Unit-4 : Integrating the System

- "Integrating the system" refers to the process of connecting and coordinating various components, tools, and processes within a software development environment to work seamlessly together.
- System integration involves making sure that all these elements collaborate effectively to automate and streamline(put in order) the software development lifecycle.
- As we know, SDLC starts from the developer writing the code. And ends at the production team deploying it into the production server.
- When the code gets pushed into local repos, the testing team has to validate the code by building the code, and performing various test cases on it.
- GIven that the developers can push code changes at any time, it is impossible for the testing to be available all the time and validate the code changes to push into the production server. Hence, this delays the project work-flow.
- To overcome these delays, we can use some automated tools that can build the code, test it with specified test cases and deploy it into the production server.
- Jenkins is one of the popular automated tools that automates the CI/CD pipeline, thereby overcoming the delays in project workflow.
- Hence, integrating the tools/systems like Jenkins, has become mandatory to have a productive and fast CI/CD pipeline.
- Hence, integration of the automated systems ensures the Robustness, Scalability, and Accuracy of SDLC.

Build Systems:

- “Build” refers to the process of compiling and making the code ready to be tested.
- It involves:
    - Compiling the code
    - Installing the dependencies
    - Running the code locally
- When the developers push the code changes into a local repo, the phase of testing comes in.
- The first task of the testing team is to pull the changes and run it on their local machines.
- If a code runs on a developers machine, the testers have to make sure that it runs on their local machine as well, to perform test cases.
- Building the code on a new machine is a time consuming process. Because testers have to ensure that:
    - Correct versions of dependencies are installed
    - Correct environment variables are set. (API links, etc)
- And this has to be done for every commit from the developers team.
- Hence, to make the build process faster, we use automations tools like Jenkins.
- Hence, the System which uses automation tools to Build a code, is known as a Build System.
- The build system varies based on the programming language.
- The build systems for various programming languages are:
    - For Java - Maven, Gradle, Ant
    - For JavaScript - Grunt
    - For Ruby - Rake
    - For Scale - sbt
- Initially, Jenkins used to support Build Systems for only Java. But now, it supports all of these build systems, which makes it a versatile automation tool.

Jenkins Build Server:

- A machine or a set of machines designed to perform the automated process of Building (compiling source code), Testing (running test cases), and Producing Deployable Files(executable files created from source code), is known as a Build Server.
- The primary purpose of a Build Server is to streamline(put in order) the software development process and automating it.
- Jenkins is a popular and open source Build Server written in Java, that is used for the same purpose of automating the CI/CD pipeline.
- It is based on Client-Server Architecture.
- It is highly extensible. (able to integrate external plugins to extend the functionalities).
- For Example: Git/GitHub (version control system) can be integrated with Jenkins to automate the process of pushing the latest code changes into a common repository and monitor the repository.
- Jenkins has a Web-Based Interface, but not an OS-Based Interface.
- Once it is installed, it runs on a local-host URL with port (8080), by default.
- Each task in Jenkins(such as compiling, running tests or packaging artifacts) is referred to as a “Job”.
- These Jobs can be triggered by various events, such as code commits, pull requests, scheduled intervals, or successful completion of other jobs.
- Benefits of using Jenkins as the Build Server:
    + Automation: Automating the various stages of CI/CD pipeline to fasten the Software Development process.
    + Extensibility: Ability to integrate the server with external plugins to enhance the functionalities.
    + Versatility: Jenkins supports Build Systems for Various programming languages.
    + Parallelism with Build Slaves: It can delegate the build tasks to multiple machines connected to the common Jenkins Server, known as Build Slaves.
    + Security and Authentication: Jenkins provides options for securing access to its web-interface, with RBAC (Role-Based-Access-Control)
    + Monitoring and Reporting: Jenkins provides monitoring and reporting features to track build statuses, view logs, and analyze build trends over time.
    + Community and Support: As Jenkins is a popular platform, It has a large community and support teams.

Managing Build Dependencies:

- One of the steps of Building an application is “installing dependencies”.
- Dependencies are the external libraries and frameworks that a software application uses to run and operate.
- Every dependency has its own version. (latest or old)
- Inorder to run an application locally, all the dependencies that are used in a project source code, must be installed with correct versions.
- But it is an inefficient and tedious process to look over the whole source code and install dependencies one-by-one.
- Hence, there must be a dependencies management tool which records the used dependencies and its versions, when developers install them, at the first place.
- Dependency Management Tool (DMT) varies based on the programming language used on the project.
- Some DMT’s based on programming languages, are:
    - For Java - Maven
    - For JavaScript/NodeJS - npm (Node-Package-Manager)
    - For Python - pip (Preferred-Installer-Program)
- Hence, in order to automate and fasten the Build Process, we use DMT.

Jenkins Plugins:

- A plugin is a software component which can be installed on any project and integrated with it, to support the specific functionality that the software component is offering.
- Plugins are used to enhance the functionalities of a software project.
- Jenkins supports a plugin environment, which makes it Extensible.
- Based on the project requirements, the plugins can be integrated to provide additional functionalities.
- Some popular Plugins supported by Jenkins are:

1. Source Code Management (SCM) Plugins:
    - Git Plugin: Integrates Jenkins with Git, enabling the automation of builds triggered by Git repository changes.
    - Subversion Plugin: Allows Jenkins to work with Subversion (SVN) repositories.
2. Build Tool Integration Plugins:
    - Maven Integration Plugin: Facilitates integration with Apache Maven, a popular build tool for Java projects.
    - Gradle Plugin: Enables integration with the Gradle build tool for Java, Groovy, and Kotlin projects.
3. Build Environment and Configuration:
    - Docker Plugin: Integrates Jenkins with Docker, enabling the use of Docker containers in build processes.
    - EnvInject Plugin: Allows injecting environment variables into the build environment.
4. Deployment Plugins:
    - Deploy to Container Plugin: Facilitates the deployment of artifacts to containers like Tomcat, JBoss, or other servlet containers.
    - Deploy to Kubernetes Plugin: Enables deployment to Kubernetes clusters directly from Jenkins.
5. Testing Plugins:
    - JUnit Plugin: Publishes JUnit test results in Jenkins, providing a detailed overview of test outcomes.
    - TestNG Plugin: Integrates Jenkins with TestNG, a testing framework for Java.
6. Notification and Reporting Plugins:
    - Email Extension Plugin: Allows sending customizable email notifications on build completion.
    - HTML Publisher Plugin: Publishes HTML reports, which can be useful for displaying test results or other build-related information.
7. Version Control Integration Plugins:
    - GitHub Plugin: Integrates Jenkins with GitHub, allowing automatic builds triggered by GitHub events.
    - Bitbucket Branch Source Plugin: Provides integration with Bitbucket Cloud and Bitbucket Server repositories.
8. Security and Authentication Plugins:
    - Role-based Authorization Strategy: Enhances Jenkins security by implementing role-based access control.
    - LDAP Plugin: Integrates Jenkins with LDAP for user authentication.
9. Pipeline and Workflow Plugins:
    - Pipeline Plugin: Allows the creation of continuous delivery pipelines as code using the Groovy scripting language.
    - Multibranch Scan Webhook Trigger: Triggers Jenkins pipeline builds based on repository events through webhooks.
10. Monitoring and Metrics Plugins:
    - Monitoring Plugin: Monitors and visualizes Jenkins resource usage, providing insights into system performance.

- Prometheus Plugin: Integrates Jenkins with Prometheus, enabling the collection and monitoring of metrics.

11. Utility Plugins:
    - Copy Artifact Plugin: Enables the copying of artifacts from one project to another.
    - Parameterized Trigger Plugin: Allows triggering other jobs with parameters.
12. Cloud Integration Plugins:
    - Amazon EC2 Plugin: Enables dynamic scaling of Jenkins build agents using Amazon EC2 instances.
    - Google Kubernetes Engine Plugin: Integrates Jenkins with Google Kubernetes Engine for dynamic Kubernetes agent provisioning.

Jenkins FileSystem Layout:

- Jenkins Server runs locally on a web-interface. So, the Jobs and Build related files and directories are stored in that local computer.
- The testing team/admin who manages the Jenkins Server, must be familiar with its filesystem too.
- There are three main reasons for that:
    1. Creating regular backups to avoid data loss in case of system failures.
    2. Migrating Jenkins Server to another system.
    3. Debugging the issues, when a Build fails.

1. Creating regular backups to avoid data loss in case of system failures:
    + In case of any system failures, those files cannot be recovered (unless it is backedUp) and the whole Build History and related data, will be lost.
    + Hence, it is a good practice to have a backup copy of Jenkins FileSystem.
2. Migrating Jenkins Server to another system:
    + For each Job, there is a separate directory in a directory called “Jobs”, in which there will be an XML file which contains the corresponding job’s description.
    + And there is a directory for each build, called “workspace”.
    + Hence, Jobs and Builds take up a lot of space.
    + Hence, Incase of any less space or any other related reason, the FileSystem must be migrated to another system.
    + For migrating, we can only consider the jobs with its XML files.
3. Debugging the issues, when a Build fails:
    + Incase of Build failures, the testing team must debug the issue and fix it.
    + For that purpose, the testing team must be involved with Jenkins FileSystem.

Host Server:

- In a client-server architecture, the Host plays an important role:
- The role of a Host Server is:
    - Listen, Accept and Manage the Client requests.
    - Provide resources and services.
- A single host server provides services to multiple clients, concurrently, at any time.
- It is nothing but a computer/system connected to a network.
- Hence, the Host Server must be built with powerful and scalable components and specifications, to be able to compute and provide fast services to its clients. (here client is a device/ an application from which the request has arrived)
- Similarly, Jenkins is a Build Server which provides automation services to its DevOps and Testing teams.

Build Slaves:

- Build Server (like jenkins) is a centralized server which provides automation services to all DevOps and Testing teams of an organization.
- When the Jobs of the Build server are long and more, it becomes tedious to package the source code and build artifacts.
- Hence, a build server can be connected/integrated with additional machines which can perform build processes, known as Build Slaves.
- The use of Build Slaves fastens the Build process, by which the Build Server can be able to provide faster automation services.
- Hence, Build Slaves are the additional machines connected to the Centralized Build Server, so that the Build Server can delegate some of the jobs to those Build Slaves to parallelize the Build Process and provide faster services.

Software on the Host:

- Jenkins or any other Build Server just triggers the execution of jobs.
- The actual Building process is performed by the softwares / plugins installed on top of the Build Server.
- Hence, A Build Server can have multiple types of softwares installed and running on top of it.
- For example, the Maven build tool is the one which builds Java projects.
- And the type of Operating System on which the Build Server is running, is also a type of software.
    I.e, OS based Software, such as linux, windows, macOS, etc.

Triggers:

- Triggers are events that initiate the execution of a Build Job, on a Build Server, based on the occurrence of some specified condition(s).
- They are essential for automating the build process.
- Testing team is the one who writes the code for the Build process and triggers.
- The testing team must ensure that build jobs are triggered correctly (in response to relevant events).
- Some common types of triggers in a Build Server are:

1. Code Commit or Code Change Trigger:
    - Definition: The build is triggered automatically when changes are committed to the version control system (e.g., Git, SVN).
    - Purpose: Ensures that builds are initiated whenever there are new code changes, allowing for continuous integration.
2. Scheduled Build Trigger:
    - Definition: Builds are scheduled to run at specific intervals, regardless of code changes.
    - Purpose: Useful for running regular builds to check for issues, perform automated testing, or generate periodic releases.
3. Pull Request Trigger:
    - Definition: The build is triggered when a new pull request is opened or when changes are pushed to an existing pull request.
    - Purpose: Enables automated testing and validation of code changes before merging them into the main codebase.
4. Dependency Trigger:
    - Definition: The build is triggered when specific dependencies are updated or when there are changes in external libraries.
    - Purpose: Ensures that builds are synchronized with changes in dependent components.

5. Manual Trigger:
    - Definition: Builds are triggered manually by a user, often through the Jenkins web interface or other administrative tools.
    - Purpose: Allows developers or release managers to initiate builds at specific times or after manual validations.
6. Upstream/Downstream Trigger:
    - Definition: Builds are triggered based on the success or failure of other builds (upstream or downstream builds).
    - Purpose: Enables the creation of build pipelines where one build triggers another, ensuring a sequential and coordinated build and deployment process.
7. SCM Polling Trigger:
    - Definition: The build server periodically polls the version control system to check for changes, triggering a build if changes are detected.
    - Purpose: Useful in scenarios where real-time hooks or webhooks are not available, and polling is used to detect changes.
8. Parameterized Trigger:
    - Definition: Builds can be triggered with specific parameters, allowing for customization of the build process.
    - Purpose: Enables the flexibility to run builds with different configurations or settings.

Job Chaining and Build Pipelines:

- As we know that. Jobs created on Jenkins by the testing team, can be triggered by various conditions such as code commits, scheduled intervals, or successful completion of other jobs.
- In the Build process, the subsequent jobs are mostly triggered by successful completion of previous jobs.
- The sequence of triggering and execution of the jobs are pre-specified by the testing team, before starting the automation build process.
- This process of determining the sequence of triggering and execution of jobs, is known as Job Chaining.

(or)

- Hence, the process of determining the streamline for a build process is known as Job Chaining.
- A Job can be any small/large part of the build process.
- Based on the type of Jobs, the Job Chaining can be Sequential or Parallel.

1. Sequential Job Chaining:
    - The Jobs are linked in a linear sequence.
    - The output or artifacts produced by one Job is the input to the next Job.
    - The successful completion of a Job triggers the next Job in the sequence.
    - Hence, this Chain is considered when Jobs are dependent on each other.
    - Ex:

        Job 1 : Code Compilation
        Job 2 : Running Test Cases
        Job 3 : Packaging the Application
2. Parallel Job Chaining:
    - The Jobs are linked parallelly to be executed concurrently.
    - Each task can be performed independently.
    - This Chain is faster than Sequential Chaining.
    - Hence, this Chain is considered when Jobs are independent of each other.
    - Ex: Performing Unit Testing on different modules of a large application.

- Jobs of a Build Process can be separated into various stages.
- A Standard Build Process usually have (6) stages, where each stage contains its own set of Jobs, they are:

1. Build Stage
2. Test Stage
3. Analysis Stage
4. Packaging Stage
5. Deploy to Staging
6. Deploy to Production

- These stages are sequential in nature. (Sequential Staging)(Executed one after the other)
- This Job Chain, that defines the streamline of execution of Jobs, is known as Build Pipeline.
- In Jenkins, the Pipeline of Build Process is defined in a script, known as “JenkinsFile”.

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                // Code compilation and basic checks
            }
        }
        stage('Test') {
            steps {
                // Unit testing
            }
        }
        stage('Analysis') {
            steps {
                // Static code analysis
            }
        }
        stage('Package') {
            steps {
                // Packaging the application
            }
        }
        stage('Deploy to Staging') {
            steps {
                // Deployment to staging environment
            }
        }
        stage('Deploy to Production') {
            steps {
                // Deployment to production environment
            }
        }
    }
}
```

Build Servers and Infrastructure as Code:

Building by Dependency Order:

- Dependencies are the libraries and frameworks on which the source code is dependent, to execute and run.
- Each Job in Jenkins has its own set of dependencies.
- A dependency (A) can also be dependent on dependency(B). In such a case, the dependency(B) must be installed and executed first.
- Hence, while performing a Build, the order of installation and execution of these dependencies does matter.
- Hence, the testing team has to ensure that dependencies are species in correct order.
- In Build tools like “Make”, the dependency order is stated as a code:

Dep(A) : Dep(B), Dep(C)
Dep(B) : Dep(C)
Dep(C) :

- Dep(A) is dependent on B and C. Hence, B and C must be installed first.
- Dep(B) is dependent on C. Hence, C must be installed first.
- Hence, the dependency installation order is: C -> B -> A
- But, In Jenkins, there is support for visually managing the dependencies, for the “Maven” build tool, in the web-interface.
- Hence, Jenkins makes it much simpler and easier for managing the order of installation of dependencies.

Build Phases

Alternative Build Servers

Collating Quality Measures