**Jenkins**

* Jenkins is an open source ci/cd tool written in java
* Jenkins is automation tool to build and deliver the software product
* Jenkins is widely used application around the world
* It has around 300k installations and growing day by day
* It is forked from another project Hudson after dispute with oracle
* It’s a server-based application and requires a web server like Apache Tomcat.

**Jenkins Installation on Ubuntu   
  
STEP\_1: Create a EC2 Machine in AWS (t2. micro)**   
  
SSH to EC2 Machine created above -> sudo apt update -y

**STEP\_2: Install Openjdk Java 17**

sudo apt install -y openjdk-17-jdk

**STEP\_3: Install Jenkins with bellow commands**

sudo wget -O /usr/share/keyrings/jenkins-keyring.asc https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key  
echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] https://pkg.jenkins.io/debian-stable binary/ | sudo tee /etc/apt/sources.list.d/jenkins.list > /dev/null  
sudo apt-get update  
sudo apt-get install -y Jenkins

**STEP\_4: Start Jenkins application suing systemctl**

**To start Jenkins service 🡪** sudo systemctl start jenkins

**To stop Jenkins service** 🡪 sudo systemctl stop jenkins

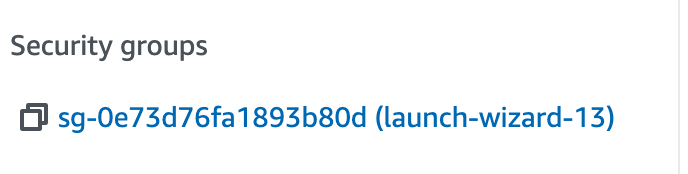
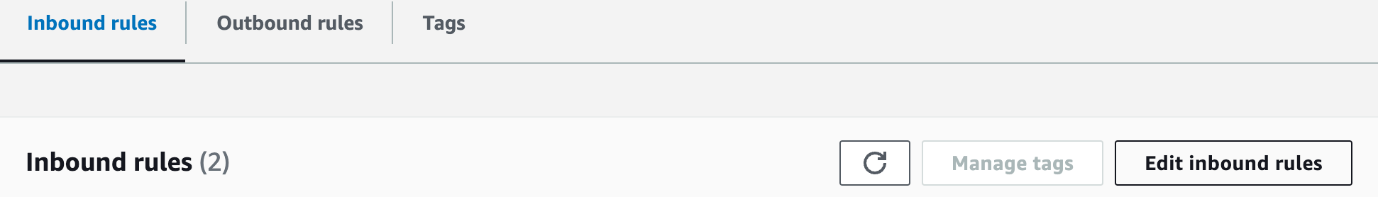
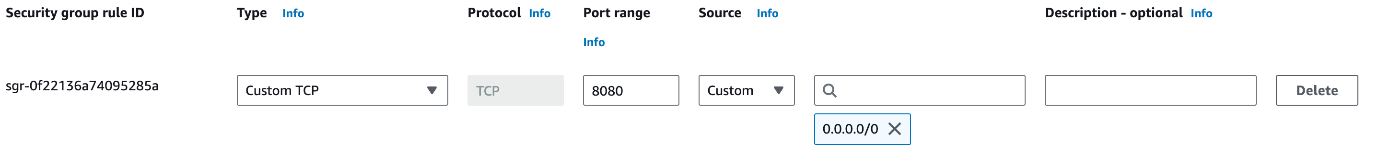
**To check current status Jenkins service 🡪** sudo systemctl status jenkins

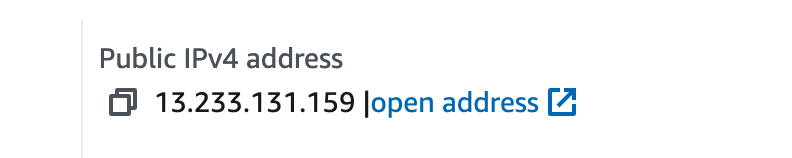
**To edit current Jenkins service settings**🡪 sudo systemctl edit jenkins

**STEP\_5: To access Jenkins from client (Browser)**

* Jenkins by default runs on 8080 port  
  To open 8080 port to access   
       Goto -> EC2 -> jenkins\_instance -> Security (Tab)

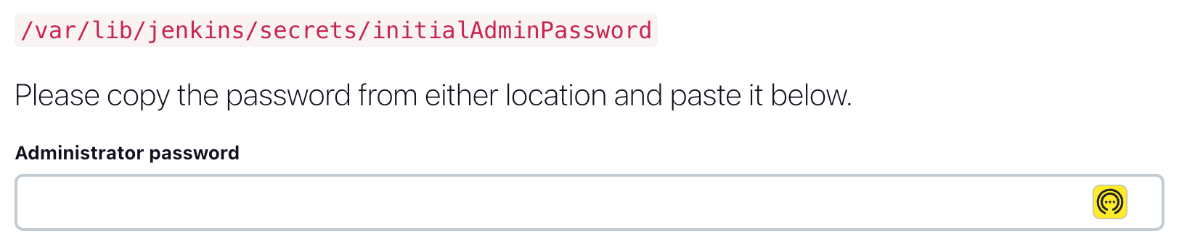


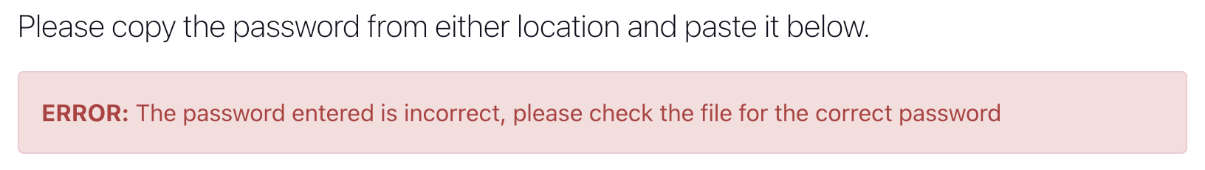
**Then**  
  
**Edit inbound rules**  
       
  
Add inbound rule for 8080   
  
     

**Access the jenkins with url <jenkins\_ec2\_public\_ip\_address>:<port> from any browser**        

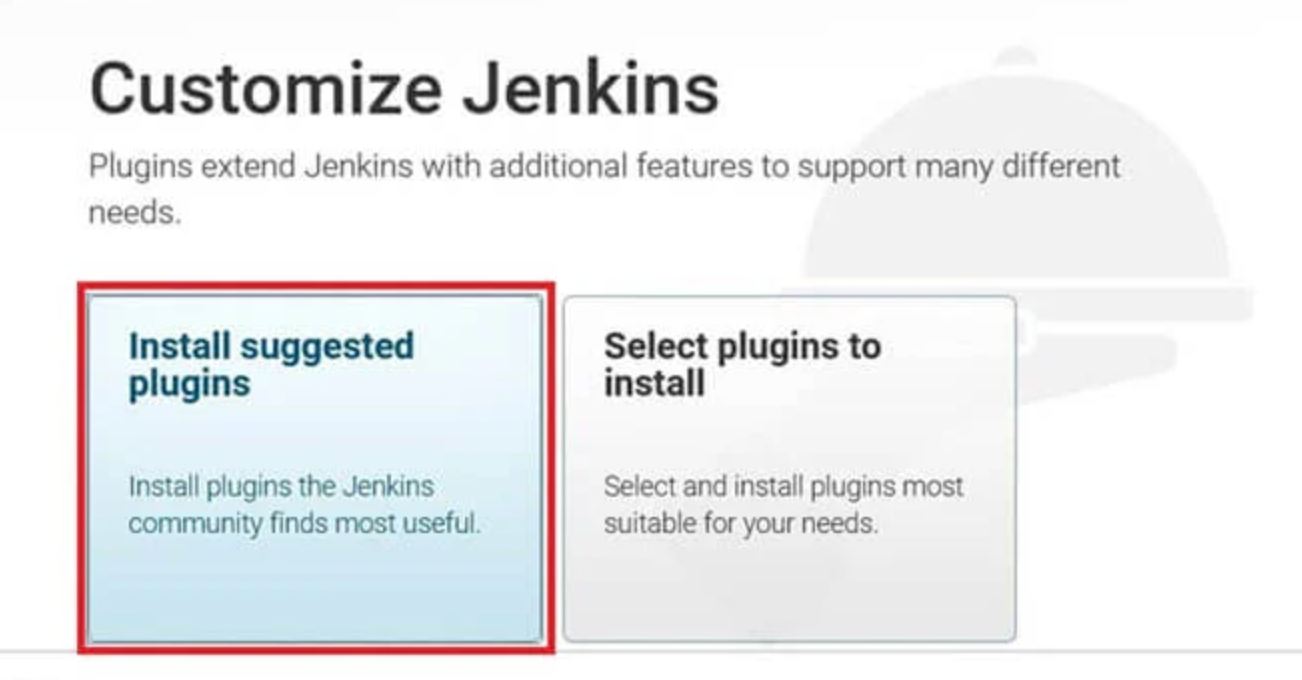
example: 13.233.131.159:8080

**STEP\_6: Initaial setup of jenkins (Unlock Jenkins)**

****  **Run the below command in jenkins EC2 machine**         
       sudo cat /var/lib/jenkins/secrets/initialAdminPassword  
         (Note: copy and paste printed key in the above windows)



* If you get above error means you copied some extra charactes with initial\_admin\_password in the terminal (like including new line)
* **Solution:  copy the password into any notepad application and copy only the password to jenkins**

****

**CI/CD**

* As soon as developer push changes into repo, jenkins pipeline will trigger build using build stages, binaries are generated. Continuous test will happen, test cases will done by selenium and JUnit. Code gets deployed into respective environment configuration.

**Continuous development:** Developers will plan and code for particular application.

**Continuous integration:** Continuous Integration (CI) is a practice where developers frequently integrate their code changes into a shared repository, often several times a day. Each integration is then verified by an automated build and automated tests, allowing teams to detect integration errors early and resolve them quickly.

**Continuous delivery**: Continuous Delivery is the practice of ensuring that code is always in a deployable state.

**Continuous Deployment (CD):** deployment of every code change that passes automated tests directly to production.

**ELASTIC IP:**

Dedicated public IP. We reserved public IP that is called elastic IP. If we are not using elastic IP they will put bill.

An Elastic IP (EIP) is a feature offered by Amazon Web Services (AWS) that provides static IPv4 addresses designed for dynamic cloud computing. Here are some key points about Elastic IPs:

1. **Static Public IPv4 Address**: An Elastic IP address is a static IPv4 address that you can allocate from AWS and associate with your AWS resources such as EC2 instances, NAT gateways, and certain types of load balancers.
2. **Usage**: Elastic IPs are particularly useful in scenarios where you need a persistent public IP address that can be quickly reassigned to another instance or resource in the event of instance failure, maintenance, or when scaling horizontally.
3. **Cost and Allocation**: While AWS allows you to allocate Elastic IPs for free as long as they are associated with a running instance, there is a charge if the IP is not associated with any running instance (idle or unused Elastic IPs).
4. **Benefits**:
   * **Static Addressing**: Avoids the need to update DNS records when instances are replaced.
   * **Quick Replacement**: Enables quick recovery and replacement of failed instances.
   * **Internet-Facing Applications**: Useful for applications requiring a consistent public IP address for clients to access.
5. **Association and Disassociation**: You can associate an Elastic IP with an instance by assigning it directly to the instance's network interface. It can also be disassociated and reassigned to another instance or resource within your AWS account.
6. **Limitations and Best Practices**:
   * AWS imposes limits on the number of Elastic IPs you can allocate per AWS account. It's a good practice to use Elastic IPs judiciously and release them when they are no longer needed to avoid unnecessary costs.
   * Always consider using AWS services like Load Balancers or AWS Elastic Beanstalk that automatically manage IP addresses for you, reducing the need for Elastic IPs.

**PLUGINS**

Plugins are essential components that extend jenkins functionality and provide additional features. Here's a brief overview of plugins in Jenkins:

1. **Purpose**: Plugins in Jenkins serve various purposes such as integrating with version control systems, enabling build tools, enhancing UI/UX, providing notifications, and more.
2. **Installation**: Plugins can be installed through the Jenkins web interface via the Plugin Manager. There are thousands of plugins available through the Jenkins Update Center.
3. **Types of Plugins**:
   * **Pipeline Plugins**: These plugins add functionality to Jenkins Pipeline, allowing for scripted and declarative pipelines.
   * **SCM Plugins**: Integrates Jenkins with various Source Code Management (SCM) systems like Git, SVN, Mercurial, etc.
   * **Build Tool Plugins**: Adds support for build tools like Maven, Gradle, Ant, etc.
   * **Notification Plugins**: Sends notifications through email, Slack, or other messaging systems.
   * **Monitoring Plugins**: Provides monitoring capabilities for Jenkins itself or the builds.
   * **Authentication and Authorization Plugins**: Enhances security by integrating with LDAP, Active Directory, etc.

**Top 10 plugins in jenkins**

1. Pipeline plugin
2. Git plugin
3. Git hub integration plugin
4. Junit plugin
5. Docker plugin (Deploy to container)
6. Credential plugin
7. SonarQube scanner plugin
8. Copy artifacts plugin
9. Amazon ec2 plugin
10. Email extension
11. Build plugin
12. Job DSL plugin
13. Maven integration plugin
14. Maven tomcat plugin
15. Cloudbee folder plugin
16. Ssh agent plugin
17. **Managing Plugins**: Once installed, plugins can be managed through the Plugin Manager. This includes updating, uninstalling, and configuring plugin settings.
18. **Plugin Configuration**: Each plugin typically has its own configuration options that can be accessed through Jenkins' web interface under the global or job-specific configuration pages.
19. **Custom Plugins**: Jenkins also allows users to develop custom plugins to cater to specific needs not covered by existing plugins.

**Authentication**

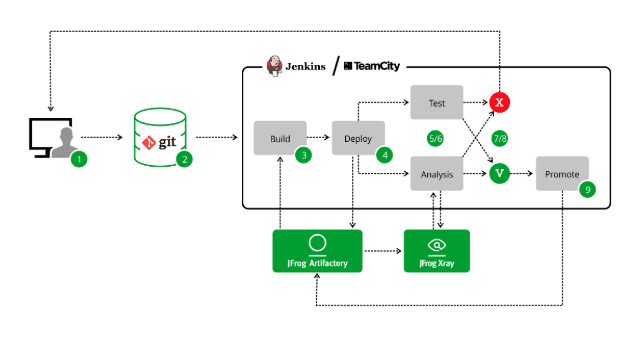
Authentication is the process of verifying the identity of a user or system attempting to access a resource. It ensures that the user is who they claim to be. Authentication typically involves presenting credentials, which can include:

* **Username and Password**: The most common form of authentication where a user provides a username and a secret password.
* **Biometric Data**: Such as fingerprints, retina scans, or facial recognition.
* **Token-based**: Where a temporary token or one-time password (OTP) is used for authentication.
* **Certificates**: Digital certificates that verify the identity of users or systems.

**Authorization**

Authorization occurs after authentication and determines what actions an authenticated user or system can perform on a given resource. It ensures that users have appropriate permissions to access specific resources or perform certain operations. Authorization is typically based on:

* **Roles**: Users are assigned roles (e.g., admin, user, manager), and permissions are assigned to these roles.
* **Permissions**: Granular permissions define what actions (read, write, delete) a user can perform on specific resources.
* **Policies**: Sets of rules or policies that govern access control based on various attributes (time of day, location, etc.).



**Add Users to jenkins**

Navigate to **Jenkins Dashboard** → **Manage Jenkins** → **Users** → **Create User**. Enter the user’s name, password, full name and email-address and then click **Create User** button.

**Credentials to Integrate with other tools**

Navigate to**Jenkins Dashboard → Manage Jenkins → Security → Credentials → (global) → Add Credentials**

* + For "Username with password", you'll need to enter the username and password.
  + For "SSH username with private key", you'll need to provide the username and paste the private key.
  + For "Secret text", you'll enter the secret value.
  + For "Secret file", you'll upload the file containing the secret.

**Select Credential Type**: Choose the type of credentials you want to add from the dropdown menu. Jenkins supports various types of credentials, such as username with password, SSH username with private key, secret text, secret file, etc. Select the appropriate type for your use case.

* **Save Credentials**: After entering the necessary details, click on the "OK" or "Save" button to add the credentials to Jenkins.
* **Use Credentials in Jobs**: Once the credentials are saved, you can use them in your Jenkins jobs and plugins. During job configuration, you'll have the option to select the appropriate credential from the list of available credentials.

**Add security**

Navigate to**Jenkins Dashboard → Manage Jenkins → Security → Project based matrix authorization → Add security**

**Project-based Matrix Authorization Strategy (User security and access control)**

Project-based Matrix Authorization Strategy allows you to define specific permissions for each project (job) within your Jenkins instance. With this strategy, you can specify which users or groups have particular permissions (such as build, configure, or delete) for individual projects.

Here's how you can set up Project-based Matrix Authorization in Jenkins:

* **Access Jenkins Configuration**: Log in to Jenkins with administrative privileges.
* **Navigate to Manage Jenkins**: Click on the "Manage Jenkins" link on the Jenkins dashboard.
* **Configure Global Security**: Under the "Configure Global Security" section, select the "Enable security" checkbox to enable security settings.
* **Select Project-based Matrix Authorization Strategy**: Scroll down to the "Authorization" section and choose "Project-based Matrix Authorization Strategy" from the available options.
* **Define Permissions**: You will see a matrix where you can define permissions for different users or groups. Here are some common permissions:
  + Overall: Permissions for overall Jenkins access.
  + Job: Permissions related to specific jobs or projects.
  + Run: Permissions to run jobs.
  + Configure: Permissions to configure jobs.
  + Read: Permissions to view job configurations and build results.
  + Workspace: Permissions to access the workspace of a job.
  + Cancel: Permissions to cancel builds.
  + Delete: Permissions to delete jobs.
* **Grant Permissions**: For each permission, you can specify which users or groups should have that permission by checking the corresponding checkboxes.
* **Save Configuration**: Once you have configured the permissions as per your requirements, click on the "Save" or "Apply" button to save the changes.
* **Test Permissions**: After configuring the permissions, it's essential to test them to ensure that users and groups have the appropriate access to Jenkins projects.

**Jenkins Freestyle Project/Job**

**General Configuration:**

* **Job Name:** Provide a meaningful name for the job to identify its purpose.
* **Description:** Optionally, add a description to provide additional context about the job's purpose.
* **Discard Old Builds:** Specifies the strategy for managing old builds to free up disk space  
  Options typically include keeping all builds, keeping only the last N builds, or keeping builds based on days.
* **Execute concurrent builds if necessary:**Allows Jenkins to execute multiple builds of this job concurrently if necessary.
* **This project is parameterized:** Allows you to define parameters for the job, enabling users to customize build behavior.  
  + Most used parameter types,  
    **String Parameter:**Accepts a single string value.  
    **Boolean Parameter:** Accepts a true/false value.  
    **Choice Parameter:**Presents a dropdown menu or radio buttons to select one option from a predefined list.

**Source Code Management (SCM):**

* **SCM:**Choose the version control system (e.g., Git) for your project.
* **Repository URL:** Provide the URL of your version control repository.
* **Credentials:** If required, provide credentials to authenticate with the version control system.

**Build Triggers:**

Specify conditions that trigger a build.

* **Build periodically:** Trigger builds based on a schedule (using cron syntax).
* **Poll SCM**: Trigger a build when changes are detected in the version control repository.
* **Build after other projects are built:** Trigger this job after other specified projects are built.
* **Trigger builds remotely:** Trigger builds remotely via HTTP.

**Build Environment:**

* **Build Environment:** Set up the environment for the build process.
* **Delete workspace before build starts:**Clean up the workspace before starting a new build.
* **Set Environment Variables:** Define environment variables for the build process.

**Build:**

* **Build:**Define the build steps for the job.
* **Execute shell:** Execute shell commands on Unix-like platforms.
* **Execute Windows batch command:** Execute batch commands on Windows.
* **Invoke Maven:** Execute Maven goals.

**Post-Build Actions:**

* **Post-Build Actions:** Define actions to be performed after the build completes.
* **Archive the artifacts:** Archive files (e.g., JAR, WAR) to keep them for future reference.
* **Email Notification:** Send email notifications to specified recipients after the build.
* **Trigger parameterized build:** Trigger another project with parameters.
* **Publish JUnit test result report:** Publish JUnit test results to Jenkins

**Crontab (Job Scheduler)**

* The crontab is a list of commands that you want to run on a regular schedule, and also the name of the command used to manage that list. Crontab stands for “cron table, ” because it uses the job scheduler cron to execute tasks; cron itself is named after “chronos, ” the Greek word for time. Cron is the system process which will automatically perform tasks for you according to a set schedule. The schedule is called the crontab, which is also the name of the program used to edit that schedule.

Linux Crontab Format:     \*         \*            \*          \*         \*              <command to execute>

       MIN    HOUR     DOM     MON     DOW                  CMD

FIELD DESCRIPTION ALLOWED VALUE

* MIN Minute field 0 to 59
* HOUR Hour field 0 to 23
* DOM Day of Month 1-31
* MON Month field 1-12
* DOW Day Of Week 0-6
* CMD Command Any command to be executed.

To view the Crontab entries: crontab -l

To edit Crontab Entries: crontab -e

To edit crontab entries of other Linux users: crontab -u username -e

example

1) **To schedule a job for every minute using Cron**

\* \* \* \* \* command/script

2) **How to Execute a Linux Cron Jobs Every Second Using Crontab.**

You cannot schedule an every-second cronjob. Because in cron the minimum unit you can specify is minute. 

3) **To schedule a background Cron job for every 10 minutes**.

\*/10 \* \* \* \* /home/maverick/check-disk-space

4) **Schedule a Job for More Than One Instance (e.g. Twice a Day)**

**executes the specified script at 11:00 and 16:00 every day**

00 11,16 \* \* \* /home/ramesh/bin/incremental-backup

5**) Schedule a Job for Specific Range of Time (e.g. Only on Weekdays)**

    This example checks the status of the database every day (including weekends) during the working hours 9 a.m – 6 p.m

00 09-18 \* \* \* /home/ramesh/bin/check-db-status

**6) Cron job to run on the last day of the month**

55 23 28-31 \* \* [[ "$(date --date=tomorrow +\%d)" == "01" ]] && myscript.sh 0 23 28-31 \* \* [ $(date -d +1day +%d) -eq 1 ] && myscript.sh

**7) Cron special keywords and their meaning Keyword Equivalent**

@yearly (0 0 1 1 \*)

@daily (0 0 \* \* \*)

@hourly (0 \* \* \* \*)

@reboot Run at startup.

a) **To schedule a job for the first minute of every year using @yearly**

@yearly /home/maverick/bin/annual-maintenance

b) **To schedule a Cron job beginning of every month using @monthly**

@monthly /home/maverick/bin/tape-backup

c) **To schedule a background job every day using @daily**

@daily /home/maverick/bin/cleanup-logs "day started"

d) **To execute a Linux command after every reboot using @reboot**

@reboot CMD

**PACKAGE MANAGER**

* Each and every OS have package manger
* To easy install, transfer and easy to deliver to customer we will do package
* **Build tools:** Automatically run any files and compiles it and build a package is called build tools
* Binary files of java code (application) file is **.class** that is after compile
* **Apache software:**  It’s a open source for build tool

Build tool to java is ANT

Upgraded tool to ANT is MAVEN (most popular)

Updated tool to ANT is GRADEL

**PATCH BUILDS:** Only changed codes will compile instead of every folder (all codes)

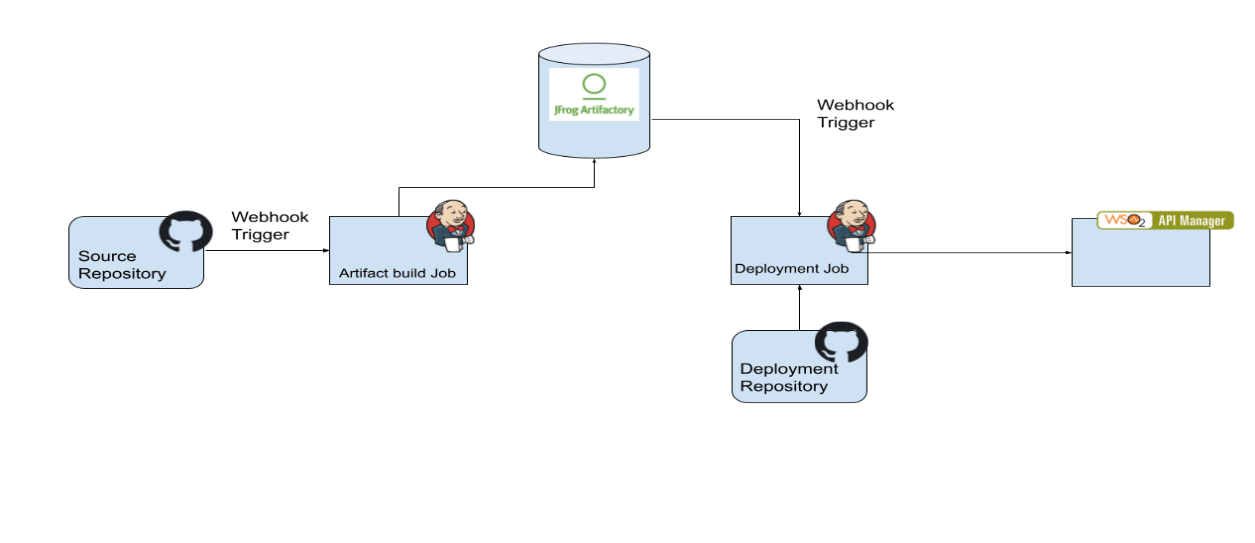
**LOAD / CLEAN BUILD:** It will run all the files to build not only changed one.

**BUILD TOOLS:**

**JAVA:** MAVEN and GRADEL

**C/C++/EMBEDDED:** Makefile

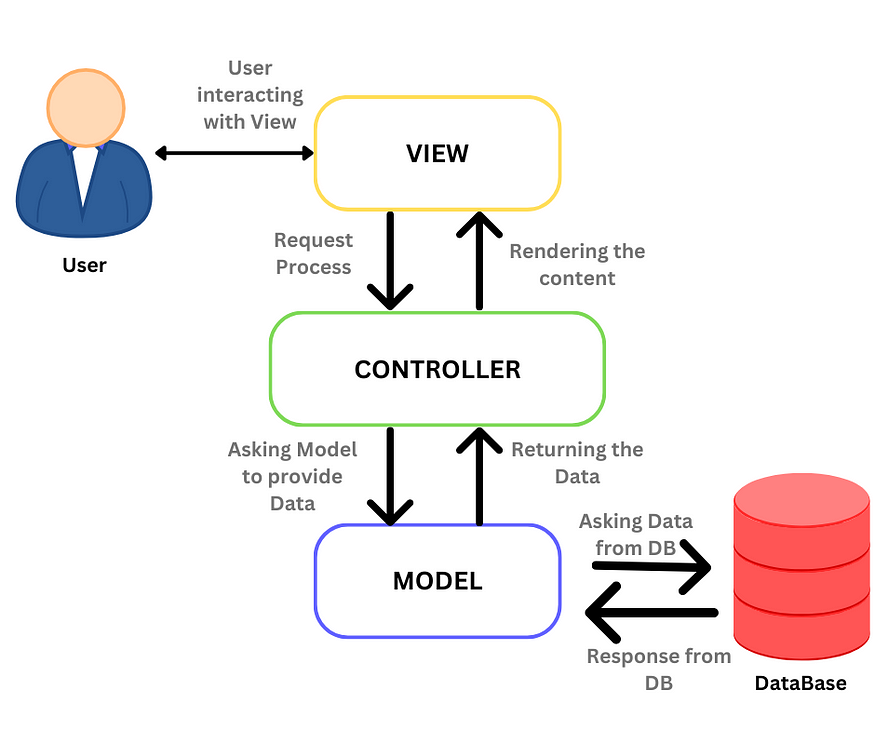
**JAVA SCRIPT BASED TECHNOLOGY:** NPM and YARN



* Artifactory stores the compile code or binary file (.exe .class)

**MVC MODEL**

* The MVC (Model-View-Controller) model is a software architectural pattern widely used in designing and developing interactive software applications. It separates an application into three interconnected components to achieve separation of concerns, modularity, and ease of maintenance. Here’s an overview of each component:



**View**

The **View** represents the presentation layer of the application. It displays the data from the Model to the user and handles user interactions. Views are typically user interfaces (UIs) such as web pages, forms, or reports that render data obtained from the Model.

* **Responsibilities**:
  + Presents data to the user in a specific format (HTML, XML, JSON, etc.).
  + Collects and validates user input.
  + Sends user actions (like button clicks) to the Controller for processing.
* **Example**: In the task management application, a View might be an HTML page that displays a list of tasks and provides forms for adding or editing tasks.

**Controller**

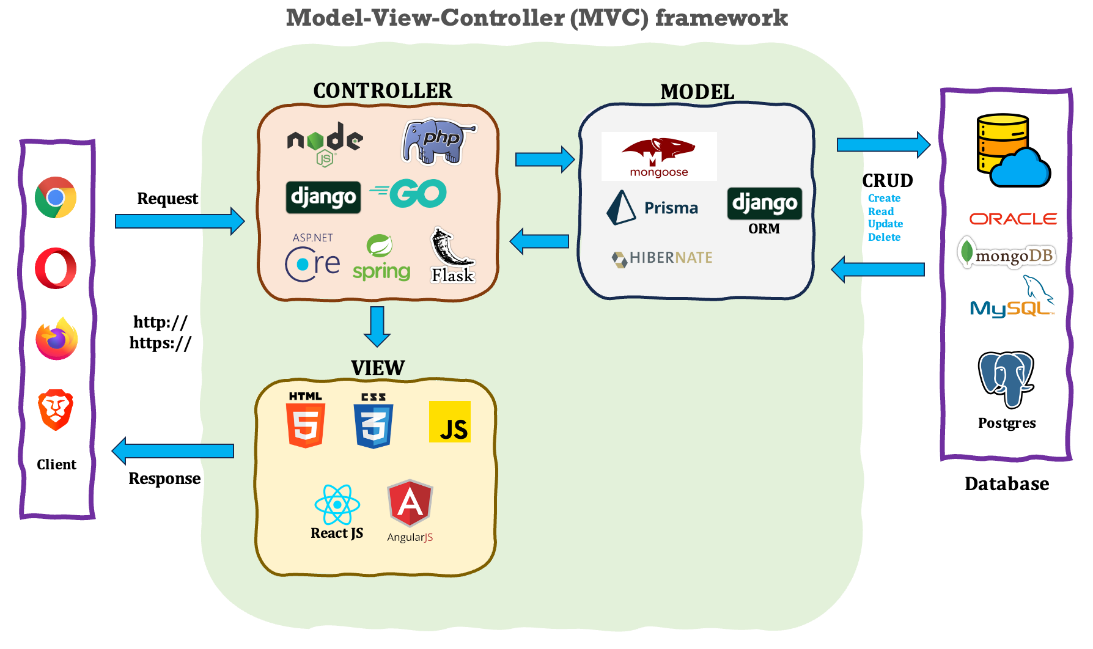
The **Controller** acts as an intermediary between the Model and the View. It receives user input from the View, processes it (possibly interacting with the Model), and decides which View to display to the user next. Controllers interpret user actions and map them to operations on the Model or View.

* **Responsibilities**:
  + Handles user input and initiates actions based on that input.
  + Updates the Model based on user actions (e.g., creating, updating, or deleting data).
  + Selects and renders Views to display to the user.
* **Example**: In the task management application, a Controller might receive a request to add a new task from the View, interact with the Model to create the task, and then return a response to update the View with the new task.

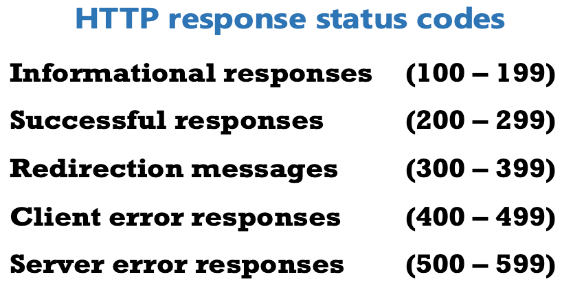
**Model**

The **Model** represents the data and business logic of the application. It encapsulates the application's data and provides methods to manipulate that data. In essence, the Model manages the state and behaviour of the application's domain, ensuring data consistency and validity.

* **Responsibilities**:
  + Manages the data and state of the application.
  + Implements business rules and logic.
  + Communicates with databases, APIs, or other data sources to retrieve or persist data.
* **Example**: In a web application for managing tasks, the Model might define classes like Task, User, and methods for CRUD (Create, Read, Update, Delete) operations on tasks stored in a database.

****

**ERROR CODES**



 1**xx (Informational):** Request received, continuing process.

* **100 Continue:** The server has received the request headers and the client should proceed to send the request body.
* **101 Switching Protocols:** The requester has asked the server to switch protocols.

 2**xx (Success):** The action was successfully received, understood, and accepted.

* **200 OK:** Standard response for successful HTTP requests.
* **201 Created:** The request has been fulfilled, resulting in the creation of a new resource.
* **204 No Content:** The server has successfully fulfilled the request and there is no additional content to send in the response.

 3**xx (Redirection):** Further action needs to be taken in order to complete the request.

* **301 Moved Permanently:** The requested resource has been permanently moved to a new URL.
* **302 Found (Previously "Moved Temporarily"):** The requested resource resides temporarily under a different URL.
* **304 Not Modified:** Indicates that the resource has not been modified since the version specified by the request headers.

 4**xx (Client Error):** The request contains bad syntax or cannot be fulfilled.

* **400 Bad Request:** The server cannot or will not process the request due to something that is perceived to be a client error.
* **403 Forbidden:** The server understood the request but refuses to authorize it.
* **404 Not Found:** The server cannot find the requested resource.

 5**xx (Server Error):** The server failed to fulfil a valid request.

* **500 Internal Server Error:** A generic error message, given when an unexpected condition was encountered and no more specific message is suitable.
* **503 Service Unavailable:** The server is currently unavailable (overloaded or down).

**To build and compile python applications:(CLI) ☹ 😊**

**STEP 1:  Install python3 and pip3**

* sudo apt update -y
* sudo apt install python3 python3-pip -y( pip3 is python package manager )
* pip3 install flask (flask applications)

**Note:**If we get "error: externally-managed-environment" -> Run below command

**sudo rm /usr/lib/python3.12/EXTERNALLY-MANAGED**

**STEP 2:  Create flask app in ec2 (vi app.py)**

from flask import Flask

import socket, Json

app = Flask(\_\_name\_\_)

@app.route('/')

def print\_ip():

hostname = socket.gethostname()

get\_ip = socket.gethostbyname(hostname)

return get\_ip

@app.route('/health')

def health\_check():

return json.dumps({'success':True}), 200, {'ContentType':'application/json'}

@app.route('/name')

def print\_name():

return 'Harsha Jain'

@app.route('/error')

def print\_error():

print('500 error')

sum = 0/100

return sum

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host="0.0.0.0", port="8080")

**STEP 3:  Run the flask app.py**

python3 app.py

**STEP 4:  Access the flask app**

To access the flask app

* + - * Open 8080 port in ec2 security group
      * <ec2\_public\_ip\_address>:<port>/<path>

ex: 172.31.10.232:8080 - Request root of application which prints ip of ec2

      172.31.10.232:8080/name - Request /name path which prints 'Harsha Jain'

**Web server:**

* Web servers are software applications or services that handle incoming network requests over the HTTP protocol from clients (typically web browsers) and deliver web pages, files, or other content to those clients. They play a crucial role in serving websites and web applications on the internet.
* It will understand the HTTP request and reply to that request. The file permission should not be 777 it should be 655.Dynamic content it will give.

**Application server:**

* An application server is a software framework that provides a runtime environment in which applications can run, including the necessary infrastructure and services to support the execution of those applications.
* Web servers pulse authentication or authorization and security with more capability is called application servers.

**NOTE:**

* To access the different applications in a single machine we will use one IP and different ports

**Type of request to send to web**

* To send request to server we use HTTP protocol Port: 80
* To HTTP protocol with some security features attaches is HTTPS Port :443
* To download and transfer file we will use FTP protocol
* NSLOOKUP Command: It will give the IP address of web site, its like resolving DNS name tor IP (DNS is human understandable name to IP address)
* To run our web application, we need application servers
* Flask and Django have inbuilt application server (web python framework)
* Java does not have inbuilt application server that’s why we go for 3rd party servers like “APACHE TOMCAT”, “JBOSS”

FRAMEWORK: Essential structure for build

ROUTING: It will redirect to server in flask, these are designated paths in web applications that corresponding to specific functions. These functions are responsible for processing request and returning the appropriate response (it is done by application server)

**Java build process using maven, tomcat (CLI) 😊(java applications)**

**STEP 1: Update the ec2 and Install Java, Maven, Tomcat10**

# Update new Ubuntu ec2 machine

sudo apt udpate

# Install java development kit

sudo apt install -y openjdk-21-jdk

# Install maven

sudo apt install -y maven

# Install Apache Tomcat10

sudo apt install tomcat10 tomcat10-admin -y

#To check Tomcat running

sudo systemctl status tomcat10

**STEP 2: Add user to tomcat10 to access Manager GUI Pannel**

sudo vi /etc/tomcat10/tomcat-users.xml

# Add below line inside <tomcat-users> ..... </tomcat-users>

<role rolename="manager-gui"/>

<role rolename="admin-gui"/>

<user username="admin" password="password" roles="manager-gui,admin-gui"/>

**STEP 3: Create your own GitHub repo with the java codebase which is in Applications/java\_app.zip**

**STEP 4: Build the calculator.war which you will get inside target/calculator.war after running below maven command**

mvn clean package

**Note:** Run the above maven command in same directory where pom.xml is present

**STEP 5: Deploy the** **calculator.war manually for now through Manager GUI**

* Access the Tomcat Manage GUI: <ec2\_public\_ip\_address>:8080/manager
* User the username and password which we configure in STEP 2
* Upload the war file



**STEP 6: Access the calculator app <ec2\_public\_ip\_address>:8080/calculator**

* Default location of Jenkins is **/var/lib/Jenkins**
* **/var/lib/Jenkins/jobs:**
* **/var/lib/Jenkins/workspace:**
* Whatever the job configuration it will be stored in jobs
* Whatever job generates it will store in workspace
* Mvn command should run in a folder where pom.xml file presents

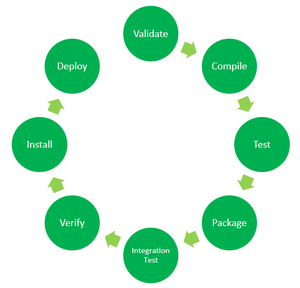
**Maven:**

* **M**aven, a standard way to build the project, a clear definition of what project consisted of, an easy way to publish project information, and a way to share JAEs across several projects
* Maven has been evolved from something called ANT
* Maven is nothing but a repository where we can access project dependencies

**Features of maven**

* Making build easy process
* Uniform build system
* Quality product/project information
* Organized development process

**Maven lifecycle**



* **validate:** validate the project is correct and all necessary information is available
* **compile:** compile the source code of the project
* **test:** test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
* **package**: take the compiled code and package it in its distributable format, such as a JAR.
* **integration-test**: process and deploy the package, if necessary, into an environment where integration tests can be run
* **verify:** run any checks to verify the package is valid and meets quality criteria
* **install**: install the package into the local repository, for use as a dependency in other projects locally
* **deploy:** done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

**PIPELINE:**

* A Jenkins pipeline allows you to define the entire build process as code, allowing for more complex, flexible, and reusable automation.
* The definition of a Jenkins Pipeline is written into a text file (called a [Jenkinsfile](https://www.jenkins.io/doc/book/pipeline/jenkinsfile)) which in turn can be committed to a project’s source control repository.

**Jenkins file:**

A Jenkinsfile is a text file that defines the configuration of a Jenkins Pipeline. It represents the entire pipeline workflow as code, allowing you to define stages, steps, and other aspects of your CI/CD process in a structured format. Here are key points about the Jenkinsfile:

* **Definition:** A Jenkinsfile is typically written in Groovy, which is a scripting language that runs on the Java Virtual Machine (JVM). It can be either a Declarative Pipeline or a Scripted Pipeline, each offering different syntaxes and levels of flexibility.
* **Location:** The Jenkinsfile is stored in the root directory of your project's source code repository. This makes it easy to version-control along with your application code, configuration files, and other artifacts.
* **Structure:**
  + **Declarative Pipeline:** Uses a more structured syntax with predefined pipeline sections (pipeline, stages, steps, etc.). This syntax is designed to be easier to read and maintain, especially for simpler workflows.
  + **Scripted Pipeline:** Offers more flexibility with a scripting environment, allowing you to define pipeline steps using Groovy scripting constructs. This syntax is suited for more complex workflows that require conditional logic, loops, or extensive programmatic control.
* **Pipeline Stages:** Both types of Jenkinsfile allow you to define stages, which represent distinct phases in your pipeline (e.g., build, test, deploy). Each stage contains one or more steps that Jenkins executes sequentially.
* **Version Control and Collaboration:** By storing the pipeline configuration as code in a Jenkinsfile, teams can manage changes through version control systems (e.g., Git). This facilitates collaboration, code review, and ensures consistency across environments.
* **Integration with Jenkins:** Jenkins automatically detects the presence of a Jenkinsfile in your project's repository. When configured to do so, Jenkins reads the file to create and execute the pipeline defined within it.
* **Benefits:** Using a Jenkinsfile promotes best practices in CI/CD, such as repeatability, transparency, and automation. It allows teams to define, version, and execute their entire software delivery process in a predictable and reproducible manner.

### Declarative versus Scripted Pipeline syntax

A Jenkinsfile can be written using two types of syntax — Declarative and Scripted.

Declarative and Scripted Pipelines are constructed fundamentally differently. Declarative Pipeline is a more recent feature of Jenkins Pipeline which:

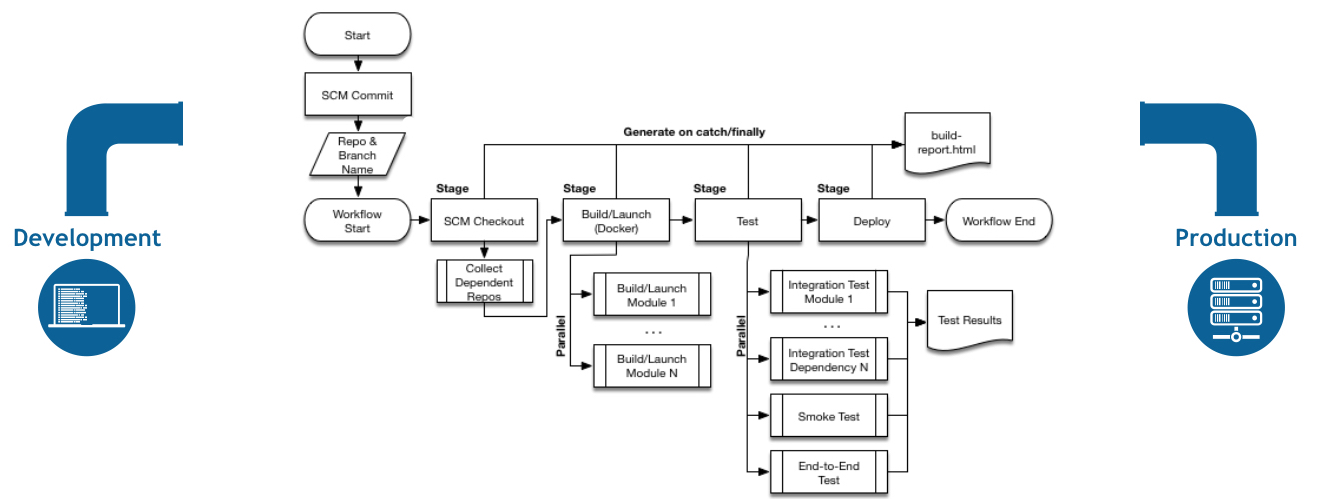
* provides richer syntactical features over Scripted Pipeline syntax, and
* is designed to make writing and reading Pipeline code easier.

Many of the individual syntactical components (or "steps") written into a Jenkinsfile, however, are common to both Declarative and Scripted Pipeline.

## **Why Pipeline?**

Jenkins is, fundamentally, an automation engine which supports a number of automation patterns. Pipeline adds a powerful set of automation tools onto Jenkins, supporting use cases that span from simple continuous integration to comprehensive CD pipelines. By modeling a series of related tasks, users can take advantage of the many features of Pipeline:

* **Code**: Pipelines are implemented in code and typically checked into source control, giving teams the ability to edit, review, and iterate upon their delivery pipeline.
* **Durable**: Pipelines can survive both planned and unplanned restarts of the Jenkins controller.
* **Pausable**: Pipelines can optionally stop and wait for human input or approval before continuing the Pipeline run.
* **Versatile**: Pipelines support complex real-world CD requirements, including the ability to fork/join, loop, and perform work in parallel.
* **Extensible**: The Pipeline plugin supports custom extensions to its DSL [[1](https://www.jenkins.io/doc/book/pipeline/#_footnotedef_1)] and multiple options for integration with other plugins.



## **Pipeline concepts**

The following concepts are key aspects of Jenkins Pipeline, which tie in closely to Pipeline syntax (see the [overview](https://www.jenkins.io/doc/book/pipeline/#pipeline-syntax-overview) below).

### Pipeline

A Pipeline is a user-defined model of a CD pipeline. A Pipeline’s code defines your entire build process, which typically includes stages for building an application, testing it and then delivering it.

Also, a pipeline block is a [key part of Declarative Pipeline syntax](https://www.jenkins.io/doc/book/pipeline/#declarative-pipeline-fundamentals).

### Node

A node is a machine which is part of the Jenkins environment and is capable of executing a Pipeline.

Also, a node block is a [key part of Scripted Pipeline syntax](https://www.jenkins.io/doc/book/pipeline/#scripted-pipeline-fundamentals).

### Stage

A stage block defines a conceptually distinct subset of tasks performed through the entire Pipeline (e.g. "Build", "Test" and "Deploy" stages), which is used by many plugins to visualize or present Jenkins Pipeline status/progress. [[6](https://www.jenkins.io/doc/book/pipeline/#_footnotedef_6)]

### Step

A single task. Fundamentally, a step tells Jenkins what to do at a particular point in time (or "step" in the process). For example, to execute the shell command make, use the sh step: sh 'make'. When a plugin extends the Pipeline DSL, [[1](https://www.jenkins.io/doc/book/pipeline/#_footnotedef_1)] that typically means the plugin has implemented a new step.

### Declarative Pipeline fundamentals

**pipeline {**

**agent any**

**stages {**

**stage('Build') {**

**steps {**

**//**

**}**

**}**

**stage('Test') {**

**steps {**

**//**

**}**

**}**

**stage('Deploy') {**

**steps {**

**//**

**}**

**}**

**}**

}

* **pipeline is** Declarative Pipeline-specific syntax that defines a "block" containing all content and instructions for executing the entire Pipeline.
* **agent** is Declarative Pipeline-specific syntax that instructs Jenkins to allocate an executor (on a node) and workspace for the entire Pipeline.
* **stage** is a syntax block that describes a stage of this Pipeline. Read more about stage blocks in Declarative Pipeline syntax on the Pipeline syntax page. As mentioned above, stage blocks are optional in Scripted Pipeline syntax.
* **steps** is Declarative Pipeline-specific syntax that describes the steps to be run in this stage.
* **sh i**s a Pipeline step (provided by the Pipeline: Nodes and Processes plugin) that executes the given shell command.
* **junit** is another Pipeline step (provided by the JUnit plugin) for aggregating test reports.
* sh is a Pipeline step (provided by the Pipeline: Nodes and Processes plugin) that executes the given shell command.

**AGENT**

* The agent section specifies where the entire Pipeline, or a specific stage, will execute in the Jenkins environment depending on where the agent section is placed. The section must be defined at the top-level inside the pipeline block, but stage-level usage is optional.

###### **Top Level Agents/Pipeline level agent**

In agents declared at the top level of a Pipeline, an agent is allocated and then the timeout option is applied. The time to allocate the agent **is not included** in the limit set by the timeout option.

pipeline **{**

agent any

options **{**

*// Timeout counter starts AFTER agent is allocated*

timeout**(time:** 1**,** **unit:** 'SECONDS'**)**

**}**

stages **{**

stage**(**'Example'**)** **{**

steps **{**

echo 'Hello World'

**}**

**}**

**}**

**}**

###### **Stage Agents**

In agents declared within a stage, the options are invoked **before** allocating the agent and **before** checking any when conditions. In this case, when using timeout, it is applied **before** the agent is allocated. The time to allocate the agent **is included** in the limit set by the timeout option.

pipeline **{**

agent none

stages **{**

stage**(**'Example'**)** **{**

agent any

options **{**

*// Timeout counter starts BEFORE agent is allocated*

timeout**(time:** 1**,** **unit:** 'SECONDS'**)**

**}**

steps **{**

echo 'Hello World'

**}**

**}**

**}**

**}**

##### **Parameters**

In order to support the wide variety of use-cases Pipeline authors may have, the agent section supports a few different types of parameters. These parameters can be applied at the top-level of the pipeline block, or within each stage directive.

**any**

Execute the Pipeline, or stage, on any available agent. For example: agent any

**none**

When applied at the top-level of the pipeline block no global agent will be allocated for the entire Pipeline run and each stage section will need to contain its own agent section. For example: agent none

**label**

Execute the Pipeline, or stage, on an agent available in the Jenkins environment with the provided label. For example: agent { label 'my-defined-label' }

Label conditions can also be used: For example: agent { label 'my-label1 && my-label2' } or agent { label 'my-label1 || my-label2' }

