**Jenkins Tutorials**

**Jenkins :**

Jenkins is an open source automation tool written in Java programming language that allows continuous integration.

Jenkins builds and tests our software projects, which continuously making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build.

It also allows us to continuously **deliver** our software by integrating with a large number of testing and deployment technologies.

Jenkins achieves CI (Continuous Integration) with the help of plugins. Plugins is used to allow the integration of various DevOps stages. If you want to integrate a particular tool, you have to install the plugins for that tool. For example: Maven 2 Project, Git, HTML Publisher, Amazon EC2, etc.

**What is Continuous Integration?**

Continuous Integration is a process of integrating code changes from multiple developers in a single project many times. The software is tested immediately after a code commit. With each code commit, code is built and tested. If the test is passed, the build is tested for deployment. If the deployment is successful, the code is pushed to production.

This commit, build, test, and deploy is a continuous process and hence the name continuous integration/deployment.

**Continuous delivery** is a software engineering method in which a team develops software products in a short cycle. It ensures that software can be easily released at any time. The main aim of continuous delivery is to build, test, and release software with good speed and frequency. It helps you to reduce the cost, time, and risk of delivering changes by allowing for frequent updates in production.

**Continuous deployment** is a software engineering process in which product functionalities are delivered using automatic deployment. It helps testers to validate whether the codebase changes are correct, and it is stable or not.

**What is the difference between CI and CD?**

[CI vs CD](https://www.guru99.com/continuous-integration-vs-delivery-vs-deployment.html): Continuous Integration (CI) is an approach of testing each change to codebase automatically, whereas Continuous Delivery (CD) is an approach to obtain changes of new features, configuration, and bug fixes.

**Continuous Delivery vs Continuous Deployment**: Key Differences. Simply put, Continuous Delivery focuses on ensuring software is always release-ready with manual approval, while Continuous Deployment automates the release process, deploying changes to production automatically once tests pass

**Jenkins Architecture**

Jenkins follows Master-Slave architecture to manage distributed builds. In this architecture, slave and master communicate through TCP/IP protocol.

Jenkins architecture has two components:

Jenkins Master/Server

Jenkins Slave/Node/Build Server

**Jenkins Master**

The main server of Jenkins is the Jenkins Master. It is a web dashboard which is nothing but powered from a war file. By default it runs on 8080 port. With the help of Dashboard, we can configure the jobs/projects but the build takes place in Nodes/Slave. By default one node (slave) is configured and running in Jenkins server. We can add more nodes using IP address, user name and password using the ssh, jnlp or webstart methods.

The server's job or master's job is to handle:

Scheduling build jobs.

Dispatching builds to the nodes/slaves for the actual execution.

Monitor the nodes/slaves (possibly taking them online and offline as required).

Recording and presenting the build results.

A Master/Server instance of Jenkins can also execute build jobs directly.

**Jenkins Slave**

Jenkins slave is used to execute the build jobs dispatched by the master. We can configure a project to always run on a particular slave machine, or particular type of slave machine, or simple let the Jenkins to pick the next available slave/node.

As we know Jenkins is developed using Java is platform independent thus Jenkins Master/Servers and Slave/nodes can be configured in any servers including Linux, Windows, and Mac.

**Jenkins Pipeline:**

Simple Words: Series of collection of jobs or events which are linked with each other in sequence which will help to build CI & CD

Technical Words: Jenkins pipeline is a suite of plugins that supports implementing and integrating continuous delivery pipeline into Jenkins

Pipeline provides an extensible set of tools for building simple-to-complex delivery pipelined “as code” with the help of the pipeline

**WebHook Trigger:**

A webhook is a way for an application to send real-time information to another application. In the context of Jenkins and GitHub, a webhook is a mechanism by which GitHub notifies Jenkins of changes to a repository. Jenkins can then automatically build and test the code in response to the webhook notification.

**Jenkins DashBoard**

**1) New Item**

In the Jenkins dashboard, the "New Item" option allows you to create a new item or project within Jenkins. When you click on "New Item," you'll be prompted to provide a name for the item and select the item type, which determines the kind of project or job you want to create.

Here are some common item types you can create using the "New Item" option:

**1) Freestyle Project:** This is the most basic item type in Jenkins. It allows you to configure and customize each aspect of your build process, such as source code management, build steps, post-build actions, and build triggers.

**2) Pipeline:** Jenkins Pipeline is a powerful way to define your build process using a Groovy-based DSL. With the Pipeline item type, you can define your build pipeline as a series of stages, each containing specific steps or tasks to be executed.

**3) Multibranch Pipeline:** This item type is suitable for projects with multiple branches, such as Git branches. It allows you to define a Jenkins pipeline that automatically discovers and builds branches as separate jobs, enabling continuous integration across all branches.

**In the Jenkins dashboard, the "Configure" option refers to the settings and configuration of a specific job or project. FREE STYLE PROJECT**

**1) General Settings:**

This section allows you to specify the job's name, description, and other general settings like whether it should be parameterized or concurrent build options.

**I)Discard Old Build:**

It determines when and how Jenkins should automatically remove older build records to manage disk space and keep the build history clean.

I) Max # of Builds to Keep: This parameter specifies the maximum number of builds to retain. Jenkins will automatically delete older builds beyond this limit, keeping the most recent ones.

II) Days to Keep: Instead of specifying a maximum build count, you can define the number of days to retain builds. Jenkins will delete builds older than the specified number of days.

**II) This Project Is Parameterized:**

The "This project is parameterized" option in Jenkins allows you to define and configure parameters for a specific job. When you enable this option, you can specify input parameters that users can provide when triggering the build, allowing for dynamic and customizable builds.

By selecting "This project is parameterized," you can add various types of parameters to your Jenkins job. Some commonly used parameter types include:

String Parameter: Allows users to enter a text value as a parameter. This can be used to pass configuration settings, file paths, or any other textual input.

Boolean Parameter: Provides a checkbox for users to select a boolean (true/false) value as a parameter. This can be useful for enabling or disabling specific build options or features.

Choice Parameter: Allows users to select a value from a predefined list of options. This can be helpful when you want users to choose from a specific set of values, such as selecting a target environment or a version number.

File Parameter: Enables users to upload a file as a parameter. This can be useful when users need to provide additional resources or configurations specific to the build.

Password Parameter: Allows users to enter sensitive information, such as passwords or API keys, securely. The entered value is masked and not displayed in the build logs.

**III) The "Throttle Builds" option**

**IV) The "Execute concurrent builds if necessary" option**

**2) Source Code Management:**

**I) GIT:**

The git plugin provides fundamentals git operation for Jenkins project it can pull, fetch, checkout, and merge content in git repositories

**3) Build Triggers:**

This section lets you define when the job should be triggered, such as periodically at specific intervals, based on changes in the repository, or by manually triggering it.

**I) Trigger Builds Remotely:**

The "Trigger builds remotely (e.g., from scripts)" option in Jenkins allows you to initiate a build for a specific job using an HTTP POST request. It provides a mechanism to trigger builds remotely, typically from external scripts, tools, or other systems.

**II) Build After Other Projects Are Built:**

It enables you to specify that a particular job should be triggered automatically after the successful completion of one or more other specified jobs.

**III) Build Periodically:**

option in Jenkins allows you to schedule a job to run at specific time intervals or at predefined time intervals using cron-like syntax. It enables you to automate the execution of a job on a regular basis without manual intervention.

Min(0-59) : Hour(0-23):Day of month(1-31 days) : Month (1-12) : Day of week (0-7)

\* \* \* \* \*

**V) The Poll SCM:**

option in Jenkins allows you to periodically check the source code repository for changes and trigger a build if any new changes are detected. It enables continuous integration by automatically monitoring the version control system (e.g., Git, Subversion) for updates and initiating builds based on those changes.

**4) Build Environment:**

Here, you can configure the environment variables, tool installations, and other settings required for the build environment.

**I) Delete workspace before build starts**

**II) Use secret text or files**

**III) add timestamps to the console output:**

If the build has been running or stopped it will give all timmings

in Jenkins allows you to include timestamps for each line of output in the build console log. When this option is enabled, Jenkins will prepend a timestamp to every line printed in the console output of a build.

**IV) inspect build log for published build scans**

The "Inspect build log for published build scans" option in Jenkins allows you to automatically scan the build log for references to published build scans. It helps in identifying build scans produced by tools like Gradle or Maven and provides a convenient way to access and analyze the detailed build scan information.

**V) terminate a build if it stuck**

**VI) with ant?**

**5) Build Steps:**

This section allows you to define the actions that Jenkins should perform as part of the build process. You can specify shell commands, build scripts, or even trigger other jobs as build steps.

**6) Post-build Actions:**

These are the actions that Jenkins performs after the build completes, such as archiving artifacts, sending notifications, generating reports, or deploying the built application.

**Jenkins Pipelines:**

In Jenkins, a pipeline is a collection of events or jobs which are interlinked with one another in a sequence.

It is a combination of plugins that support the integration and implementation of **continuous delivery pipelines** using Jenkins.

**What is Continuous Delivery Pipeline?**

In a Jenkins Pipeline, every job has some sort of dependency on at least one or more jobs or events.

 It contains a collection of states such as build, deploy, test and release. These jobs or events are interlinked with each other. Every state has its jobs, which work in a sequence called a continuous delivery pipeline.

**JenkinsFile**

Jenkins Pipeline can be defined by a text file called JenkinsFile. You can implement pipeline as code using JenkinsFile, and this can be defined by using a DSL (Domain Specific Language). With the help of JenkinsFile, you can write the steps required for running a Jenkins Pipeline.

The benefits of using JenkinsFile are:

* You can make pipelines automatically for all branches and can execute pull requests with just one JenkinsFile.
* You can review your code on the pipeline.
* You can review your Jenkins pipeline.
* This is the singular source for your pipeline and can be customized by multiple users.

JenkinsFile can be defined by using either Web UI or with a JenkinsFile.

**Pipeline syntax**

Two types of syntax are used for defining your JenkinsFile.

**Declarative**

Declarative pipeline syntax offers a simple way to create pipelines. It consists of a predefined hierarchy to create Jenkins pipelines. It provides you the ability to control all aspects of a pipeline execution in a simple, straightforward manner.

**Scripted:**

Scripted Jenkins pipeline syntax runs on the Jenkins master with the help of a lightweight executor. It uses very few resources to convert the pipeline into atomic commands.

Both scripted and declarative syntax are different from each other and are defined totally differently.

**Fields available in declarative pipelines:**

1) agent: Specifies where the pipeline will run, such as on a specific agent or a group of agents.

2) Stages - Inside the stages you have to write the stage. Defines the stages of the pipeline. Each stage represents a logical division of work in the pipeline.

3) Stage - Inside the stage you have to write the steps

4) Steps - what should we have to run we will specify in this steps . Specifies the steps to be executed within a stage. Steps can include various actions like building, testing, deploying, etc.

**what we will specify in agent decalartion**

In a Jenkins declarative pipeline, the **agent** declaration is used to specify where the pipeline will run, either on the Jenkins master node or on a specific slave node (also known as a worker node or build agent). Here are the common ways to use the **agent** declaration:

**1. Run on Jenkins master:**

**agent { label 'master' }**: Specifies that the pipeline should run on the Jenkins master node itself.

**2. Run on a specific slave node:**

**agent { label 'my-slave' }**: Specifies that the pipeline should run on a specific slave node labeled as "my-slave". The label is typically assigned to the slave node during configuration.

**agent { node { label 'my-slave' } }**: Similar to the previous example, this syntax explicitly specifies that the pipeline should run on the node labeled as "my-slave".

**3. Run on any available slave node:**

**agent { any }**: Allows the pipeline to run on any available slave node. Jenkins will automatically select an available slave node to execute the pipeline.

**4. Use a custom workspace:**

**agent { label 'my-slave', customWorkspace '/path/to/workspace' }**: Specifies a custom workspace path on the selected slave node.

These are some examples of how the **agent** declaration can be used. You can choose the appropriate syntax based on your Jenkins setup and requirements. It's important to note that the **agent** declaration is not mandatory in a declarative pipeline. If it is not specified, the pipeline will run on the Jenkins master node by default.

**what we will specify in stages decalaration**

In a Jenkins declarative pipeline, the **stages** declaration is used to define the logical divisions of work or stages within the pipeline. Each stage represents a specific phase of the build or deployment process. Within the **stages** declaration, you can specify the following elements:

**1. Stage name:**

The name of the stage is specified as a string, enclosed in double quotes.

Example: **stage('Build')**

**2. Stage content:**

Inside each stage, you define the steps or actions to be executed.

Steps can include building, testing, deploying, or any other task required for your pipeline.

You can specify multiple steps within a stage.

Example:

stage('Build') {

steps {

// Build steps go here

}

}

**3. Parallel stages:**

You can use the **parallel** directive to execute multiple stages in parallel.

Each parallel stage can have its own set of steps.

Example:

stage('Parallel Build and Test') {

parallel {

stage('Build') {

steps { // Build steps go here } }

stage('Test') { steps { // Test steps go here } } } }

**Stage-specific environment:**

You can use the **environment** directive to define environment variables specific to a particular stage.

Example:

stage('Build') {

environment { MY\_VARIABLE = 'some value' }

steps { // Build steps go here } }

These are some examples of what you can specify within the **stages** declaration in a Jenkins declarative pipeline. You can define multiple stages, specify steps, and use directives like **parallel** and **environment** to structure your pipeline according to your specific requirements.

**what we will specify in steps declaration**

In a Jenkins declarative pipeline, the **steps** declaration is used to specify the actions or steps to be executed within a stage. Steps define the actual tasks that are performed as part of the build or deployment process. Inside the **steps** declaration, you can specify various types of steps based on your requirements. Here are some commonly used steps:

**1. Shell/Batch Script:**

**sh** (for Unix-like systems) or **bat** (for Windows) steps are used to execute shell commands or batch scripts.

Example:

steps { sh 'echo "Hello, World!"' }

**2. Command Execution:**

The **echo** step is used to print messages to the console log.

Example:

steps { echo 'Running the build...' }

**3. Build/Compilation Steps:**

Steps to compile source code, run build tools, or perform any build-related actions.

Example:

**steps { sh 'mvn clean compile' }**

**4. Testing Steps:**

Steps to run unit tests, integration tests, or any other testing frameworks.

Example:

steps { sh 'mvn test' }

**5. Artifact Archiving:**

The **archiveArtifacts** step is used to archive build artifacts for later use.

Example:

steps { archiveArtifacts artifacts: 'target/\*.jar', fingerprint: true }

**6. Deployment Steps:**

Steps to deploy the application or artifacts to a target environment.

Example:

steps { sh 'kubectl apply -f deployment.yaml' }

**7. Conditional Steps:**

The **script** step allows you to write custom Groovy scripts for conditional logic or complex actions.

Example:

steps { script { if (env.BRANCH\_NAME == 'master') { sh 'mvn deploy' } else { sh 'mvn package' } } }

These are just a few examples of the steps that can be specified within the **steps** declaration in a Jenkins declarative pipeline. Depending on your specific build, test, and deployment requirements, you can include various other steps or plugins to customize your pipeline accordingly.

**Jenkins Shared Library:**

A Jenkins shared library, often referred to simply as a "Jenkins library," is a collection of reusable Groovy code and resources that can be used across multiple Jenkins pipelines or jobs. Jenkins is an open-source automation server commonly used for building, deploying, and automating software projects. Shared libraries in Jenkins provide a way to encapsulate common functionality, promote code reuse, and maintain consistency in your Jenkins pipelines.

Here are some key points about Jenkins shared libraries:

**Reusable Code**: Shared libraries allow you to write Groovy code that can be shared and reused across multiple Jenkins pipelines. This can include functions, classes, and even entire pipelines.

**Organizational Standards**: Shared libraries are a way to enforce coding standards and best practices across all of your Jenkins pipelines. This ensures consistency in how Jenkins jobs are defined and executed.

**Version Control**: Shared libraries are typically stored in a version control system (e.g., Git) and can be versioned just like any other codebase. This allows you to track changes, collaborate with others, and roll back to previous versions if needed.

**Custom DSL**: You can define custom DSLs (Domain-Specific Languages) using shared libraries. This makes it easier to create pipelines by abstracting complex logic into simpler, more readable code.

**Security**: Shared libraries can be restricted to be used only by authorized users or teams, enhancing security and access control.

**Testing**: Like any other codebase, shared libraries can be tested to ensure their correctness and reliability. Jenkins provides mechanisms for testing shared library code.

To use a shared library in a Jenkins pipeline, you typically need to:

Define the library in your Jenkins configuration, specifying the Git repository or other version control location where it is stored.

Import and use functions or classes from the library within your pipeline scripts.

Here is a simplified example of how you might use a shared library in a Jenkins pipeline:

groovyCopy code

@Library('my-shared-library') // Import the shared library

import com.example.utils.MySharedLibraryUtils // Import a class or function from the library

pipeline {

agent any

stages {

stage('Build') {

steps {

script {

// Use a function from the shared library

def result = MySharedLibraryUtils.doSomething()

echo "Result: ${result}"

}

}

}

}

}

In this example, **my-shared-library** is the name of the shared library configured in Jenkins, and **MySharedLibraryUtils** is a class or function defined in the library.

Shared libraries in Jenkins are a powerful way to promote code reuse, maintainability, and consistency in your automation workflows. They are especially useful when you have complex or repetitive tasks that need to be performed across multiple pipelines or projects.

**GROOVY SCRIPT?**

A Groovy script is a script written in the Groovy programming language. Groovy is a dynamic, object-oriented scripting language that is often used for scripting tasks, automation, and as an alternative to Java for various purposes. It runs on the Java Virtual Machine (JVM) and is compatible with Java, which means you can seamlessly use Java libraries and integrate Groovy code with Java code

**FORTIFY SECURITY TOOL**

fortify is a software security tool suite developed by Micro Focus (formerly part of Hewlett Packard Enterprise) that is designed to help organizations identify and remediate security vulnerabilities in their software applications. Fortify provides a range of tools and services aimed at improving the security of software during the development and testing phases. Some of the key components and features of Fortify include:

**Static Application Security Testing (SAST)**: Fortify SAST analyzes the source code, byte code, or binary code of an application to identify security vulnerabilities, coding errors, and potential threats. It can detect issues such as SQL injection, cross-site scripting (XSS), and buffer overflows by examining the code and its execution paths.

**Dynamic Application Security Testing (DAST)**: Fortify DAST focuses on the runtime behavior of applications. It identifies vulnerabilities by sending HTTP requests to the application and analyzing the responses. DAST is often used to find security issues like authentication flaws and session management problems.

**Interactive Application Security Testing (IAST)**: Fortify IAST combines elements of both SAST and DAST. It instruments the application code to provide real-time feedback during testing, identifying vulnerabilities as they occur during dynamic testing.

**Software Composition Analysis (SCA)**: Fortify SCA checks for vulnerabilities and licensing issues in third-party libraries and components used in an application. It helps organizations manage the risks associated with open-source and commercial software dependencies.

**Fortify Secure Coding Rules**: Fortify provides a set of secure coding rules and best practices that developers can follow to prevent common security vulnerabilities. These rules are integrated into the development process to help developers write secure code from the outset.

**Integration with Development Tools**: Fortify can be integrated into popular integrated development environments (IDEs), build systems, and continuous integration/continuous delivery (CI/CD) pipelines. This allows security testing to be seamlessly incorporated into the software development workflow.

**Reporting and Remediation**: Fortify generates detailed reports highlighting security vulnerabilities, their severity, and recommended fixes. Developers and security teams can collaborate to remediate issues efficiently.

**Custom Rules and Policies**: Organizations can create custom security rules and policies to align with their specific security requirements and compliance standards.

**Scalability**: Fortify is suitable for both small development teams and large enterprises, and it can scale to accommodate the needs of complex software projects.

**Integration with Other Security Tools**: Fortify can be integrated with other security tools and services, such as web application firewalls (WAFs) and vulnerability management systems, to create a comprehensive security ecosystem.

**First stage in Pipeline is Git Checkout**

It will create a local copy of our source code in Jenkins workspace

**Second Stage in Pipeline is Compile**

It will compile the source code to byte code

**Third stage in Pipeline is Unit Test**

It will test all the test cases in source code

**Fourth stage in Pipeline is SonarQube**

**SonarQube Analysis:**

Imagine you're building a house. Before moving in, you want to make sure it's safe and well-constructed. SonarQube is like a tool that inspects the code you've written for your software project to make sure it's safe and well-structured, just like checking a house's structural integrity.

Here's how it works:

SonarQube (Community Edition) is an open source static + dynamic code analysis platform developed by SonarSource for continuous inspection of code quality to perform fully automated code reviews / analysis to detect code smells, bugs, performance enhancements and security vulnerabilities for 20+ programming languages.

**Code Inspection:** SonarQube looks at your code, piece by piece, and checks it for issues. These issues could be bugs, security vulnerabilities, or code quality problems.

**Quality Metrics:** It also calculates metrics to measure the quality of your code, such as complexity, maintainability, and code duplication.

**Reports:** After analyzing your code, SonarQube provides you with a detailed report. It tells you what issues it found and provides suggestions on how to fix them. It also gives your project an overall grade, like a report card for your code.

**Continuous Improvement:** Developers can use this feedback to improve the code quality and security of their software. It's like having a home inspector come by regularly to make sure your house stays in good shape.

**Security vulnerability:** SonarQube can detect security issues that a code may face.  
**Example:** If a developer forgets to close an open SQL database or if important details like username and password have been directly written in the code.  
If the website or the application is hacked, then the hacking another person can figure out these details, access more company applications, and cause a lot of damage; to solve this, SonarQube can identify these errors.

**Fifth Stage in Pipeline is OWASP Dependency Check**

OWASP Dependency Check is like a tool that checks these Lego bricks to make sure none of them are damaged or have hidden dangers. It's specifically focused on checking for security issues in the external pieces you're using.

Or

OWASP Dependency Check is like a security scanner for computer programs. Imagine you're building a house, and you need to make sure all the materials you use are safe and up to code. In the world of software development, these "materials" are the third-party libraries and components that developers use to build their applications.

Here's how it works:

OWASP Dependency Check is a software composition analysis (SCA) tool that identifies project dependencies with known vulnerabilities. It helps developers and security professionals identify and mitigate potential risks associated with using vulnerable libraries and components.

**Dependency Scanning:** OWASP Dependency Check scans your project's dependencies (the external pieces you've used, like libraries) to see if any of them have known security vulnerabilities.

**Database of Vulnerabilities:** It has a database of known vulnerabilities, like a list of dangerous Lego bricks. It compares your dependencies against this list.

**Reports:** If it finds any known security issues in your dependencies, it provides you with a report. It tells you which pieces (dependencies) have problems and what you should do to fix or replace them.

**Security Assurance:** By using OWASP Dependency Check, you ensure that your project isn't built on unstable or risky foundations. It helps you keep your software safe by checking the safety of the building blocks you use.

**Collect Dependencies**: When you're building a software application, you often use pre-built pieces of code created by other developers. These are called "dependencies." OWASP Dependency Check helps you identify and list all the dependencies your project is using.

**Check for Vulnerabilities**: Just like you want to make sure the materials in your house are safe, you want to ensure your software dependencies are secure. OWASP Dependency Check scans your list of dependencies to see if any of them have known security vulnerabilities. These vulnerabilities are like weaknesses or flaws in the materials that could be exploited by malicious hackers.

**Alert on Issues**: If OWASP Dependency Check finds any vulnerabilities in your dependencies, it raises an alert. This alert tells you what the problem is, which dependency is affected, and often provides information on how to fix it. It's like a warning sign that says, "Hey, this part of your software might not be safe!"

**Take Action**: After receiving alerts, you, as the developer or project manager, can take action. This might involve updating the vulnerable dependency to a newer, secure version, finding an alternative dependency, or implementing other security measures to protect your application.

In essence, OWASP Dependency Check is a tool that helps developers ensure they're not using "unsafe" building blocks in their software, making their applications more secure from potential threats. It's like having a safety inspector for your software's construction materials.

**Fifth Stage in Pipeline is NEXUS**

Nexus Repository Manager: Nexus Repository Manager is a popular artifact repository management tool often used in DevOps environments. It serves as a central hub for storing and managing software artifacts such as binary files, libraries, and dependencies. The Nexus Repository Manager provides version control, access control, and distribution capabilities, making it easier for development and operations teams to collaborate on building and deploying software. By having a centralized repository, teams can ensure consistency, traceability, and efficient sharing of artifacts across the software development lifecycle.

**Fifth Stage in Pipeline is JFROG**

What is JFrog used for?

JFrog Artifactory is a repository manager that supports all available software package types, enabling automated continous integration and delivery. Add Artifactory to your toolchain and store build artifacts in your Artifactory repository.

JFrog Artifactory is a versatile repository manager used primarily in the software development and DevOps processes. Its main purpose is to efficiently store, manage, and organize various types of software artifacts and dependencies used in the software development lifecycle. Here's what JFrog Artifactory is used for:

**Artifact Management**: Artifactory serves as a central repository for storing binary artifacts, including libraries, dependencies, and build artifacts. It supports a wide range of package types, such as Docker images, Java libraries (Maven and Gradle), JavaScript packages (npm and Yarn), RubyGems, Python packages (PyPI), and more. This makes it a one-stop-shop for managing all your software assets.

**Build and CI/CD Integration**: Artifactory seamlessly integrates with continuous integration and continuous deployment (CI/CD) pipelines. Developers can publish build artifacts directly to Artifactory during the build process, ensuring that they are stored in a reliable and versioned manner. This makes it easy to reproduce builds and deployments.

**Dependency Management**: Artifactory helps developers manage project dependencies by caching and versioning external dependencies (such as libraries and frameworks) in a local repository. This reduces external network dependencies and speeds up builds.