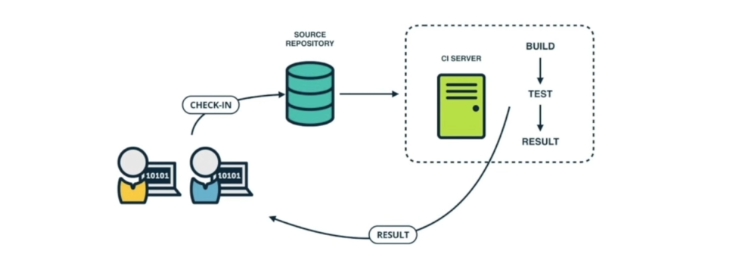
**Jenkins**

1. **Continuous Integration:**



Continuous Integration is a development practice that requires developers to integrate code into a repository several times a day.

Each check in is then verified by an automated build, allowing teams to detect problems easily.

If build is not green, system notify developer immediately. By this , developer can detect errors quickly , and locate them more easily.

1. **Stage of Adopting Continuous Integration.**

Continuous Integration is backed by several important principles and practices.

1. **Maintain a Single Source Repository**

Everyone on the team keeps their code in one central place, like a shared folder. This way, everyone works with the same code, and all changes are stored in one location. It helps everyone stay on the same page and reduces confusion about where the latest code is.

1. **Automate the build**

Instead of manually checking if the code works (like compiling it), a computer does it automatically every time someone changes the code. This is called "building" the code. It saves time and quickly catches mistakes, so they can be fixed before they cause bigger problems.

1. **Make your code self-testing**

Whenever the code is built, it automatically runs tests to check if everything still works as expected. This helps catch bugs early, so new code doesn’t break existing features.

1. **Make It Easy for Anyone to Get the Latest Executable Version**

The latest version of the software (like a ready-to-use app) is always available for anyone on the team to download and use. It makes it simple for team members to test, use, or show the latest version of the software without waiting for someone to prepare it.

1. **Everyone Can See What’s Happening**

Everyone on the team can easily see if the code is working, if the tests passed, and if there were any issues. It keeps the team informed about the project’s status, helps catch problems early, and promotes teamwork.

1. **How to do it**
2. **Developers Check Out the Code into Their Own Workspaces**

Each developer makes a copy of the code from the central repository to work on their own computer. This is called "checking out" the code. It allows developers to work on their own tasks independently without affecting the main codebase.

1. **When Done, Commit the Changes to the Repository**

After making changes to the code (like adding features or fixing bugs), the developer "commits" these changes back to the central repository. Committing changes regularly ensures that everyone has access to the latest code and reduces the chances of conflicts when merging different pieces of work.

1. **CI Server Monitors the Repository and Checks Out Changes When They Occur**

A CI server (like Jenkins) constantly watches the central repository. When it notices new changes, it automatically pulls those changes (checks them out) to start the CI process. This ensures that any new changes are immediately tested and integrated, so issues can be identified quickly.

1. **CI Server Builds the System and Runs Unit and Integration Tests**

The CI server compiles the code to make sure it works and then runs automated tests. These tests check if the individual parts (unit tests) and combined parts (integration tests) of the software work correctly. This step ensures that the code is always in a working state and that new changes don’t break existing functionality.

1. **CI Server Releases Deployable Artifacts for Testing**

If the build and tests are successful, the CI server packages the software into a "deployable artifact" (like an executable file or a Docker image) that can be used for further testing or deployment. This makes it easy to deploy the latest version of the software to a testing environment or even to production, ensuring that the software is always ready for release.

1. **CI Server Assigns a Build Label to the Version of the Code It Just Built**

The CI server gives this version of the software a unique name or "build label," like a version number. This helps keep track of different versions of the software.

1. **CI Server Informs the Team of the Successful Build**

The CI server lets the team know that everything went well with the build, usually by sending a message through email, Slack, or another communication tool.

1. **If the Build or Tests Fail, the CI Server Alerts the Team**

If something goes wrong—like the code doesn’t build correctly or a test fails—the CI server immediately alerts the team. This way, everyone knows there’s an issue that needs fixing.

1. **Continuous Deployment**

**Continuous Delivery** is a method used in software development to make sure that code changes are delivered smoothly and quickly to users. Here’s a simple breakdown:

* 1. **Code Changes:** Developers write and update the code.
  2. **Automated Build:** The code is automatically built and packaged into a deployable version.
  3. **Testing:** The new version of the code is automatically tested to ensure it works correctly.
  4. **Staging Environment:** The code is first deployed to a staging environment, which is a replica of the production environment. This is where Quality Assurance (QA) and other reviews happen.

**Example:** Before a new feature or fix is released to everyone, it’s tested in a staging environment to make sure it doesn’t break anything.

* 1. **Approval:** After testing in staging, if everything looks good, the code is approved for release.
  2. **Production Environment:** Finally, the code is deployed to the production environment where users can see and use the changes.

1. **Jenkins**

Jenkins is an open-source automation server widely used for continuous integration (CI) and continuous delivery (CD) in software development. It helps automate the parts of software development related to building, testing, and deploying, facilitating a CI/CD pipeline.

It is written in java

It is an automated tool, used to build and deliver the software product

Jenkins is a widely used application around the world that has around 300k installations and growing day by day.

**Why Jenkins?**

* Code is built and test as soon as developer commits code. Jenkins will build and test code many times during the day.
* On successful build, jenkins will deploy the source into the test server and notifies the deployment team.
* On build failures, jenkins will notify the errors to the developer team

1. **Installing Jenkins On linux/Unix Cloud Machine**

Java must be install on host machine

Make sure Java 8 is installed **(Now you can use java 17 version too for jenkins installation)**

Verify java version: **java - -version**

**Steps**

1. **Install Java to execute Jenkins war files**

Sudo add-apt-repository ppa:webupd8team/java

Sudo apt-get update

Sudo apt install openjdk-8-jdk

1. **Install Jenkins on Machine**

wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key |sudo gpg --dearmor -o /usr/share/keyrings/jenkins.gpg

1. sudo sh -c 'echo deb [signed-by=/usr/share/keyrings/jenkins.gpg] http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'
2. sudo apt-get update
3. sudo apt-get install Jenkins

**(if youre using your linux unix on your local machine you can run Jenkins on localhost:8080**

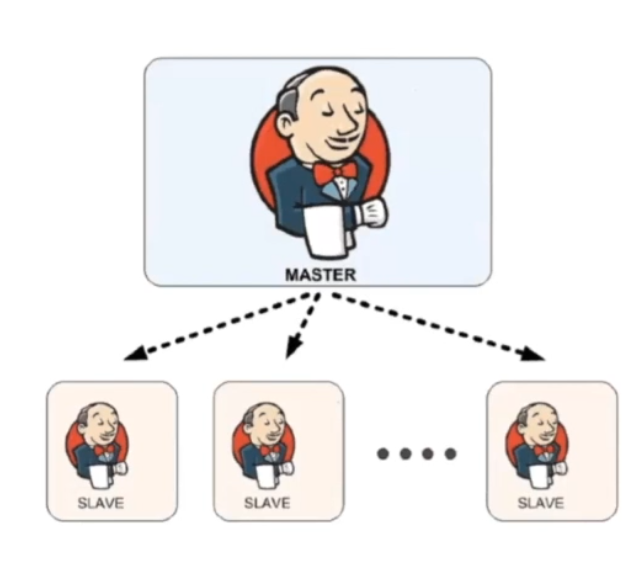
**If youre using the cloud machine then you need to find the ip of this machine and access it with the ip)**

**(If youre using aws machine you need to open the inbound traffic with the help of your security group)**

1. ip addr show

(get the id and use it in web browser. Suppose ip address is 192.168.0.210

**192.168.0.210 :8080** )

1. cat <the path given to unlock Jenkins> **(to get the initial admin password)(use the password to login)**
2. Install suggested plugins
3. Create first admin user(Make sure this account will be your admin account)
4. **Jenkins Master and slave Architecture**

**Jenkins Master(Controller)**

1. **Scheduling Build Jobs**

The Jenkins Master is like a project manager. When you create a new task (a "build job"), the Master is responsible for organizing when and how that task will be done. It keeps track of all the tasks and decides the best time to start each one.

1. **Dispatching Builds to the Slave**

Once the Jenkins Master has scheduled a task, it doesn’t do the work itself. Instead, it finds a worker (called a "Slave" or "Agent") to do the job. The Master sends the task to the Slave, which is like assigning the work to an employee who has the right tools and skills to complete it.

1. **Monitoring the Slave and Recording the Results**

While the Slave is working on the task, the Jenkins Master keeps an eye on it to make sure everything is going smoothly. Once the job is done, the Slave reports back to the Master with the results. The Master then records these results so that you can review how the task went, whether it was successful or if there were any issues.

**Jenkins Slave(Agent)**

The Jenkins Slave’s main job is to do the work assigned to it by the Jenkins Master. It doesn’t decide what work to do on its own; it waits for the Master to give it a task.

1. **Jenkins Jobs**

In Jenkins, a **job** (or sometimes called a **project**) refers to a task or set of tasks that Jenkins will perform. This could be something simple, like compiling code, or something more complex, like running a full software build, testing, and deployment pipeline.

**Slave agents on slave**

**"Slave Agent on Slave"** refers to the setup where a Jenkins Slave (Agent) machine runs another Jenkins agent within itself.

* **Jenkins Slave Agent**: A small program running on a Slave machine that allows it to receive and execute jobs from the Jenkins Master.
* **Job Distribution**: The Jenkins Master assigns jobs to available Slaves to balance the workload and execute tasks.
* **Node**: Any machine in Jenkins, including both the Master and all Slaves, that is part of the Jenkins system.

**Jenkins** **Executor**: In Jenkins, an **executor** is a component that handles the execution of jobs on a Jenkins Slave (or Master, if it's running jobs directly). Think of an executor as a worker on a Jenkins Slave machine that runs the build tasks or jobs. Each executor can run one job at a time. If you have multiple executors on a Slave, that Slave can run multiple jobs simultaneously.

**Jenkins Plugins:**

Jenkins plugins are modules that can be installed to add new capabilities to the Jenkins server. They help Jenkins integrate with various tools and services used in software development, testing, deployment, and more.

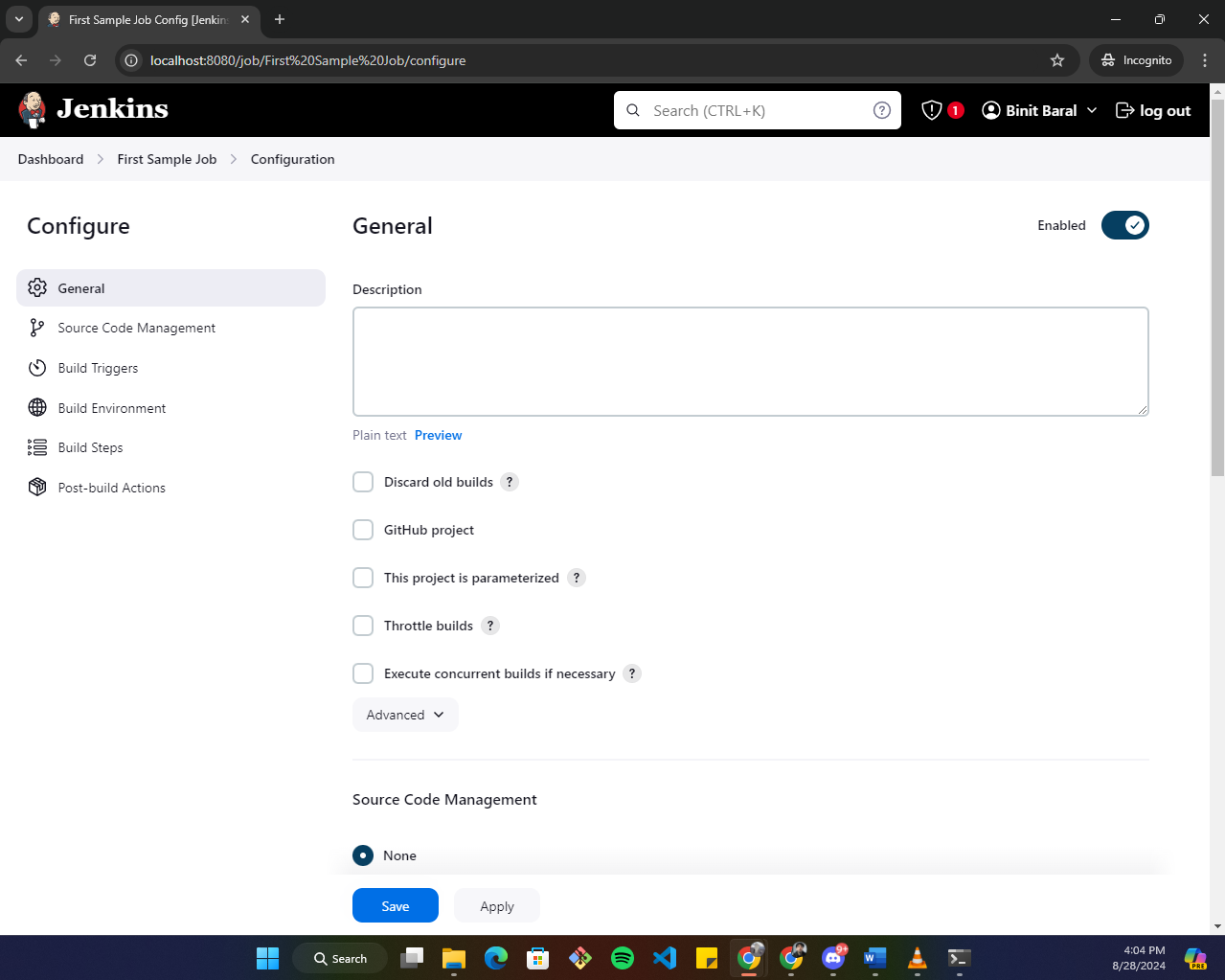
1. **Creating Jenkins job**
2. **Create job**
3. **Select freestyle project**

* Freestyle Project: A basic job type where you can define simple build steps like compiling code, running tests, and deploying applications.
* Pipeline: A job type that allows you to define your entire build process as code using a script, making it easier to automate complex workflows.
* Multi-Configuration Project: A job type used to run builds in different environments or configurations, like testing on multiple operating systems or Java versions.
* Folder: A way to organize your jobs and other items into folders, making it easier to manage large numbers of projects.
* Multibranch Pipeline: A job type that automatically creates pipelines for each branch in your version control system, allowing you to run builds and tests on multiple branches simultaneously.
* Organization Folder: A job type that scans your organization’s version control repositories and automatically creates jobs for all the projects, managing them under one folder.

1. **Configuration**

* General:
* Source Code Management:
* Build Triggers:
* Build Environment: Configuration options that prepare the environment before the build starts, like setting up variables or cleaning the workspace.
* Build Steps:
* Post-Build Actions: Actions that Jenkins should take after the build is completed, like sending notifications, archiving artifacts, or triggering other jobs.

**Parameters:** In Jenkins, **parameters** are variables or inputs that you can define for a job, allowing you to customize how the job runs each time. They let you pass different values or options into the job when it starts, which can change its behavior based on those inputs.

**In General:** Basic settings for your Jenkins job, like naming the job and setting job descriptions or parameters.

* Discard Old Builds: Automatically delete old build data after a certain number of builds or days to save space
* GitHub Project: Link your Jenkins job to a specific GitHub repository, making it easier to track changes and view project information.
* This Project is Parameterized: Allows you to define parameters that can be customized each time the job is run, like input values or options for the build.
* Throttle Builds: Limit the number of builds that can run simultaneously, either globally or per project, to avoid overloading resources.
* Execute Concurrent Builds if Necessary: Let Jenkins run multiple builds of the same job at the same time, useful if different builds don’t interfere with each other.
* Advanced Options: Additional settings for more specific configurations, like setting custom workspace locations or blocking builds based on certain criteria.

**In source code management:** Where you connect Jenkins to your version control system (e.g., Git, SVN) to pull the source code for the job.

* None: No source code management is used; the job doesn’t pull code from a version control system.
* Git: Connects Jenkins to a Git repository to pull source code for the job.

**In build triggers:** Settings that determine when and how your job should be automatically started, such as scheduling builds or triggering on code changes.

* **Trigger Builds Remotely**: Start a build by sending a request to Jenkins from an external system or service.
* **Build After Other Projects are Built**: Automatically start this job after one or more other specified jobs have finished.
* **Build Periodically**: Schedule the job to run at regular intervals, like daily or weekly, using a cron-like syntax.
* **GitHub Hook Trigger for GIT**: Automatically start a build when a change is pushed to a GitHub repository, using a webhook.
* **Poll SCM**: Regularly check the version control system for changes, and start a build if new changes are detected.

**In build Environment:** Configuration options that prepare the environment before the build starts, like setting up variables or cleaning the workspace.

* **Delete Workspace Before Build Starts**: Remove any existing files from the workspace before starting the new build to ensure a clean slate.
* **Use Secret Text(s) or File(s)**: Access and use sensitive information, like passwords or API keys, securely during the build process.
* **Add Timestamps to the Console Output**: Include timestamps in the build logs to see when each part of the build process occurred.
* **Inspect Build Log for Published Build Scans**: Check the build log for detailed reports or scans that were published as part of the build.
* **Terminate a Build if It’s Stuck**: Automatically stop a build if it’s taking too long or seems to be hanging.
* **With Ant**: Configure the build environment to use Apache Ant, a tool for automating build processes, if you are using it in your project.

**In build steps:** The actual tasks Jenkins will perform to build your project, like compiling code, running tests, or deploying the application.

**In Post-Build Actions**: Actions that Jenkins should take after the build is completed, like sending notifications, archiving artifacts, or triggering other jobs.