**SESSION - 76**

--> Create jenkins instances.

--> Connect the server.

--> install and connect the Jenkins.

**Shift Left**

**========**

app deploy in QA SIT UAT --> test

PROD deploy --> testing, no scanning

we are using shift-left strategy for our CICD pipelines, instead of testing and scanning after deploying the application in high environment we are doing testing and scanning in DEV environment, so we can find the issues in early stages.

**--> curl -o /etc/yum.repos.d/jenkins.repo \**

**<https://pkg.jenkins.io/redhat-stable/jenkins.repo>**

**--> rpm --import <https://pkg.jenkins.io/redhat-stable/jenkins.io-2023.key>**

# Add required dependencies for the jenkins package

**--> yum install fontconfig java-21-openjdk jenkins -y**

**--> systemctl daemon-reload**

**--> systemctl start jenkins**

**--> systemctl enable jenkins**

--> take IP and connect in Browser

--> Install system plugins

--> Jenkins -- sso

--> after joining company they will provide username and password. Total in integrate this only

--> Getting started

--> username

--> password

--> Email address

--> save and continue.

--> Install configuration --> save and continue.

--> Jensins is ready.

freestyle pipelines --> create the pipeline in UI with some clicks.

1. cant' restore if error comes

2. can't track, no version control

3. no body remember how to do it again

4. can't re use

5. time consuming

--> New item

--> Everything is called as job in Jenkins.

--> hello-pipeline

--> free style project

We have 3 stages in pipeline

Pre-Build Build Post-Build

Pre production production Post production

Pre-Build -> Options, environment, where to run

Build --> Actual pipeline

Post-Build -> fail, success or anything notifications

--> Description : Hello World

--> below all options are pre build.

--> Will do it in pipeline

--> When it will trigger

--> Here build

--> Command

#!/bin/bash

echo “Hello Jenkins Freestyle Pipeline”

--> save and apply

--> Console output

--> Pipeline syntax

--> Here through Jenkins pipe we should do

--> New item --> Hello-pipeline

--> give pipeline and ok

--> pipeline --> Pipeline script from scm

--> SCM (source code management) - which one is better

--> SCM means directly pasted.

--> It is better to store the pipeline git so that you can truncate version control it restore it and you can apply

--> Create a repository

--> example

Pipeline {

Options {

}

Stages {

Stage(‘Build’){

}

}

}

--> Pipeline script from SCM

--> SCM --Git

--> Repository URL : <https://github.com/Shan23-hash/jenkins-practice.git>

--> Branch -- Main

--> Script Path -- Jenkinsfile --> Save

--> Jenkins --> Mange Jenkins --> Plugins --> Available plugins --> pipeline Stageview

--> After success you can comeback

--> See hello pipeline

--> Console output --> You can see everything

**Master Agent/Node**

**=================**

Jenkins has to run different project pipelines. diff languages, diff versions, diff devices

Jenkins master responsibility to distribute the builds to different agents. We can have more agents with diff programming languages, diff OS and diff environment. So that load will be reduced on the master. Master collects the logs from agents and monitor them.

--> For agent need to create one more server.

--> Jenkins-agent -- server name (take private ip)

--> Jenkins -- settings -- manage jenkins -- set up agent -- nodes -- Configure --> Don’t mark agents temparory offline --> give all places

--> Node name : roboshop

--> Permanent Agent --> Create

--> How many pipelines you have to run at a time - ex:3

--> Remote root diectroy --> /home/ec2-user/jenkins-agent

--> Labels : AGENT -1

--> usage : Only build jobs with label expressions

--> Launch agents via SSH

--> Host : IP Address

--> Credentials : Add username and password

--> user-name - ec2-user

--> DevOps321

--> ID : ssh-auth

--> Description - ssh-auth

--> ADD

--> select stored one

--> Host verification strategy : Non verifying verification strategy

--> Availability : keep this agent online as much as possible

--> SAVE

--> See the log

--> We need to involve java and as well as

--> Agent also build with java

--> Agent regarding package also

--> Launch new instance

--> Jenkins-agent

--> Recent - 4cf

--> Processed without key pair

--> allow-all

--> launch instances

--> Take ip and connect

--> Change ip address --> take private ip --> configure in jenkins --> Host :ip --> Save

--> Here need to install java that is required.

**--> sudo yum install fontconfig java-21-openjdk jenkins -y**

--> After downloading relaunch the agent

--> Connections is not low memory

--> it’s connected but data not enough

--> agents are successfully connected in online

--> Jenkins

**Jenkins-practice/Jenkinsfile**

pipeline {

agent {

label 'AGENT-1'

}

stages {

stage('Build') {

steps {

echo 'Building..'

}

}

stage('Test') {

steps {

echo 'Testing..'

}

}

stage('Deploy') {

steps {

echo 'Deploying....'

}

}

}

}

**--> git add . ; git commit -m "conflicts resloved"; git push origin main**

--> Rebuild in jenkins

--> Console output -- > check log

--> Push and run again

--> Running in roboshop

--> Go to server

--> That path will create

--> cd jenkins-agent

--> remoting.jar--that is agent software

--> remote.jar will use and connect in this.

--> will track through this

--> cd workspace/

--> cd hello-pipeline1 --> ls -l

--> that one will execute here ad result will go to master.

--> This is master agent

--> you can configure many jenkins under jenkins project

--> different agents you can configure.

--> use labels and call

--> Post build -- failure/success/aborted

--> Pipeline failure or suceess -- always only will run

##### 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' }

**node**

agent { node { label 'labelName' } } behaves the same as agent { label 'labelName' }, but node allows for additional options (such as customWorkspace).

**Conditions**

**always**

Run the steps in the post section regardless of the completion status of the Pipeline’s or stage’s run.

**changed**

Only run the steps in post if the current Pipeline’s run has a different completion status from its previous run.

**fixed**

Only run the steps in post if the current Pipeline’s run is successful and the previous run failed or was unstable.

**regression**

Only run the steps in post if the current Pipeline’s or status is failure, unstable, or aborted and the previous run was successful.

**aborted**

Only run the steps in post if the current Pipeline’s run has an "aborted" status, usually due to the Pipeline being manually aborted. This is typically denoted by gray in the web UI.

**failure**

Only run the steps in post if the current Pipeline’s or stage’s run has a "failed" status, typically denoted by red in the web UI.

**success**

Only run the steps in post if the current Pipeline’s or stage’s run has a "success" status, typically denoted by blue or green in the web UI.

**unstable**

Only run the steps in post if the current Pipeline’s run has an "unstable" status, usually caused by test failures, code violations, etc. This is typically denoted by yellow in the web UI.

**unsuccessful**

Only run the steps in post if the current Pipeline’s or stage’s run has not a "success" status. This is typically denoted in the web UI depending on the status previously mentioned (for stages this may fire if the build itself is unstable).

**cleanup**

Run the steps in this post condition after every other post condition has been evaluated, regardless of the Pipeline or stage’s status.

**Jenkins-practice/Jenkinsfile**

pipeline {

agent {

label 'AGENT-1'

}

environment {

COURSE = 'jenkins'

}

options {

timeout(time: 30, unit: 'MINUTES')

disableConcurrentBuilds()

}

parameters {

string(name: 'PERSON', defaultValue: 'Mr Jenkins', description: 'Who should I say hello to?')

text(name: 'BIOGRAPHY', defaultValue: '', description: 'Enter some information about the person')

booleanParam(name: 'TOGGLE', defaultValue: true, description: 'Toggle this value')

choice(name: 'CHOICE', choices: ['One', 'Two', 'Three'], description: 'Pick something')

password(name: 'PASSWORD', defaultValue: 'SECRET', description: 'Enter a password')

}

// Build

stages {

stage('Build') {

steps {

script{

sh """

echo "Hello Build"

sleep 10

env

echo "Hello ${params.PERSON}"

"""

}

}

}

stage('Test') {

steps {

script{

echo 'Testing..'

}

}

}

stage('Deploy') {

input {

message "Should we continue?"

ok "Yes, we should."

submitter "alice,bob"

parameters {

string(name: 'PERSON', defaultValue: 'Mr Jenkins', description: 'Who should I say hello to?')

}

}

steps {

script{

echo "Hello, ${PERSON}, nice to meet you."

echo 'Deploying..'

}

}

}

}

post {

always {

echo 'I will always say Hello again!'

deleteDir()

}

success {

echo 'Hello Success'

}

failure {

echo 'Hello Failure'

}

}

}

**declarative vs scripted pipeline**

**==========================**

scripted pipeline is old, declrative is new pipeiline from jenkins-2.X

scripted --> groovy based pipeline, feels little bit tough but you will have more control.

declarative --> syntax is easy, entire pipeline compiles before run the pipeline.

scripted --> compiles at the time of execution.

we are using a mix of declarative and scripted pipeline.

##### Available Options

**buildDiscarder**

Persist artifacts and console output for the specific number of recent Pipeline runs. For example: options { buildDiscarder(logRotator(numToKeepStr: '1')) }

**checkoutToSubdirectory**

Perform the automatic source control checkout in a subdirectory of the workspace. For example: options { checkoutToSubdirectory('foo') }

**disableConcurrentBuilds**

Disallow concurrent executions of the Pipeline. Can be useful for preventing simultaneous accesses to shared resources, etc. For example: options { disableConcurrentBuilds() } to queue a build when there’s already an executing build of the Pipeline, or options { disableConcurrentBuilds(abortPrevious: true) } to abort the running one and start the new build.

**disableResume**

Do not allow the pipeline to resume if the controller restarts. For example: options { disableResume() }

**newContainerPerStage**

Used with docker or dockerfile top-level agent. When specified, each stage will run in a new container deployed on the same node, rather than all stages running in the same container deployment.

**overrideIndexTriggers**

Allows overriding default treatment of branch indexing triggers. If branch indexing triggers are disabled at the multibranch or organization label, options { overrideIndexTriggers(true) } will enable them for this job only. Otherwise, options { overrideIndexTriggers(false) } will disable branch indexing triggers for this job only.

**preserveStashes**

Preserve stashes from completed builds, for use with stage restarting. For example: options { preserveStashes() } to preserve the stashes from the most recent completed build, or options { preserveStashes(buildCount: 5) } to preserve the stashes from the five most recent completed builds.

**quietPeriod**

Set the quiet period, in seconds, for the Pipeline, overriding the global default. For example: options { quietPeriod(30) }

**retry**

On failure, retry the entire Pipeline the specified number of times. For example: options { retry(3) }

**skipDefaultCheckout**

Skip checking out code from source control by default in the agent directive. For example: options { skipDefaultCheckout() }

**skipStagesAfterUnstable**

Skip stages once the build status has gone to UNSTABLE. For example: options { skipStagesAfterUnstable() }

**timeout**

Set a timeout period for the Pipeline run, after which Jenkins should abort the Pipeline. For example: options { timeout(time: 1, unit: 'HOURS') }

triggers

The triggers directive defines the automated ways in which the Pipeline should be re-triggered. For Pipelines which are integrated with a source such as GitHub or BitBucket, triggers may not be necessary as webhooks-based integration will likely already be present. The triggers currently available are cron, pollSCM and upstream.

**GitHub Webhook**

**===============**

When developer is in feature branch, we he push the code to remote I need to immidiately trigger jenkins pipeline automatically..

--> Webhooks different system

--> Jenkins different system

## How to Set Up the Jenkins + GitHub Integration

### Configuring GitHub

****Step 1:**** go to your GitHub repository and click on ****‘Settings’.****

The referenced media source is missing and needs to be re-embedded.

****Step 2:**** Click on ****Webhooks**** and then click on ****‘Add webhook’.****

The referenced media source is missing and needs to be re-embedded.

****Step 3:**** In the ****‘Payload URL’**** field, paste your Jenkins environment URL. At the end of this URL add /github-webhook/. In the****‘Content type’**** select: ‘application/json’ and leave the ****‘Secret’****field empty.

The referenced media source is missing and needs to be re-embedded.

****Step 4:**** In the page ****‘Which events would you like to trigger this webhook?’**** choose ‘Let me select individual events.’ Then, check ‘Pull Requests’ and ‘Pushes’. At the end of this option, make sure that the ‘Active’ option is checked and click on ‘Add webhook’.

The referenced media source is missing and needs to be re-embedded.

We're done with the configuration on GitHub’s side! Now let's move on to Jenkins.

### Configuring Jenkins

****Step 5:**** In Jenkins, click on****‘New Item’**** to create a new project.

The referenced media source is missing and needs to be re-embedded.

****Step 6:****Give your project a name, then choose ‘Freestyle project’ and finally, click on ‘OK’.

The referenced media source is missing and needs to be re-embedded.

****Step 7:**** Click on the ****‘Source Code Management’**** tab.

The referenced media source is missing and needs to be re-embedded.

****Step 8:**** Click on Git and paste your GitHub repository URL in the****‘Repository URL’**** field.

****Step 9:**** Click on the****‘Build Triggers’****tab and then on the ‘GitHub hook trigger for GITScm polling’. Or, choose the trigger of your choice.

The referenced media source is missing and needs to be re-embedded.

That's it! Your GitHub repository is integrated with your Jenkins project. With this Jenkins GitHub integration, you can now use any file found in the GitHub repository and trigger the Jenkins job to run with every code commit.

For example, I will show you how to run a Taurus script that I uploaded to my GitHub repository from my Jenkins project. [Taurus](http://gettaurus.org/" \o "(opens in a new window)" \t "https://www.blazemeter.com/blog/_blank) is an open source load testing solution, enabling developers to run load testing scripts from sophisticated platforms like JMeter and Selenium, but with a simple YAML code.

### Triggering the GitHub Jenkins Integration With Every Code Commit

****Step 10:****Click on the ****‘Build’**** tab, then click on ‘Add build step’ and choose ‘Execute shell’.

The referenced media source is missing and needs to be re-embedded.

****Step 11:**** To run a Taurus test, simply use the ****‘bzt’**** command, followed by the name of your YML file and click on****‘Save’.****

****Step 12:****Go back to your GitHub repository, edit the Taurus script and commit the changes. We will now see how Jenkins ran the script after the commit.

The referenced media source is missing and needs to be re-embedded.

****Step 13:**** Go back to your Jenkins project and you'll see that a new job was triggered automatically from the commit we made at the previous step. Click on the little arrow next to the job and choose****‘Console Output’.****

****Step 14:****You can see that Jenkins was able to pull the Taurus script and run it!

The referenced media source is missing and needs to be re-embedded.

GitHub -> Jenkins

--> information passing from git to jenkins

--> jenkins to git these all are events

--> Github-jenkins-webhooks

--> Settings --> webhook

--> Jenkins URL ends need to add one

--> payload URL : <https://ip:8080/github-webhook/>

--> application/json

--> You have lot of events you can trigger

--> Cron top syntax is there --> You can periodically build

--> scheduke \*\*\*\*\* --> every minute

**Input:**The input directive on a stage allows you to prompt for input, using the [input step](https://www.jenkins.io/doc/pipeline/steps/pipeline-input-step/" \l "input-wait-for-interactive-input). The stge will pause after any options have been applied, and before entering the agent block for that stage or evaluating the when condition of the stage. If the input is approved, the stage will then continue. Any parameters provided as part of the input submission will be available in the environment for the rest of the stage.