**Jenkinsfile**

**Introduction**

Creating a Jenkinsfile, which is checked into source control, provides a number of immediate benefits:

* Code review/iteration on the Pipeline
* Audit trail for the Pipeline
* Single source of truth for the Pipeline, which can be viewed and edited by multiple members of the project.

While the syntax for defining a Pipeline, either in the web UI or with a Jenkinsfile, is the same, it’s generally considered best practice to define the Pipeline in a Jenkinsfile and check that into source control.

## **Creating a Jenkinsfile**

A Jenkinsfile is a text file that contains the definition of a Jenkins Pipeline and is checked into source control. Consider the following Pipeline which implements a basic three-stage continuous delivery pipeline.

**Jenkinsfile (Scripted Pipeline)**

*node { // <1>  
 stage('Build') { // <2>  
 /\* .. snip .. \*/  
 }  
 stage('Test') {  
 /\* .. snip .. \*/  
 }  
 stage('Deploy') {  
 /\* .. snip .. \*/  
 }  
}*

|  |  |
| --- | --- |
|  | **node** allocates an executor and workspace in the Jenkins environment. |
|  | **stage** describes distinct parts of the Pipeline for better visualization of progress/status. |

Not all Pipelines will have these same three stages, but this is a good continuous delivery starting point to define them for most projects. The sections below will demonstrate the creation and execution of a simple Pipeline in a test installation of Jenkins.

**Note:** It is assumed that there is already a source control repository set up for the project and a Pipeline has been defined in Jenkins following these instructions.

|  |  |
| --- | --- |
|  |  |

Using a text editor, ideally one which supports Groovy syntax highlighting, create a new Jenkinsfile in the root directory of the project.

In the example above, node is a crucial first step as it allocates an executor and workspace for the Pipeline. In essence, without node, a Pipeline cannot do any work! From within node, the first order of business will be to checkout the source code for this project. Since the Jenkinsfile is being pulled directly from source control, Pipeline provides a quick and easy way to access the right revision of the source code

**Jenkinsfile (Scripted Pipeline)**

*node {  
 checkout scm // <1>  
 /\* .. snip .. \*/  
}*

**Note:** The checkout step will checkout code from source control; scm is a special variable which instructs the checkout step to clone the specific revision which triggered this Pipeline run.

**Build**

For many projects the beginning of "work" in the Pipeline would be the "build" stage. Typically this stage of the Pipeline will be where source code is assembled, compiled, or packaged. The Jenkinsfile is **not** a replacement for an existing build tool such as GNU/Make, Maven, Gradle, etc, but rather can be viewed as a glue layer to bind the multiple phases of a project’s development lifecycle (build, test, deploy, etc) together.

Jenkins has a number of plugins for invoking practically any build tool in general use, but this example will simply invoke make from a shell step (sh). The sh step assumes the system is Unix/Linux-based, for Windows-based systems the bat could be used instead.

**Jenkinsfile (Scripted Pipeline)**

*node {  
 /\* .. snip .. \*/  
 stage('Build') {  
 sh 'make' // <1>  
 archiveArtifacts artifacts: '\*\*/target/\*.jar', fingerprint: true // <2>  
 }  
 /\* .. snip .. \*/  
}*

**Note:** The sh step invokes the make command and will only continue if a zero exit code is returned by the command. Any non-zero exit code will fail the Pipeline

*archiveArtifacts* captures the files built matching the include pattern (\*\*/target/\*.jar) and saves them to the Jenkins master for later retrieval.

Archiving artifacts is not a substitute for using external artifact repositories such as Artifactory or Nexus and should be considered only for basic reporting and file archival.

### **Test**

Running automated tests is a crucial component of any successful continuous delivery process. As such, Jenkins has a number of test recording, reporting, and visualization facilities provided by a number of plugins. At a fundamental level, when there are test failures, it is useful to have Jenkins record the failures for reporting and visualization in the web UI. The example below uses the junit step, provided by the JUnit plugin.

In the example below, if tests fail, the Pipeline is marked "unstable", as denoted by a yellow ball in the web UI. Based on the recorded test reports, Jenkins can also provide historical trend analysis and visualization.

**Jenkinsfile (Scripted Pipeline)**

*node {  
 /\* .. snip .. \*/  
 stage('Test') {  
 /\* `make check` returns non-zero on test failures,  
 \* using `true` to allow the Pipeline to continue nonetheless  
 \*/  
 sh 'make check || true' // <1>  
 junit '\*\*/target/\*.xml' // <2>  
 }  
 /\* .. snip .. \*/  
}*

**Note:** Using an inline shell conditional (sh 'make || true') ensures that the sh step always sees a zero exit code, giving the junit step the opportunity to capture and process the test reports. Alternative approaches to this are covered in more detail in the Handling Failures section below.

junit captures and associates the JUnit XML files matching the inclusion pattern (\*\*/target/\*.xml).

### 

### 

### 

### 

### 

### 

### 

### **Deploy**

Deployment can imply a variety of steps, depending on the project or organization requirements, and may be anything from publishing build artifacts to an Artifactory server, to pushing code to a production system.

At this stage of the example Pipeline, both the "Build" and "Test" stages have successfully executed. In essence, the "Deploy" stage will only execute assuming previous stages completed successfully, otherwise the Pipeline would have exited early.

**Jenkinsfile (Scripted Pipeline)**

*node {  
 /\* .. snip .. \*/  
 stage('Deploy') {  
 if (currentBuild.result == 'SUCCESS') { // <1>  
 sh 'make publish'  
 }  
 }  
 /\* .. snip .. \*/  
}*

**Note:** Accessing the currentBuild.result variable allows the Pipeline to determine if there were any test failures. In which case, the value would be UNSTABLE.

Assuming everything has executed successfully in the example Jenkins Pipeline, each successful Pipeline run will have associated build artifacts archived, test results reported upon and the full console output all in Jenkins.

A Scripted Pipeline can include conditional tests (shown above), loops, try/catch/finally blocks and even functions. The next section will cover this advanced Scripted Pipeline syntax in more detail.