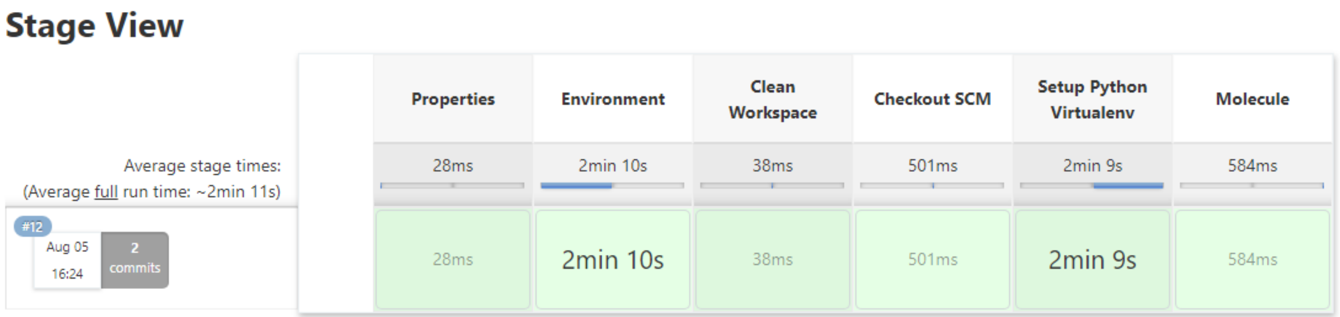
01 Jenkinsfile explained

This page gives you information about the concept of a Jenkinsfile and the different types of it. The Prerequisites for this Tutorial is a minimal understanding of Jenkins and what it is used for. If you have never heard about Jenkins or CI/CD in general, I would recommend to do some reading first. Here are two links that will explain the topic to you: CI/CD and Jenkins

Jenkinsfile

What Do I Need The Jenkinsfile For?

The Jenkinsfile is what describes and defines your pipeline. According to the Jenkinsfile, Jenkins assembles the pipeline and starts the build. In the end you should have a build that passes all the build steps. In Jenkins this might look light this:


Stage View

	Properties	Environment	Clean Workspace	Checkout SCM	Setup Python Virtualenv	Molecule
Average stage times: (Average full run time: ~2min 11s)	28ms	2min 10s	38ms	501ms	2min 9s	584ms
#12 Aug 05 16:24 2 commits	28ms	2min 10s	38ms	501ms	2min 9s	584ms

---

From Declarative Jenkinsfile To Jenkins-Shared-Libraries

Jenkinsfile Vocabulary

Before you can go on about the details of writing a Jenkinsfile you first need to learn about some of the vocabulary used here.

The Jenkinsfile can come in different appearances:

- **Declarative Pipline/Jenkinsfile:** The Jenkinsfile is written in a descriptive language.
- **Scripted Pipeline/Jenkinsfile:** The Jenkinsfile is written in groovy based scritping language.
- **Mixed Pipeline/Jenkinsfile:** The Jenkinsfile is written in a mix between declarative and scripted pipeline script. In this tutorial you won't go any deeper into that topic, instead you'll continue directly with Jenkins-Shared-Libraries.
- **Jenkins-Shared-Library:** All the Code used for our Jenkinsfile is contained in a Library. You import the Library into your Jenkinsfile and used the different pipeline steps as predefined methods.

DSL

Both declarative and scripted piplines are DSL's

Terms used in the Jenkins DSL:

- **pipeline:** A Pipeline is a user-defined model of a CD pipeline. A Pipeline's code defines your entire build process, which typically includes stages for building an application, testing it and then delivering it.
- **node/agent:** The node is the VM/Container/Server/Host on which the Pipelines is executed.
- **options:** Allows you to define Pipeline-Specific options. The Jenkins documentation provides a list of the most common options: Options and Properties.
- **environment:** This term describes the environment in which the Pipeline runs. For a *bash shell* you would then define the *shell variables* needed for our build.
- **stages:** A stage gives you the possibility to segregate different steps in your pipeline by topic. For example, all steps concerning unit test are int the stage called 'TEST'.
- **steps:** When you hear the word steps you think about the DSL specific functions you can use within a Jenkinsfile. There are steps like *make*, *echo* and *sh*. On your Jenkins there is a Pipeline-Syntax-Page where can have a look at all the steps available to your Jenkins host. When installing Plugins you may add more steps to your Jenkins. There is also a Pipeline Steps reference in the official Jenkins documentation.
- **directives:** Directives are areas in the code that describe how the Pipeline behaves. Hence, directives contain *steps* or do specifically describe the Pipeline like the *environment* directive.
- **sections:** Sections in Declarative Pipeline typically contain one or more Directives or Steps.

Now that you have your vocabulary, you can start looking into Jenkinsfiles themselves and how they are written. You will cover three different types of Jenkinsfile here, all these were already mentioned above:

- **Declarative Jenkinsfile**
- **Scripted Jenkinsfile**
- **Jenkinsfile with Jenkins-Shared-Library**

Declarative Jenkinsfile

Let's start with the declarative Jenkinsfile. Declarative means here that you describe your Pipeline with only predefined directives and steps. This is the "most user friendly" approach of writing a Jenkinsfile. Because id doesn't need skills in programming. Due to the userfriendliness your are limited in complexity. Describing complex building procedures is not easy in most cases impossible or just doable with additional plugins. This approach works perfect for small comprehensible projects but should be enhanced with scripted pipeline or even Jenkins-Shared-Library in projects that need scalability in size and complexity.

Let's have a closer look at the code below.

Sections and Directives

Declarative Jenkinsfiles always start with the *pipeline* section as you can see in he first line. With the *agent* section you describe on what agents this pipeline can be built. In your case you only want it to run on agents that have the 'my_agent' label. After the first two sections of the Pipeline you describe what you can call *meta directives* (no official term). there are a few other directives that fall in that category: *parameters*, *triggers*, *tools*, etc. For a full list of all directives and sections see the official documentation.

- The *options* directive describes the options/properties you want have for our pipeline.
- *Environment* describes all environment variables you need. This is done exactly the same way as you already know from our favorite unix shell.

Stages

Now you are coming to the next section called *stages*. As the name implies this section contains directives called *stages*. You can see now what is meant by segregation with stages. You segregated them here by build topics: Build, Test and Deploy. All the segregates are later shown in the Jenkins-UI when building a Pipeline, so you'll have a good graphical reference to your Jenkinsfile.

Steps

You have already covered steps in the vocabulary above. Inside a stage you have *steps.* Here You can see now how they are used. You can just write out the step you need and add the arguments within single quotes. Be aware that some steps use slightly different syntax like *sh* below. Some others might have syntax that reminds you more of a directive like the *script* step. I would recommend to consult the documentation if you are not sure what to do. For the Documentation see the three links above about pipeline syntax.

Mixed Jenkinsfile

The script step was mention here. This step allows you to write Groovy code inside a stage. that way you can get a mixed Jenkinsfile. You could also define methods and variables outside of the pipeline section scope und us those inside the script step. For more information consult the officila Jenkins documentation.

Jenkinsfile descreptive

```
pipeline {

    agent {
        label 'my_agent'
    }

    options {
        gitLabConnection('my_gitlab_connection')
    }

    environment {
        PATH=".local/bin:$PATH"
    }

    stages {
        stage('Build') {
            steps {
                echo 'Building..'
                sh '''
                    . ~/bin/build.sh
                '''
            }
        }
        stage('Test') {
            steps {
                echo 'Testing..'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying....'
            }
        }
    }
}
```

Scripted Jenkinsfile

The Scripted DSL give you the possibility to add complex structures to your Jenkinsfile. Scripts in you Pipeline are written in Groovy. Groovy is a object-oriented programming language for the Java Platform. If you're not familiar with Groovy or Java at all, it is recommended to go through a tutorial or introduction first: Groovy Tutorial, Java Tutorial. Scripted DSL uses the same vocabulary we already covered in the section above.

Methods and Global Variables

At the top of you Jenkinsfile you can define methods and global variables with def keyword. The def keyword is a generic identifiers for new declaration. The return type and the type will be determined automatically for you. It is also possible to declare a method or variables in the classic Java manner "*type name = value, type name methodName()*".

In the code below we just have one method called **buildWithTest**. This method awaits a Boolean called bool, the default value for bool is true. If bool is set true then a *sh* step is used to execute "~/bin/test.sh"

Inside node

In the example for the declarative Jenkinsfile you saw, that all the sections and directives are inside the *pipeline* section. In Scripted Jenkinsfile we use the *node* section instead. The node section here describes the same as the *agent* section in the declarative Pipeline. The name inside the quotes determines the host on which the build is run. *Stages* work quite similarly to declarative Pipeline, except that you don't need to have a *stages* section. You can simply put all yout stages inside the node section. Inside the a stage you can use steps or you predefined methods. The steps syntax works different here than the steps syntax in declarative DSL. The exact syntax can be generated with the Snippet Generator. The Snippet Generator also provides a little documentation, about the functionality of the different steps.

Segergation with Stages

The stages here are also used for segregation, like in the declarative Pipeline. Remember the segregation happens by topic: Clean Workspace, Checkout SCM, Build and Test. Don't forget the give meaningful names to your stages. If you can't name your stage in a meaningful way you should probably split it up or integrate into another step.

Scripted Jenkinsfile

```
def buildWithTest(Boolean bool = true) {
    if (bool) {
    sh (script: "~/bin/test.sh")
    }
}

node ('ansible') {
    stage('Clean Workspace')
        cleanWs()
    stage('Checkout SCM')
        checkout scm
    stage('Build')
        sh (script: "~/bin/build.sh")
    stage('Test')
        buildWithTest($BUILD_WITH_TEST) //$BUILD_WITH_TEST = false
}
```

Scripted Pipeline With Jenkins-Shared-Library

You know how Jenkins Pipelines are described and how to describe complex logic with Scripted DSL. With these tools you are now able to do almost everything with your Jenkinsfile. Of course there are some limitations of computation you cannot overcome. But since this Tutorial is about Jenkins and not the theory of computation, we don't really care 😉. Let's get back to the topic and first talk about what a Jenkins-Shared-Library is and how it's used. Then we go over the benefits of Jenkins-shared-Libraries why you should consider using Jenkins-Shared-Libraries especially in Project with high complexity and the need of scalability.

What the is a Library an how can I use it for my Jenkins Pipeline?

According to Wikipedia a Library in computer science is described as following:

"In computer science, a library is a collection of non-volatile resources used by computer programs, often for software development. These may include configuration data, documentation, help data, message templates, pre-written code and subroutines, classes, values or type specifications. In IBM's OS /360 and its successors they are referred to as partitioned data sets." Wikipedia - Library (computing): https://en.wikipedia.org/wiki/Library_(computing)

Load a Library and use Class

With a Jenkins-Shared-Library we are able to load resources into our Jenkinsfile. When we think about declarative/scripted DSL, we can use it load in predefined methods (subs) Classes or resources (JSON-Templates, bash scripts). In the Example below we can see that we first load the *'aveniq-jenkins-libraries'* Library with the *@Library* directive. When the library is loaded you can Import the Classes you need. In the example below we want to build the linux-configuration project so we only need to import LinuxConfiguration Class. The import is done as we already know it from Java. You might wonder now how you know what Classes exists and what they do. Don't worry this step will be covered in the next page of this tutorial. Next we create an object instance from our LinuxConfiguration class. now you can call methods from that class within your Jenkinsfile. Here is a list of the methods use blow in the code example:

- linuxConfiguration.getProjectName()
- linuxConfiguration.pythonVirtualEnvironment()
- linuxConfiguration.invokeMolecule()

Global Variables in Jenkins-Shared-Library

A Jenkins-Shared-Library does not only provide you with classes. It also may provide you with global variables. A global variables is groovy file that provides you with a subroutine. In our example there are two global variables we use: options and environment.

- options takes the project name as argument, it will then set the options for the pipeline quite similar to the *options* directive in the declarative DSL.
- environment takes the project name as argument, it will then create the environment needed to build that project. This global variable is quite similar to the *environment* directive in the declarative DSL.

Scope of Jenkins-Shared-Libraries

Subroutines that come from a Jenkins-Shared-Library can be used either within declarative or scripted Jenkinsfile. Jenkins-Shared-Libraries but are always written in Groovy. So if you want to use Libraries no way to come around writing Groovy code or at least understanding it.

Benefits of Jenkins-Shared-Libraries

As long as you use Jenkins for just on build there is no need for Libraries. You can just all your build steps in a single Jenkinsfile and you're good to go. Normally it won't be that easy, you have multiple projects with different versions. For each Project and version you need a different Jenkinsfile because they need to be build differently. Imagine then starting to copy past some subroutines or stages from one Jenkinsfile into another and even adjust the code of it. For two subroutines in two files this seem not like that much of a big deal. But it will become as soon as you add more Jenkinsfiles in more projects and do more copy pasting. It's very likely that you will end up in a big mess of different Jenkinsfiles. Some are written just with declarative DSL some just with scripted DSL and some others that are mixed.

Here Jenkins-Shared Libraries provide a big remedy. You can organize all your Code within a centralized repository. That way you can maintain your code in a consistent way. If there is need for a new logic in all Jenkinsfile. You just add a new global variable to the library. Voila, the new subroutine is now available in all Jenkinsfiles. With Jenkins-Shared-Libraries it is also possible to unit-test your code. So you can make sure that global variables that are used by all Jenkinsfile won't just break because of change. In general you can say that Libraries scale better for big projects. For small projects the overhead of a library might be to large.

Scripted Pipeline With Jenkins-Shared-Library

```
@Library('aveniq-jenkins-libraries')
import ch.aveniq.jenkins_libraries.ansible.projects.linux_configuration.LinuxConfiguration
def linuxConfiguration = new LinuxConfiguration()

node('ansible') {
  stage('Properties')
    options.setOptions(linuxConfiguration.getProjectName())
  stage('Environment')
    environment.setEnvironment(linuxConfiguration.getProjectName())
  stage('Clean Workspace')
    cleanWs()
  stage('Checkout SCM')
    checkout scm
  stage('Setup Python Virtualenv')
    linuxConfiguration.pythonVirtualEnvironment()
  stage('Molecule')
    linuxConfiguration.invokeMolecule()
    }
```

Legend

sections

directives

steps

miscellaneous technologies and terms

https://www.jenkins.io/doc/book/pipeline/

https://www.jenkins.io/doc/book/pipeline/jenkinsfile/