Jenkins Interview questions:

1) How to do the parallel execution of Multiple stages

A) To perform parallel execution of multiple stages in Jenkins, you can use the Jenkins Pipeline feature along with the parallel directive. Here's how:

Create a Jenkinsfile: Open the Jenkins project and go to the Pipeline section. Choose "Pipeline script from SCM" and provide the repository link and Jenkinsfile path.

Configure stages in script: Add multiple stages in the Jenkinsfile using the stage directive. For example, consider two stages - "Build" and "Test":

```
groovy
Copy code
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                // Define build steps here
            }
        }
        stage('Test') {
            steps {
                // Define test steps here
            }
        }
    }
}
```

Parallel execution: Use the parallel directive to specify the stages that need to run in parallel. Each stage will be executed on a different agent or executor.

```
groovy
Copy code
pipeline {
    agent any
    stages {
        stage('Parallel Stages') {
            parallel {
                stage('Build') {
                    steps {
                        // Define build steps here
                    }
                }
                stage('Test') {
                    steps {
                        // Define test steps here
                    }
                }
            }
        }
    }
}
```

In the above example, both the "Build" and "Test" stages will be executed in parallel.

Configure other stages: You can configure additional stages before or after the parallel block using the stage directive as shown in step 2.

Save and run the pipeline: Save the Jenkinsfile and run the Jenkins job. The stages will now execute in parallel.

Note: The parallel directive distributes the workload across agents or executors, so make sure you have sufficient resources available to run each stage in parallel effectively.

2) How do you skip the particular stages

A) To skip particular stages in Jenkins, you can use the when directive in your Jenkinsfile. Here's how you can do it:

Open your Jenkinsfile for editing.

Navigate to the stage you want to skip and add the when directive to it.

Specify the condition under which the stage should be skipped. You can use any valid Groovy expression or Jenkins predefined conditions to define the condition. For example, you can use an environment variable, a parameter, or the result of a previous stage.

Here's an example Jenkinsfile demonstrating how to skip a particular stage based on a condition:

```
groovy
Copy code
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                // Your build steps
            }
        }

        stage('Deploy') {
            when {
                environment name: 'SKIP_DEPLOY', value: 'true' // Skips the stage if SKIP_DEPLOY
variable is set to true
            }

            steps {
                // Your deploy steps
            }
        }

        stage('Test') {
            steps {
                // Your test steps
            }
        }
    }
}
```

In this example, the Deploy stage will be skipped if the SKIP_DEPLOY environment variable is set to true. You can customize the condition according to your requirements.

Make sure to commit and push the modified Jenkinsfile to your source code repository for Jenkins to pick up the changes.

3) How to skip the failed stages
A) https://www.baeldung.com/ops/jenkins-pipeline-skip-stage#:~:text=Using%20Function%20to%20Control%20Pipeline,the%20same%20code%20multiple%20times.
https://stackoverflow.com/questions/44022775/jenkins-ignore-failure-in-pipeline-build-step

To skip the failed stages in Jenkins, you can use the feature called "Conditional Stage Skip":

Open your Jenkins Pipeline in which you want to skip the failed stages.

Locate the stage that you want to skip if any previous stages have failed.

Wrap the stage within a "when" block with a condition to check the previous stage's result. For example:

```
scss
Copy code
stage('My Stage') {
  when {
    not {
      allOf {
        previousStages().isFailed()
      }
    }
  }
  steps {
    // Your stage steps here
  }
}
```

This "when" block checks if any previous stages have failed using the previousStages().isFailed() condition. If the condition evaluates to true, the stage will be skipped.

Save the changes to your Jenkinsfile.
Now, when running the pipeline, if any previous stages have failed, Jenkins will skip the specified stage and move on to the next one.

Alternative:
>>>
To skip the failed stages in Jenkins, you can add a condition to your stages using the Groovy script. Here's how you can do it:
Open your Jenkins pipeline job.
Open your Jenkinsfile or create one if you don't have it.

Locate the stage that you want to skip if any previous stages have failed.

Use the catchError and error steps to catch any failures and set a variable to indicate whether the stage has failed or not. For example:

```
groovy
Copy code
stage('My Stage') {
    steps {
        script {
            def skipStage = false
            catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE') {
                // Your stage steps here
            }
            if (currentBuild.result == 'FAILURE') {
                skipStage = true
            }
            if (!skipStage) {
                // Execute the stage that should be skipped on failure
            } else {
                echo 'Skipping this stage as previous stages have failed.'
            }
        }
    }
}
```
In this example, the catchError step catches any failures, and if the current build's overall result is 'FAILURE', it sets the skipStage variable to true.

Save the changes to your Jenkinsfile.
Now, when running the pipeline, if any previous stages have failed, Jenkins will skip the specified stage, executing the final else block that logs the message "Skipping this stage as previous stages have failed."
>>>

4) How to use the cron job inside the pipeline?

A)

To use a cron job inside a Jenkins pipeline, you can follow these steps:

Open your Jenkins pipeline script or create a new Jenkinsfile.

Inside your pipeline script, define a cron trigger using the triggers block. For example, to schedule the job to run every day at 9 AM, you can use the following code:

```
groovy
Copy code
triggers {
  cron('H 9 * * *')
}

or

properties([
  pipelineTriggers([
    cron('H 9 * * *')
  ])
])
```

The H in the cron expression ensures that the job triggers at a random minute within the 9 AM hour.

Under the triggers block, define your pipeline stages or steps as usual. For example:
```
groovy
Copy code
stages {
  stage('Build') {
    steps {
      // Your build steps here
    }
  }
  stage('Test') {
    steps {
      // Your test steps here
    }
  }
  // Add more stages as needed
}
```
Save the Jenkins pipeline script or Jenkinsfile.
Now, your Jenkins job will be triggered automatically based on the defined cron expression. Note that you should configure Jenkins to periodically scan the Jenkinsfile for changes and update the pipeline accordingly.

5) How do you write both declarative and scripted pipeline in single job

A) https://dsstream.com/declarative-vs-scripted-pipeline-key-differences/

6) How do you publish the artifact in jenkins

A) We can use mvn deploy to publish the artifact in Jenkins.

7) How to retry the specific build

A)

To retry a specific build in Jenkins, you can follow these steps:

Go to the Jenkins dashboard or home page.

Locate the specific build that you want to retry in the "Build History" section. Builds are typically listed with a build number and a short description.

Click on the build number or the build description of the specific build you want to retry.

On the build details page, look for the "Actions" or "Build Actions" menu on the left side of the screen.

Expand the "Actions" menu and locate the "Retry" option. This option might also be available directly as a button on the build details page.

Click on the "Retry" option or button. This will trigger the retry process for the specific build.

Jenkins will now start a new build with the same parameters and configuration as the original build.

Monitor the new build's progress on the build details page. You can view the console output to check for any errors or issues.

Note: It's important to analyze the cause of the build failure before retrying. Retrying without fixing the underlying problem may lead to repeated failures.

8) Different paramater you have used in pipeline job

In a pipeline job, there are different parameters that can be used. Some commonly used parameters are:

String parameter: Allows the user to enter a simple text string as a parameter.
Example: parameters {string(name: 'ENVIRONMENT', defaultValue: 'dev', description: 'Enter environment (dev/staging/production)')}

Choice parameter: Allows the user to select a value from a predefined set of choices.
Example: parameters {choice(name: 'BRANCH', choices: ['master', 'develop'], description: 'Select branch')}

Boolean parameter: Allows the user to select either true or false.

Example: parameters {booleanParam(name: 'RUN_TESTS', defaultValue: true, description: 'Run tests?')}

File parameter: Allows the user to upload a file as a parameter.
Example: parameters {file(name: 'CONFIG_FILE', description: 'Upload config file')}

Password parameter: Allows the user to enter a password, which gets masked in the Jenkins logs.
Example: parameters {password(name: 'DB_PASSWORD', defaultValue: '', description: 'Enter database password')}

Multibranch parameter: Represents a branch source as a parameter.
Example: parameters {multibranch(name: 'BRANCH', defaultValue: 'master', description: 'Select branch')}

These are just a few examples of the parameters that can be used in a Jenkins pipeline job. The parameters can be customized based on the specific needs of the job.

9) Explain uses of Shared library

A) Shared libraries in Jenkins are reusable code dependencies that can be used across multiple Jenkins pipelines and jobs. Here are some common uses of shared libraries in Jenkins:

Code Reusability: Shared libraries allow you to write custom functions, steps, or entire pipelines that can be shared and reused across different projects or pipelines within an organization. This promotes code reusability and reduces code duplication.

Standardization: Shared libraries enable you to define standardized methodologies, compliance checks, and best practices for your organization's Jenkins pipelines. Developers can then utilize these shared libraries to ensure consistency and avoid reinventing the wheel.

Simplified Maintenance: By maintaining centralized shared libraries, you can make changes or updates in one place, which will be reflected across all the pipelines or jobs using that shared library. This simplifies maintenance, improves consistency, and reduces the effort required for updates.

Security and Compliance: Shared libraries help maintain security and compliance standards by providing a centralized location to manage security-related code, like authentication, authorization, or secret management. Developers can follow predefined security practices and avoid introducing vulnerabilities into their pipelines.

Custom Functionality: Shared libraries allow you to extend Jenkins' functionality by integrating additional tools, APIs, or services that are not available out-of-the-box. For example, you can write shared libraries to integrate with external systems, cloud platforms, or custom reporting tools.

Testing and Quality Assurance: Shared libraries enable you to define and enforce standardized testing and quality assurance practices across all your Jenkins pipelines. You can create shared library functions for testing code, performing static code analysis, generating reports, or executing specific quality checks.

In summary, shared libraries in Jenkins provide a way to share reusable code, standardize and enforce best practices, simplify maintenance, enhance security, and extend the capabilities of your Jenkins pipelines.

10) How to you handle the try catch block

A) catchError: Catch error and set build result to failure
If the body throws an exception, mark the build as a failure, but nonetheless continue to execute the Pipeline from the statement following the catchError step. The behavior of the step when an

exception is thrown can be configured to print a message, set a build result other than failure, change the stage result, or ignore certain kinds of exceptions that are used to interrupt the build. This step is most useful when used in Declarative Pipeline or with the options to set the stage result or ignore build interruptions. Otherwise, consider using plain try-catch(-finally) blocks. It is also useful when using certain post-build actions (notifiers) originally defined for freestyle projects which pay attention to the result of the ongoing build.

```
node {
    catchError {
        sh 'might fail'
    }
    step([$class: 'Mailer', recipients: 'admin@somewhere'])
}
```

If the shell step fails, the Pipeline build's status will be set to failed, so that the subsequent mail step will see that this build is failed. In the case of the mail sender, this means that it will send mail. (It may also send mail if this build succeeded but previous ones failed, and so on.) Even in that case, this step can be replaced by the following idiom:

```
node {
    try {
        sh 'might fail'
    } catch (err) {
        echo "Caught: ${err}"
        currentBuild.result = 'FAILURE'
    }
    step([$class: 'Mailer', recipients: 'admin@somewhere'])
}
For other cases, plain try-catch(-finally) blocks may be used:

node {
    sh './set-up.sh'
    try {
        sh 'might fail'
        echo 'Succeeded!'
    } catch (err) {
        echo "Failed: ${err}"
    } finally {
        sh './tear-down.sh'
    }
    echo 'Printed whether above succeeded or failed.'
}
// …and the pipeline as a whole succeeds
```

11) Explain CI/CD work flow

A) CI/CD stands for continuous integration and continuous deployment (or continuous delivery). It is a development practice that enables teams to automatically build, test, and deploy software changes to production. The CI/CD workflow involves several steps:

Continuous Integration (CI): Developers integrate their code changes to a shared repository frequently, typically multiple times per day. This ensures that all changes are continuously merged with the main codebase, reducing the risk of conflicts and integration issues.

Automated Build: Whenever new code changes are pushed to the repository, an automated build process is triggered. The build process compiles the code, resolves dependencies, and generates executable files or artifacts.

Automated Testing: After the build is completed, automated tests are run to validate the functionality and performance of the software. These tests can include unit tests, integration tests, regression tests, or any other relevant test cases.

Code Quality Checks: Automated tools are used to analyze the code for coding standards, code complexity, security vulnerabilities, and other quality aspects. This helps in maintaining code quality and identifying potential issues early in the development process.

Continuous Deployment (CD): Once the code changes pass all the automated tests and quality checks, they are deployed to the production environment. CD involves automating the deployment process to minimize manual interventions and ensure consistency across deployments.

Monitoring and Feedback Loop: Continuous monitoring of the deployed application helps identify issues in production, such as performance bottlenecks or errors. This feedback loop is essential for improving the software and iterating on development and deployment processes.

The CI/CD workflow is typically supported by a variety of tools and technologies, such as version control systems (e.g., Git), build automation tools (e.g., Jenkins), testing frameworks (e.g., Selenium), containerization platforms (e.g., Docker), and deployment orchestration tools (e.g., Kubernetes). The ultimate goal of the CI/CD workflow is to increase development speed, agility, and reliability, while delivering high-quality software to end-users more frequently and with minimal manual effort.

12) Can we perform CVS testing on sonar qube

A) No, SonarQube does not provide specific support for CVS testing. SonarQube is primarily used for static code analysis, providing insights into code quality, bugs, vulnerabilities, and code smells. CVS, on the other hand, is a version control system that tracks and manages changes in source code. While SonarQube can analyze the code in a repository managed by CVS, it does not directly interact with or test CVS itself.

13) What is username of the job you running

A) JENKINS.
The username of the job running in Jenkins can vary depending on the specific configuration and setup of your Jenkins environment. Typically, Jenkins uses the credentials and access rights of the user who triggers the job to run. If you have specific permissions set up for different users, the username will be the one associated with the user running the job.

14) Where do you find settings.xml and what is use of it.

A)
The settings.xml file in Jenkins is used to define various configurations and settings for the Jenkins server. It is typically located in the $HOME/.m2 directory of the user that Jenkins is running as. However, the location can be customized by setting the M2_HOME environment variable.

The settings.xml file is primarily used to configure Maven settings for Jenkins. Maven is a build automation tool widely used in Java projects. With settings.xml, you can define the following:

Repository Configurations: It allows you to define repositories where Jenkins can download dependencies for your Maven projects. These repositories can be local or remote.

Authentication Credentials: If your repositories require authentication, you can specify the credentials in settings.xml.

Custom Plugin Repositories: Jenkins plugins can have their own repositories. These can be defined in settings.xml.

Proxy Settings: If your Jenkins server is behind a firewall or proxy, you can configure proxy settings in settings.xml.

By modifying the settings.xml file, you can customize various Maven options and integrate Jenkins with your organization's Maven repository infrastructure.

15) What are the other tools than Maven that can be used to build Java project? [Gradle, Ant, Bazel, sbt, Make]

A)
There are several other build tools that can be used to build Java projects besides Maven. Some popular alternatives include:

Gradle: Gradle is a powerful build automation tool that uses Groovy or Kotlin as its build language. It offers a flexible and declarative build configuration and is highly extensible.

Ant: Ant is an older build tool that uses XML-based configuration files. It provides a simple and flexible way to build Java projects and is often used for legacy projects.

Bazel: Bazel is a build tool developed by Google that focuses on scalability and performance. It uses a build language similar to Python and is designed to handle large projects with multiple languages.

sbt: sbt (Scala Build Tool) is a build tool specifically designed for Scala projects. However, it can also be used to build Java projects and provides features like dependency management, incremental compilation, and code testing.

Make: Make is a general-purpose build tool that can be used for building Java projects. It uses makefiles to define build targets and their dependencies.

Each of these build tools has its own strengths and weaknesses, and the choice depends on the specific requirements of the project and personal preference.

16) What is command used to upload the artifactory

A) We can use Jfog and save the Artifactories.

17) Differnce between maven package and maven install.

A) Maven package and Maven install are two commands used in the Maven build process, but they serve different purposes:

Maven Package:
The "mvn package" command is used to package and compile the source code of a project into a distributable format, such as a JAR (Java ARchive) or WAR (Web Application Archive). It executes the "package" phase of the Maven build lifecycle, which includes compilation, running tests, and packaging the compiled code and dependencies. However, it does not install the built artifacts into the local repository or any remote repository.

Maven Install:
The "mvn install" command is used to compile, package, and install the project artifacts into the local Maven repository. It executes the "install" phase of the Maven build lifecycle, which includes compilation, running tests, packaging, and installing the built artifacts into the local repository. This allows other projects within the same development environment to use the artifacts as dependencies.

In summary, "mvn package" compiles and packages the project code but does not install it into the repository, whereas "mvn install" compiles, packages, and installs the project code into the local repository so that it can be used as a dependency by other projects.