**junit 'results.xml'**

To include JUnit test results from a results.xml file in a **Jenkins Pipeline**, you can use the junit step provided by Jenkins' **JUnit Plugin**. Here's a detailed example of how to do this in a **Declarative Pipeline**:

**✅ Jenkinsfile Example with Detailed JUnit Integration**

pipeline {

    agent any

    stages {

        stage('Build') {

            steps {

                echo 'Building the project...'

                // Add your build commands here

            }

        }

        stage('Test') {

            steps {

                echo 'Running tests...'

                // Run your tests here, which should generate results.xml

                // Example: sh 'mvn test' or sh './gradlew test'

            }

        }

        stage('Publish Test Results') {

            steps {

                echo 'Publishing JUnit test results...'

                junit allowEmptyResults: true, testResults: '\*\*/results.xml'

            }

        }

    }

    post {

        always {

            echo 'Pipeline completed.'

        }

        success {

            echo 'Build succeeded!'

        }

        failure {

            echo 'Build failed!'

        }

    }

}

**🔍 Explanation of Key Parts**

* **junit step**: This tells Jenkins to parse the JUnit XML file and display the results in the UI.
  + testResults: '\*\*/results.xml': This is a glob pattern to locate your XML file.
  + allowEmptyResults: true: Prevents the build from failing if no test results are found (optional).
* **post block**: Ensures messages are printed based on the pipeline result.

**📌 Requirements**

* Make sure the **JUnit Plugin** is installed in Jenkins.
* Your test framework (like JUnit, TestNG, etc.) must generate a valid results.xml file in JUnit format.

**Who Creates results.xml?**

The results.xml file is typically created by your **test runner or build tool**, such as:

* **JUnit** (via Maven or Gradle)
* **Maven Surefire Plugin**: creates target/surefire-reports/\*.xml
* **Gradle**: creates build/test-results/test/\*.xml
* **pytest** (with --junitxml=results.xml)
* **Ant** (with <junit> task)

**archiveArtifacts 'build/\*.jar'**

In a Jenkins Pipeline, the archiveArtifacts step is used to archive build artifacts so they can be accessed later (e.g., for download from the Jenkins UI or for use in downstream jobs). Here's a **detailed example** of how to use:

pipeline {

    agent any

    stages {

        stage('Build') {

            steps {

                echo 'Building the project...'

                // Simulate building a JAR file

                sh 'mkdir -p build && echo "dummy jar content" > build/my-app.jar'

            }

        }

        stage('Archive Artifacts') {

            steps {

                echo 'Archiving JAR files...'

                archiveArtifacts artifacts: 'build/\*.jar', fingerprint: true

            }

        }

    }

    post {

        success {

            echo 'Build and archive completed successfully.'

        }

        failure {

            echo 'Build failed.'

        }

    }

}

**Explanation of Key Parts:**

* **archiveArtifacts artifacts: 'build/\*.jar'**: This tells Jenkins to archive all .jar files in the build directory.
* **fingerprint: true**: This enables fingerprinting, which helps track where artifacts are used across builds and jobs.
* **sh step**: Used to simulate a build process that creates a .jar file.
* **post block**: Provides notifications or actions based on the build result.

**📁 What does "archive" mean in Jenkins?**

When you use the archiveArtifacts step in a Jenkins pipeline, Jenkins:

1. **Finds the files** you specify (e.g., build/\*.jar)
2. **Copies them** from the build workspace
3. **Stores them** with the build record
4. **Displays them** in the Jenkins UI under "Archived Artifacts"

**stash / unstash**

In Jenkins Pipelines, stash and unstash are used to **temporarily save and restore files** between different stages of the same pipeline — especially useful when stages run on different agents or nodes.

**📦 stash – Save Files for Later Use**

stash name: 'my-artifacts', includes: 'build/\*.jar'

* **name**: A unique name to identify the stash.
* **includes**: File pattern to specify what to stash.
* This saves the files from the current workspace into Jenkins' internal storage.

**📤 unstash – Restore Previously Stashed Files**

unstash 'my-artifacts'

* This restores the files saved under the name 'my-artifacts' into the current workspace.

**✅ Example: Using stash and unstash Across Stages**

pipeline {

    agent none

    stages {

        stage('Build') {

            agent { label 'linux' }

            steps {

                sh 'mkdir -p build && echo "jar content" > build/app.jar'

                stash name: 'jar-files', includes: 'build/\*.jar'

            }

        }

        stage('Test') {

            agent { label 'windows' }

            steps {

                unstash 'jar-files'

                sh 'echo "Testing app.jar..." && ls build'

            }

        }

    }

}

**🔍 When to Use stash/unstash vs archiveArtifacts**

| **Feature** | **stash / unstash** | **archiveArtifacts / copyArtifacts** |
| --- | --- | --- |
| Scope | Within the **same pipeline** | Across **different jobs** |
| Storage Duration | Temporary (only during pipeline run) | Persistent (stored with the build) |
| Use Case | Share files between stages on different nodes | Share files between jobs or for download |

**🧪 Scenario: Multi-Agent Pipeline**

Imagine you have a pipeline with two stages:

* **Stage 1: Build** — runs on a Linux agent
* **Stage 2: Test** — runs on a Windows agent

Since these stages run on **different machines**, the workspace is not shared. So, if you build a .jar file in Stage 1, it won’t automatically be available in Stage 2.

This is where stash and unstash come in.

**withSonarQubeEnv('Name')**

The withSonarQubeEnv('Name') block in a Jenkins pipeline is used to **set up the environment for SonarQube analysis**. It ensures that the necessary environment variables (like authentication tokens and server URLs) are available to the steps inside the block.

**✅ Syntax**

withSonarQubeEnv('My SonarQube Server') {

    // Commands that require SonarQube environment

    sh 'mvn sonar:sonar'

}

**🔍 Explanation**

* **'My SonarQube Server'**: This is the name of the SonarQube server configured in Jenkins under **Manage Jenkins → Configure System → SonarQube Servers**.
* **Inside the block**: You typically run a command like mvn sonar:sonar or sonar-scanner to perform static code analysis.

**🧪 Example: Jenkins Pipeline with SonarQube**

pipeline {

    agent any

    tools {

        maven 'Maven 3.8.1'

    }

    stages {

        stage('Build') {

            steps {

                sh 'mvn clean install'

            }

        }

        stage('SonarQube Analysis') {

            steps {

                withSonarQubeEnv('My SonarQube Server') {

                    sh 'mvn sonar:sonar'

                }

            }

        }

    }

}

**🧠 Why use withSonarQubeEnv?**

* It injects the correct **SonarQube server URL**, **authentication token**, and other required environment variables.
* It ensures your scanner (like Maven or CLI) can **communicate securely** with the SonarQube server.

**step([$class: '...'])**

The step([$class: '...']) syntax in Jenkins pipelines is used to invoke **lower-level, advanced build steps** that are typically part of Jenkins plugins. This is most commonly seen in **Scripted Pipelines** or when using **steps that don’t have a declarative wrapper**.

**🧠 What does it mean?**

* step is a generic way to call a Jenkins build step.
* [$class: 'SomeClassName'] tells Jenkins which plugin class to use.
* This is often used when a plugin doesn’t have a simple DSL (domain-specific language) method available.

**✅ Example: Using step([$class: 'ArtifactArchiver'])**

This is equivalent to archiveArtifacts:

step([$class: 'ArtifactArchiver', artifacts: 'build/\*.jar', fingerprint: true])

This archives .jar files and fingerprints them — just like:

archiveArtifacts artifacts: 'build/\*.jar', fingerprint: true

**🔍 When to use this?**

* When you're using a **plugin** that only exposes its functionality via a class name.
* When you're writing a **Scripted Pipeline** and need to access a plugin step that doesn’t have a shorthand.
* When you're copying pipeline code from the **Pipeline Syntax Generator** in Jenkins.

**🛠️ How to find the correct $class name?**

1. Go to your Jenkins instance.
2. Click on **"Pipeline Syntax"** (usually in the left sidebar of a job).
3. Choose the step you want (e.g., "Archive the artifacts").
4. Click **"Generate Pipeline Script"**.
5. It will show you the step([$class: ...]) syntax.