusing Concepts for Beginners**:
   - Concepts like **staging area**, **commit**, **push**, **pull**, **rebase**, and **merge** can be confusing for new users. Understanding the flow of work in Git requires learning and practicing.

3. Git Bash, GitHub

Git Bash is a command-line tool that allows you to use **Git** (the version control system) on your computer.

- **What is Git Bash?**
  - It's a program that gives you a **command line interface** (CLI), where you can type commands to interact with Git.
  - It **emulates a Unix shell** (a type of command-line interface) on your Windows computer, which makes it behave like a Linux or macOS terminal, even if you're using Windows.
- **Why do you need Git Bash?**
  - **Git Bash** helps you use Git commands on Windows, which are normally designed for Unix-based systems (Linux/macOS). It makes managing your projects (tracking changes, collaborating with others, etc.) easier, using a simple set of commands.
- **Key Functions:**
  - You can **clone repositories**, **commit changes**, **push/pull code**, and **create branches**, all from Git Bash.
  - For example, you can use commands like:
    - `git clone` — to download a project to your computer.
    - `git add` — to add changes to your project.
    - `git commit` — to save those changes.
    - `git push` — to upload changes to a remote server.

GitHub:

GitHub is an online service that hosts Git repositories (your projects or code) on the cloud.

- **What is GitHub?**
  - It's a website where you can store and manage Git repositories (code projects).
  - GitHub allows you to **share your code with others** and **collaborate** on projects.
- **Why do you need GitHub?**
  - **GitHub** allows multiple developers to work together on the same project. You can see the changes other people make, review code, and contribute to a project.
  - It provides a **centralized place** to store your project code, which makes it easier to **collaborate**, **version control**, and **backup** your work.
- **Key Functions:**
  - **Repository Hosting**: GitHub lets you store your project code, including its history, online.
  - **Collaboration**: Other developers can "fork" (copy) your project, make their own changes, and then propose merging those changes back into your project (via **pull requests**).
  - **Issues and Discussions**: You can manage bugs, feature requests, and ideas through **issues** and **discussions**.

- **Visibility and Social**: GitHub lets others find your code, follow you, and collaborate. It's often used to share open-source projects with the world.

4. Protocols:(HTTPS, SSH)

1. HTTPS (HyperText Transfer Protocol Secure)

HTTPS is a secure version of **HTTP** (the protocol used for transferring data over the web). It is the most common way to communicate with GitHub and other repositories.

- **How it works**:
  - HTTPS uses the **SSL/TLS protocol** to encrypt the communication between your local machine and the remote server (like GitHub).
  - When you perform actions like cloning a repository, pulling code, or pushing changes to GitHub, the communication happens over HTTPS.
- **How to use HTTPS in Git**:
  - When you clone a repository using HTTPS, the URL looks like this: `https://github.com/username/repository.git`
  - You will typically need to provide your **GitHub username** and **password** (or **personal access token** if two-factor authentication is enabled) each time you push or pull code.

SSH (Secure Shell)

SSH is a cryptographic network protocol that allows you to securely access a remote machine and perform tasks like transferring files or managing code repositories. SSH keys are used to authenticate you when interacting with Git repositories.

- **How it works**:
    - SSH allows you to authenticate with **SSH keys** (pairs of public and private keys), rather than a password.
    - The **public key** is added to your GitHub account, and the **private key** is stored securely on your local machine. When you push or pull from GitHub, SSH uses these keys to authenticate you, without needing a password.
- **How to use SSH in Git**:
    - When you clone a repository using SSH, the URL looks like this: `git@github.com:username/repository.git`
    - To push or pull code, your machine will authenticate automatically using your SSH keys.

5. Real Time Environments

1. Development (Dev) Environment

What it is:

- The **Development environment** is where developers work on writing and modifying code. It is the first stage of the software development lifecycle.

Purpose:

- **Code writing**: Developers create new features, fix bugs, and make changes to the application.
- **Local testing**: Developers test their code locally to ensure it works as expected before sharing it with others.

Characteristics:

- **Personalized for developers**: Each developer typically has their own isolated development environment (like a local machine or a container).
- **Frequent changes**: The code here changes very frequently, as developers work on new features, bug fixes, or improvements.
- **Low stability**: Since code is actively being written and tested, the environment is subject to frequent breaks or issues.

Tools often used:

- **IDE/Editor**: Visual Studio Code, IntelliJ IDEA, etc.
- **Local Database**: MySQL, PostgreSQL, MongoDB (locally set up versions).
- **Version Control**: Git (local repositories, branches).

2. Quality Assurance (QA) Environment

What it is:

- The **QA environment** is where the **Quality Assurance (QA)** team performs testing to ensure the application meets its functional and non-functional requirements.

Purpose:

- **Testing**: QA engineers test the application for bugs, issues, and user experience problems.
- **Bug fixing**: QA reports issues back to the development team, who then fix the problems in the development environment.

Characteristics:

- **More stable than Development**: While the code is still being tested, it's generally in a better state than in the Development environment.
- **Testing focus**: This environment focuses on testing functionality (does it work?) and performance (does it scale under load?).
- **Automated and manual testing**: A combination of **automated tests** (unit tests, integration tests, etc.) and **manual testing** is performed.

Tools often used:

Test management tools: Jira, TestRail.

- **Test automation tools**: Selenium, JUnit, Cypress.
- **Bug tracking systems**: Jira, Bugzilla.

3. User Acceptance Testing (UAT) Environment

What it is:

- The **UAT environment** is where the **User Acceptance Testing (UAT)** process occurs. This environment simulates the real-world environment in which the application will run, and end users or stakeholders validate whether the software meets business requirements.

Purpose:

- **Final validation**: Before the code is released to production, UAT allows the business or product owners to validate if the software meets their expectations.
- **Real-world conditions**: This environment often mirrors the **Production environment** as closely as possible.

Characteristics:

- **Realistic testing**: UAT involves testing the application with the **end-user perspective** in mind.
- **Business-driven**: It focuses on ensuring the application aligns with the business needs and requirements.
- **Minimal changes**: The code here should be in a feature-complete state, with minimal changes or bugs.

Tools often used:

- **User feedback tools**: Surveys, feedback forms, or user interviews.
- **UAT test cases**: Written based on business requirements.

- **Bug tracking**: Similar to the QA environment, tools like Jira may be used to log and track issues.

4. Production (Prod) Environment

What it is:

- The **Production environment** is the live, customer-facing environment where the application is fully deployed and available for use.

Purpose:

- **Live usage**: This is the final stage where the software is actually used by real users. It's the environment where end-users interact with the application.
- **Stable and reliable**: The production environment must be stable, secure, and performant. It should be thoroughly tested and free from bugs, as issues in this environment directly affect end users.

Characteristics:

- **Highly stable**: Once the software is deployed here, it should be functioning without major issues.
- **Continuous monitoring**: Monitoring tools are often employed to keep track of the system's performance, uptime, and user activity.
- **Scalability**: The production environment should be able to handle the actual user load and traffic of the live application.

Tools often used:

- **Monitoring tools**: New Relic, Datadog, Grafana.
- **Deployment tools**: Jenkins, Docker, Kubernetes, CI/CD pipelines.
- **Performance monitoring**: Google Analytics, server logs.

6. Git Commands:

1. `git version`

What it does:

Shows the currently installed version of Git.

Usage:

```
git --version
```

Purpose:

- This command is helpful to verify if Git is installed on your system and to check its version.

Example output:

```
git version 2.34.1
```

2. `git init`

What it does:

Initializes a new **Git repository** in your current directory.

Usage:

```
git init
```

Purpose:

- Converts a directory into a Git repository, creating a `.git` directory where Git stores configuration files, history, etc.
- After running this, you can start tracking files with Git (e.g., adding, committing).

3. `git config`

What it does:

Configures Git settings like your name, email, and other preferences.

Usage:

```
git config --global user.name "Your Name"
git config --global user.email "youremail@example.com"
```

Purpose:

- **Set user details**: The `user.name` and `user.email` are required when committing changes, so Git can attribute commits to the correct user.
- You can also configure settings like default editor, line endings, and more.

Other examples:

Set your default editor:

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

View all configuration settings:

```
git config --list
```

4. `git pull`

What it does:

Fetches changes from a remote repository and **merges** them into your local branch.

Usage:

```
git pull origin main
```

Purpose:

- It fetches the latest commits from the specified remote (like GitHub or GitLab) and merges them into your current local branch.
- **`origin`** is the default name for the remote repository, and **`main`** is the branch you are pulling from.

5. `git push`

What it does:

Upload your local changes to the remote repository.

Usage:

```
git push origin main
```

Purpose:

- Send your committed changes from your local machine to a remote repository (like GitHub, GitLab, etc.).
- **`origin`**: The remote repository.
- **`main`**: The branch you're pushing to (you may use other branch names, like `develop` or `feature-branch`).

6. `git commit`

What it does:

Records your changes locally in the repository.

Usage:

```
git commit -m "Your commit message"
```

Purpose:

- After making changes to files, you use **`git add`** to stage them, and **`git commit`** to save those changes with a message describing what was changed.
- The **`-m`** flag allows you to specify a message directly in the command.

Example message:

```
git commit -m "Fix bug in user login form"
```

Note: You should always write meaningful commit messages that describe what the change does.

7. `git remote`

What it does:

Manages the remote repositories (where your code is stored online).

Usage:

```
git remote -v
```

Purpose:

- The command `git remote -v` shows the URL of your remote repository (e.g., on GitHub, GitLab).
- You can add, remove, or rename remote repositories.

Examples:

To add a remote repository:

```
git remote add origin
```

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

To change the URL of an existing remote:

```
git remote set-url origin
```

```
https://github.com/username/new-repository.git
```

8. `git add`

What it does:

Stages changes in your working directory for the next commit.

Usage:

```
git add filename
```

Purpose:

- **Staging files**: Before committing, you need to tell Git which changes to include in the next commit. The **`git add`** command stages files, making them ready to be committed.
- You can add individual files or all files with `git add .`