**CI / CD**

CI/CD stands for **Continuous Integration** and **Continuous Deployment/Delivery**. It’s a way to make the process of developing, testing, and delivering software faster and more reliable by using automation.

### How Does CI/CD Work?

Think of CI/CD as an **assembly line** for software:

**Continuous Integration (CI):**

* 1. **Developers write code** and add it to a shared codebase (like on *GitHub* or *GitLab*).
  2. Once the code is added, an **automated process**:
     1. Checks if the new code works with the existing code.
     2. Runs tests to catch bugs early.
  3. If everything looks good, the changes are merged into the main project.

**Continuous Delivery (CD):**

* 1. After the code is tested, it’s automatically **packaged and prepared** for release.
  2. The code is ready to be deployed, but it waits for **manual approval** before going live.

**Continuous Deployment (CD):**

* 1. The code is **automatically deployed** to production (live environment) as soon as it passes all tests.
  2. No human approval is needed.

### Why Use CI/CD?

* **Catch bugs early:** Automated tests run whenever code changes, so problems are fixed before they grow.
* **Fast delivery:** Code changes are quickly moved to production, reducing the time from idea to release.
* **Less manual work:** Automation handles repetitive tasks, like testing and deployment.
* **Consistent results:** Reduces human error and ensures smooth updates for users.

**Key Components of CI/CD**

* **Source Code Repository:** A version control system like GitHub, GitLab, or Azure DevOps to store and manage code.
* **Build Automation:** Tools like Azure Pipelines, GitHub Actions, Jenkins, or GitLab CI/CD build the codebase and check for issues.
* **Testing:** Automated tests (unit, integration, and end-to-end tests) ensure the reliability of code.
* **Deployment:** Code is deployed to environments like staging, QA, and production using automation tools.

### Example in Simple Words:

Imagine you’re baking cookies with friends:

1. Each person works on a different type of cookie dough (**writing code**).
2. Before mixing all the dough together, you check each one for quality (**testing in CI**).
3. If all dough looks good, you bake them and package them (**delivery in CD**).
4. You ship them to customers immediately without any extra checking (**deployment in CD**).

In software, CI/CD ensures that everything moves smoothly from "writing code" to "delivering it to users."

**Configuring CI/CD for a .NET Web API**

**1. Setting Up the Repository**

* Push the .NET Web API code to a repository on platforms like GitHub, Azure DevOps, or GitLab.

**2. Choosing a CI/CD Tool**

* Use a platform-supported CI/CD pipeline such as:
* Azure DevOps Pipelines: Recommended for .NET applications.
* GitHub Actions: Offers flexible workflows for CI/CD.
* Jenkins: An open-source automation server.
* GitLab CI/CD: A built-in solution with GitLab.

**3. CI Configuration**

* Create a pipeline file to define build and test steps
* This can be done using defining rules in YAML file for Azure Pipelines and for GitHub actions

**4. CD Configuration**

* Integrate deployment steps into the pipeline. Use tools like Azure Web Apps, AWS Elastic Beanstalk, or Docker containers to deploy the application.

Here's a simple explanation of the difference between **Git**and **GitLab**:

**Git:**

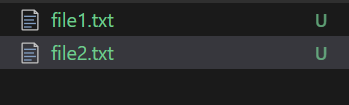
* **What it is:** Git is a version control system (**VCS**). It's a tool used to track changes in your code and collaborate with others.
* **What it does:** It helps you manage code versions, branch out for new features, and merge changes back into the main code.
* **Where it's used:** You use Git locally on your computer or with any server hosting service.
* **Example:** Think of Git as the engine that powers version control for your projects.

**GitLab:**

* **What it is:** GitLab is a platform built on top of Git. It's a web-based tool that provides additional features like project management, code hosting, CI/CD (Continuous Integration/Delivery), and collaboration tools.
* **What it does:** GitLab hosts your Git repositories online and adds features to plan, test, deploy, and manage your code collaboratively.
* **Where it's used:** It's used by teams or individuals who want a central place to store their Git repositories and access extra tools.
* **Example:** Think of GitLab as a garage that houses the Git engine and gives you extra tools to tune, manage, and monitor it.

git config --list : Lists all the configuration

git init -> Initialize a repository



U -> **untracked**

In Git, **untracked files** are files in your project folder that are **not yet being tracked by Git**.

This means:

* The files are **new** and haven’t been added to Git's version control.
* Git doesn't know about these files until you explicitly tell it to track them.

To start tracking them, you need to add them to the staging area by running:

**git add <file\_name>**

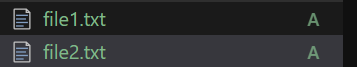
Git status -> will show the status of the files.

In Git (and in VS Code), the **"Added index"** refers to the state of files that have been staged, meaning they are added to the Git **staging area**. These are the files you have explicitly marked using git add and are now ready to be included in the next commit.

### Key Points:

**Staging Area**:

* 1. The "index" is another term for the staging area.
  2. Files in this state are tracked and prepared for a commit.



Once files are staged, create a commit with a meaningful message:

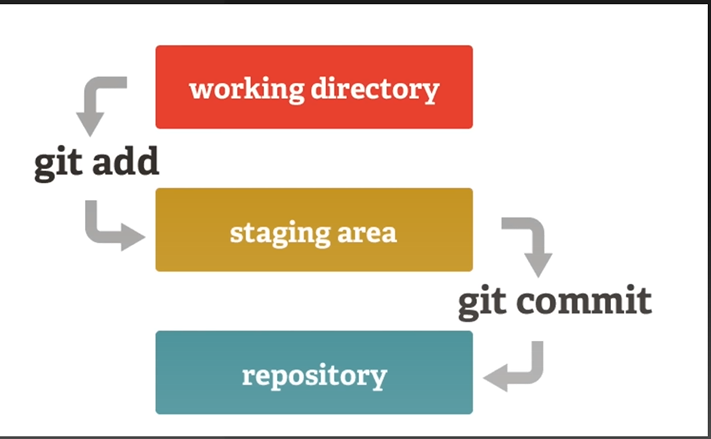
**Commit command**:

git commit -m "Your commit message here"

The command **git commit -am "message"** combines two actions: **adding changes** and **committing them** in one step.

### Breakdown:

* -a: Stages all modified and deleted files that are **already tracked** by Git (i.e., files previously added to Git with git add).
* -m: Specifies the commit message directly in the command.



**Steps to Push in Git**

**1. \*\*Ensure Your Local Repository Is Initialized:\*\***

- If your repository is not yet initialized, run:

git init

**2. \*\*Check the Status of Your Repository:\*\***

- View which files are tracked, untracked, staged, or unstaged:

```

git status

```

**3. \*\*Stage Your Changes:\*\***

- Add specific files to the staging area:

```

git add <filename>

```

- Or, add all changes:

```

git add .

```

**4. \*\*Commit Your Changes:\*\***

- Save the staged changes with a commit message:

```

git commit -m "Your commit message here"

```

**5. \*\*Set the Remote Repository (Only Once):\*\***

- Link your local repository to a remote repository:

```

**git remote add origin <remote-repository-URL>**

```

**6. \*\*Pull Latest Changes from Remote (Optional but Recommended):\*\***

- Ensure your local repository is synced with the remote branch to avoid conflicts:

```

git pull origin <branch-name>

```

**7. \*\*Push Changes to Remote Repository:\*\***

- Push committed changes to the desired branch:

```

git push origin <branch-name>

```

- Example for the `main` branch:

```

git push origin main

```

**8. \*\*Verify the Push:\*\***

- Check the remote repository (e.g., on GitHub or GitLab) to ensure the changes are updated.

**Additional Notes:**

- If you are working in a new branch, ensure you’ve created and switched to the branch:

```

git checkout -b <branch-name>

git push origin <branch-name>

```

- For repositories requiring authentication, ensure you provide credentials or set up SSH keys.

Here are the steps to perform a pull operation in Git, listed systematically:

1. Navigate to the Repository

Open your terminal or Git .

Navigate to the folder of the repository:

cd /path/to/your/repository

2. Check the Current Branch

Verify the branch you’re currently on:

git branch

The active branch will be marked with an asterisk (\*).

3. Fetch and Merge Changes

Use the pull command to fetch and merge changes from the remote branch into your local branch:

git pull origin <branch-name>

Replace <branch-name> with the branch you want to pull from (e.g., main, master, dev).

4. Resolve Conflicts (if any)

If there are merge conflicts, Git will pause the process and show you the conflicting files.

Open the conflicting files, resolve the conflicts, and save them.

Once resolved, stage the changes:

git add <conflicting-file>

Then, complete the merge:

git commit -m "Resolved merge conflicts"

5. Verify the Pull

Check the log to verify that the latest changes have been pulled successfully:

git log --oneline

Alternatively, you can check for updated files in your repository.

Alternative Commands

If you don’t specify a branch name, Git will pull changes from the branch you are currently on:

git pull

Important Notes

Ensure your local branch is clean (no unstaged or uncommitted changes) before pulling. Use:

git status

If you want to avoid automatic merges, fetch the changes first, then merge manually:

git fetch origin

git merge origin/<branch-name>

### What Are BLL and DAL in ASP.NET Web API?

**BLL (Business Logic Layer):**  
This is where the core business rules and logic are implemented. It acts as a bridge between the presentation layer (like controllers in Web API) and the data layer (DAL).  
Example: If your application requires a discount calculation or validation before saving data, it happens in the BLL.

**DAL (Data Access Layer):**  
This is where you interact with the database. The DAL contains methods for CRUD operations (Create, Read, Update, Delete) and ensures all database-related tasks are isolated from the rest of the application.  
Example: Querying user details or saving order details to a database.