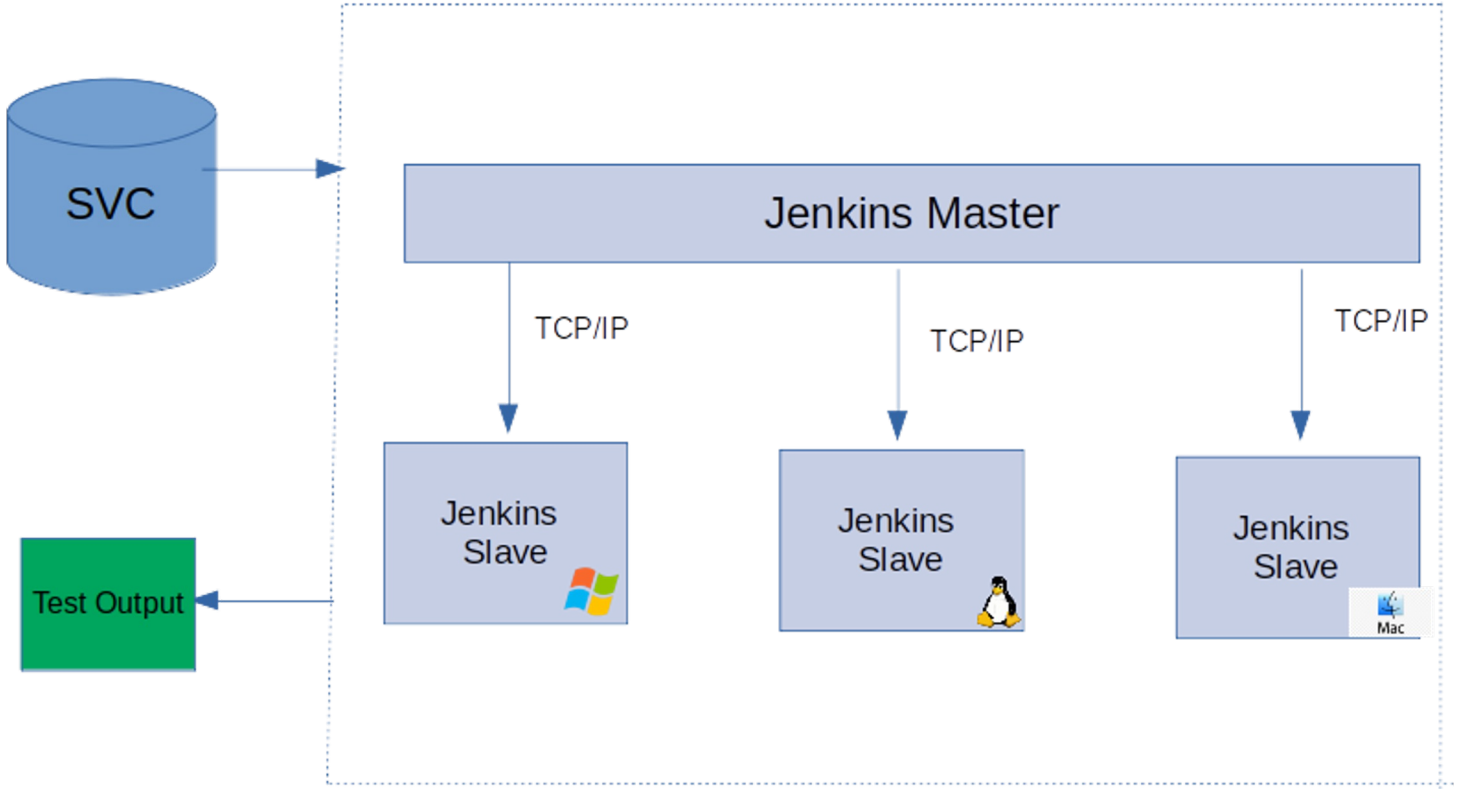

SVC
Jenkins Master
TCP/IP
TCP/IP
TCP/IP
Jenkins Slave
Jenkins Slave
Jenkins Slave
Mac
Test Output

Jenkins Master

The Master’s job is to handle:

- Scheduling build jobs.
- Dispatching builds to the slaves for the actual execution.
- Monitor the slaves (possibly taking them online and offline as required).
- Recording and presenting the build results.
- A Master instance of Jenkins can also execute build jobs directly*.

Jenkins Slave

- It hears requests from the Jenkins Master instance.
- Slaves can run on a variety of operating systems.
- The job of a Slave is to do as they are told to, which involves executing build jobs dispatched by the Master.
- You can configure a project to always run on a particular Slave machine, or a particular type of Slave machine, or simply let Jenkins pick the next available Slave*.

How to upgrade Jenkins (in Docker) correctly - plugins ?

- By default, plugins will be upgraded if they haven't been upgraded manually and image version is newer than that in container
- To force upgrades of plugins that have been manually upgraded, run the docker image with **-e PLUGINS_FORCE_UPGRADE=true**
- Upgraded plugins from UI won't be reverted on next start - eventually you can change it by adding .override to file /usr/share/jenkins/ref/config.xml.override

Useful plugin - Lockable resource plugin

- Use it when a few projects have a common resource that can be used only by one at given moment

Useful plugin - Purge job history plugin

- You can use it when you want to delete all jobs from history

Useful plugin - Job Configuration History

- **Check who updated your job and restore when needed !**

Multi-configuration (matrix) project type

- Requires the **Jenkins Matrix Project plugin**
- Matrix job let you build the same project in many different configurations
- Useful for testing or deploying an application in many different environments, with different databases, or even on different build machines.

Configuration matrix

- The **Configuration Matrix** allows you to specify what steps to duplicate, and create a multiple-axis graph of the type of builds to create.
- You can define a User Defined Axis to create your matrix of combinations.

Default Axes types for Matrix

- label expressions (label-expression)
- user-defined values (user-defined)
- slave name or label (slave)
- JDK name (jdk)

Additional plugin-defined axes

Each of below axes types requires plugin:

- DynamicAxis (dynamic), requires the Jenkins DynamicAxis Plugin
- GroovyAxis (groovy), requires the Jenkins GroovyAxis Plugin
- YamlAxis (yaml), requires the Jenkins Yaml Axis Plugin

Do we really want deploy each version on each env ? – Matrix Combination Filter

- Combination Filter let us write expression to run only a few combination of axis
- Below expression run:
  - Master version for PROD environment
  - Develop version for STAGE environment
  - release-1.0 for TEST environment
  - release-2.0 for DEV environment

Expression : (env=="PROD" && version=="master") || (env=="STAGE" && version=="develop") || (env=="TEST" && version=="release-1.0") || (env=="DEV" && version=="release-2.0")

What is Jenkins Pipeline ?

- Jenkins pipeline is a group of events or jobs which are interlinked with one another in a sequence
- Project build/test/deploy steps defined in a Jenkinsfile
- Requires Jenkins Pipeline Plugin (Built-in in fresh Jenkins installation)

What is Jenkinsfile ?

- Pipelines are defined using text file called Jenkinsfile
- Pipeline as a code is defined with Jenkinsfile and this code can be defined using domain specific language (DSL)

Key features of the Jenkins Pipeline

- Deployment flow defined as a code
- Multiple “standard” Jenkins jobs can be expressed in one script
- Simplicity in defining commonly used tasks
- Storing pipeline configuration (Jenkinsfile) in source control, versioning, independent tests
- As plugin uses Groovy each operation can be defined with relative ease (Conditions, loops, variables)
- We can use Groovy DSL to access almost any existing plugins and Jenkins Core features

Declarative vs Scripted Pipeline

Declarative:

- Easy to use
- Concise
- Validation before running
- Visual editor
- Syntactic and semantic validation available

Perfect for regular users

Scripted:

- Raw Apache Groovy with a little bit of Jenkins flavor
- Groovy syntax is only limitation
- Can create pretty unreadable pipeline code and cause a maintenance headache.
- Not everything you do in a scripted pipeline will render well in the Jenkins UI.

More flexibility for power users

Pipeline Syntax Documentation

- https://www.jenkins.io/doc/book/pipeline/syntax/
- https://www.jenkins.io/doc/pipeline/steps/

Declarative Pipeline

```
// Declarative //
pipeline {
    agent { docker 'maven:3-alpine' }
    stages {
        stage('Example Build') {
            steps {
                sh 'mvn -B clean verify'
            }
        }
    }
}
// Script //
```

Declarative Pipeline Syntax

A valid Declarative pipeline must be defined with the “pipeline” sentence and include the next **required** sections:

- agent
- stages
- stage
- steps

Optional:

- directives (environment, options etc.)

Agent

Specifies where pipeline or specific stage will be executed.
Agent section:

- **must** be defined on top-level pipeline block
- **can** be defined for each stage (optional)

Agent section parameters

- any
- none
- label
- node (similar behavior as label)
- docker
- dockerfile

Some of parameters has additional options (for example node can use `customWorkspace` option)

Docker agent on top level – Docker plugin required

```
pipeline {
    agent {
        docker {
            image 'maven:3-alpine'
            args '-v $HOME/.m2:/root/.m2'
        }
    }
```

Docker agent on stage level

1. Don’t allocate agent on top level

```
pipeline {
    agent none ❶
    stages {
        stage('Example Build') {
            agent { docker 'maven:3-alpine' } ❷
            steps {
                echo 'Hello, Maven'
                sh 'mvn --version'
            }
        }
        stage('Example Test') {
            agent { docker 'openjdk:8-jre' } ❸
            steps {
                echo 'Hello, JDK'
                sh 'java -version'
            }
        }
    }
}
```

2. Allocation docker agent for stage “Example Build”

3. Allocation docker agent for stage “Example Test”

Stages

The *stages* section contains one or more *stage* blocks

- Think of each *stage* block as like an individual Build Step in Freestyle job
- There need to be a *stages* section present in your *pipeline* block

Simple pipeline including stages

```
pipeline {
    agent any
    stages {
        stage ('example') {
            steps {
                echo "hello"
            }
        }
    }
}
```

Steps

- Defines a series of steps to be executed in a given stage directive.
- Must contain one or more steps.

Step Syntax in Pipeline

Example with "readFile" (built-in (basic) step)

- Shorthand syntax:

```
readFile 'build.properties'
```

- Syntax when passing multiple parameters

```
readFile file:'build.properties', encoding: 'ISO-8859-1'
```

=

```
readFile ([file:'build.properties', encoding: 'ISO-8859-1'])
```

Step Syntax in Pipeline

Example with "checkout" step offered by SCM plugin

- ***Git*** step is kind of checkout step

```
git url: 'git 'https://github.com/helm/helm''
```

- ***Checkout*** step provides more functionality than ***git*** step
  - We have to provide class because this step offers different classes for different SCM

```
checkout([$class: 'GitSCM', branches: [[name: '*/master']],
userRemoteConfigs: [[url: 'http://git-
server/user/repository.git']]])
```

- Check out *https://www.jenkins.io/doc/pipeline/steps/workflow-scm-step/*

Declarative Directive Generator - DEMO

Built-in tool that allows generate Pipeline code for a Declarative Pipeline directive.

Go to your pipeline project -> Find and click **“Pipeline Syntax”** button on the top right options -> Choose **“Declarative Directive Generator”**

Snippet Generator - DEMO

Easy to use built-in tool that helps you generate you Pipeline Code used in various steps

Go to your pipeline project -> Find and click **“Pipeline Syntax”** button on the top right options -> Choose **“Snippet Generator”**

Jenkins folders – jobs view isolation

- This allows us to manage the jobs much like we would files on a file system. Folders can also be used to manage permissions on a per folder basis to ease security administration.

Optional pipeline directives

- environment (Defined at stage or pipeline level)
- input (Defined at stage level)
- options (Defined at stage or pipeline level)
- parallel
- parameters
- post
- script
- tools
- triggers
- when

Environment

The **environment** directive specifies a sequence of key-value pairs which will be defined as environment variables

- **environment** directive used in the top-level **pipeline** block will apply to all steps within the Pipeline.
- **environment** directive defined within a **stage** will only apply the given environment variables to steps within the **stage**
- The **environment** block has a helper method **credentials()** defined which can be used to access pre-defined Credentials by their identifier in the Jenkins environment.

```
pipeline {
    agent any
    environment {
        CC = 'clang'
    }
    stages {
        stage('Example') {
            environment {
                DOCKER_REGISTRY = credentials('docker-registry')
            }
            steps {
                println CC
                println DOCKER_REGISTRY_USR
                println DOCKER_REGISTRY_PSW
            }
        }
    }

}
```

Handling credentials

Jenkins credential's helper method **"credentials()"** can be used with:

- **secret text**
- **user names and passwords**
- **secret files**

For other type use:

- ***withCredentials*** step

Options

The **options** directive allows configuring Pipeline-specific options from within the Pipeline itself.

- ***buildDiscarder (global)***
- ***disableConcurrentBuilds (global)***
- ***skipDefaultCheckout (global)***
- ***timeout (global, stage-level)***
- ***retry (global, stage-level)***
- ***timestamps (global, stage-level)***

```
pipeline {
    agent any
    options {
        timeout(time: 1, unit: 'HOURS')
        buildDiscarder(logRotator(numToKeepStr: '3'))
        timestamps()
    }
    stages {
        stage('Example') {
            steps {
                echo 'Hello World'
            }
        }
    }
}
```

Parallel

Parallel directive allows to run multiple stages in parallel. It is defined at stage level.

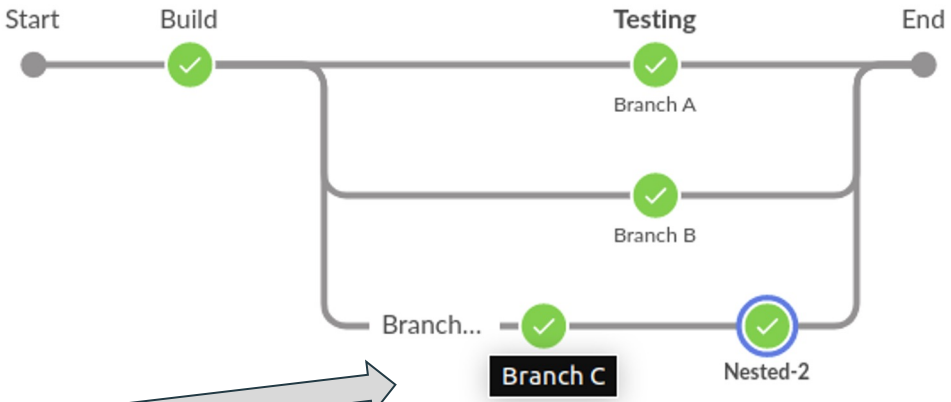
Restrictions when using parallel directive in stage:

- A stage directive can have either a **parallel** or **steps** directive but not both.
- A stage directive inside a parallel one cannot nest another parallel directive, only **steps** are allowed.
- Stage section that have a parallel directive inside cannot have **“agent”** or **“tools”** directives defined.

Sequential build with parallel

```
stage ('Branch C') {
    stages {
        stage('Nested-1') {
            steps {
                echo "Nested-1"
            }
        }
        stage('Nested-2') {
            steps {
                echo "Nested-2"
            }
        }

    }
}
```



Parameters

The **parameters** directive provides a list of parameters which a user should provide when triggering the Pipeline. Only one definition is allowed.

Parameters types:

- string
- text
- booleanParam
- choice
- password

Active Choices Parameter

A Jenkins UI plugin for generating and rendering multiple value options for a job parameter. The parameter options can be dynamically generated from:

- Groovy Script
- Scriptler Script

Script step

- Script step takes a block of **Scripted Pipeline** and executes that in the **Declarative Pipeline**.
- Extends functionality of Declarative Pipeline

Script directive example

```
parameters {
    choice (name: 'Environment', choices:['dev','stage','prod'], description: 'Environment name for deployment')
}
stages {
    stage('Deploy') {
        steps {
            script {
                if (params.Environment == "dev") {
                    echo "Deploying on dev"

                } else if (params.Environment == "stage") {
                    echo "Deploying on stage"

                } else if (params.Environment == "prod") {
                    echo "Deploying on prod"

                }
            }
        }
    }
}
```

When directive

When directive allows the Pipeline to determine whether the stage should be executed depending on the given condition.

- Define inside a stage directive
- Must contain at least one condition when multiple defined logical AND is applied

Most popular Built-in “when” conditions

- branch (Multi-branch pipeline)
- tag
- environment
- expression
- triggeredBy

Evaluation condition for when directive

By default when condition :

- for a **stage** will be evaluated after entering the **agent** for that **stage !**
  - to change default activate - when { **beforeAgent true** ...}
- condition for a stage will not be evaluated before the input, if one is defined
  - to change default activate - when { **beforeInput true** }
- for a **stage** will be evaluated after entering the **options** for that **stage**
  - to change default activate - when { **beforeOptions true** }

Post

Can be added at pipeline or stage level. Sentences included in it are executed once the stage or pipeline completes.

Post-conditions can be used to control whether the post executes or not. Some of post conditions:

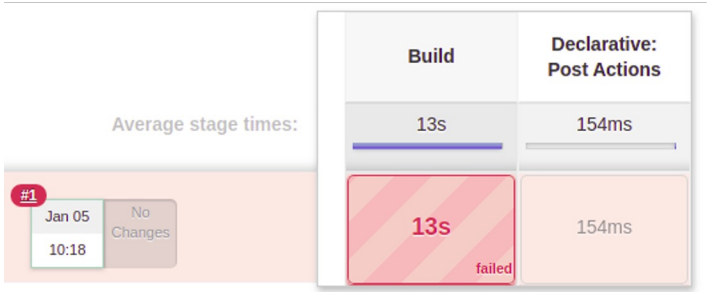
- always - Steps are executed regardless of the completion status.
- changed: Executes only if the completion results in a different status than the previous run.
- failure: Steps are executed only if the pipeline or stage fails.
- success: Steps are executed only if the pipeline or stage succeeds.
- unstable: Steps are executed only if the pipeline or stage is unstable.

Example with Post

```
stages {
    stage('Build') {
        steps {
            sleep 10
            echo "Not ok"
            sh 'exit 1'
        }
    }


}
post {
    failure {
    echo "Post execution will be executed always"
    }
}
```

Post executed



Triggers

Allows to automatically trigger pipelines. Pipelines can be triggered by following ones:

- cron
- pollSCM
- upstream

Global Variables

https:// jenkins.mgworkshop.eu /pipeline-syntax/globals

Global Variable

- Available directly in Pipeline
- Expose methods and variables in Pipeline script

```
pipeline {
    agent any
    }
    stages {
        stage('Image pull') {
            steps {
                script {
                    myImageAlpine = docker.image('alpine:latest')
                    myImageAlpine.pull()
                }
            }
        }
    }
}
```

In-process script approval

Jenkins pipeline script can be run:

- **outside Groovy Sandbox** - entire script have to be approved by administrator
- **inside Groovy Sandbox** - every method call (other than native Pipeline steps provided by Jenkins) have to be approved

Pipeline script loaded from SCM can be run only inside Groovy Sandbox

How to catch exception in Pipeline?

- Catch exception in your script
- Catch exception in Jenkins Pipeline
  - try/catch
  - catchError
  - warnError

catchError pipeline step

catches Error and sets build result to failure with default option

Additional options:

- add message for error:
  - catchError(message:"There was error") {sh "script.sh}
- change default behavior:
  - catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE') {

try/catch/finally blocks

- try/catch/finally must be placed in ***script*** step of pipeline

```
script {
    try {
        sh 'exit 1'
    }
    catch (Exception e) {
        println "exception ${e}"
    }
    finally {
        sh 'exit 0'
        currentBuild.result = 'UNSTABLE'
        echo "RESULT: ${currentBuild.result}"
    }
}
echo "RESULT: ${currentBuild.result}"
}
```

Shared libraries

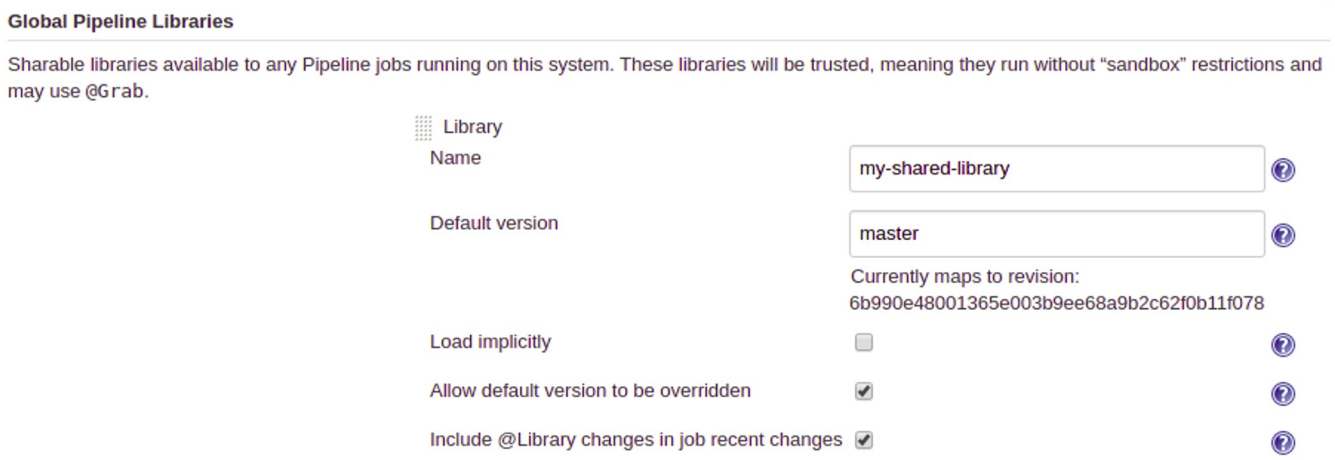
“Jenkins shared library is the concept of having a pipeline code in the version control system that can be used by any number of pipeline just by referencing it” ~ devopscube

Directory Structure of a Shared Library

```
(root)
+- src                     # Groovy source files
|   +- org
|       +- foo
|           +- Bar.groovy  # for org.foo.Bar class
+- vars
|   +- foo.groovy          # for global 'foo' variable
|   +- foo.txt             # help for 'foo' variable
+- resources               # resource files (external libraries only)
|   +- org
|       +- foo
|           +- bar.json    # static helper data for org.foo.Bar
```

Configuration Shared Libraries

Manage Jenkins » Configure System » Global Pipeline Libraries



Using Shared Libraries

- If you use ***"Load implicitly"*** option Pipelines can immediately use classes or global variables defined by any such libraries.
- To use Shared Library ***Jenkinsfile*** need to use ***@Library*** annotation (1)
- If you want to use ***src/*** directory with classes you should import your classes (2)

```
@Library('my-shared-library') _
/* Using a version specifier, such as branch, tag, etc */
@Library('my-shared-library@1.1') _
/* Accessing multiple libraries with one statement */
@Library(['my-shared-library', 'terraform-library@develop']) _
```

1

```
@Library('some-library')
import com.mycorp.pipeline.somelib.UsefulClass
```

2

Library versions

- ***“Default version”*** from Shared Library configuration is used only when :
  - Load implicitly is defined
  - pipeline references library only by name
- If ***“Default version”*** is not defined pipeline **must** specify version - *@Library(‘my-shared-library@master’) _*
- *If* ***"Allow default version to be overridden"*** is enabled you can override default version from configuration

Setting up a backup policy

Finding your $JENKINS_HOME for operating system that you use:

1. Windows - Jenkins will be installed as a service and the default *$JENKINS_HOME* will be "C:\Program Files (x86)\jenkins".
2. Ubuntu/Debian - By default, the *$JENKINS_HOME* will set to "/var/lib/jenkins" and your $JENKINS_WAR will point to "/usr/share/jenkins/jenkins.war".

```
JENKINS_HOME
 +- config.xml     (Jenkins root configuration file)
 +- *.xml          (other site-wide configuration files)
 +- identity.key   (RSA key pair that identifies an instance)
 +- secret.key     (deprecated key used for some plugins' secure operations)
 +- secret.key.not-so-secret  (used for validating _$JENKINS_HOME_ creation date)
 +- userContent    (files served under your https://server/userContent/)
 +- secrets        (root directory for the secret+key for credential decryption)
     +- hudson.util.Secret   (used for encrypting some Jenkins data)
     +- master.key           (used for encrypting the hudson.util.Secret key)
     +- InstanceIdentity.KEY (used to identity this instance)
 +- fingerprints   (stores fingerprint records, if any)
 +- plugins        (root directory for all Jenkins plugins)
     +- [PLUGINNAME]   (sub directory for each plugin)
         +- META-INF       (subdirectory for plugin manifest + pom.xml)
         +- WEB-INF        (subdirectory for plugin jar(s) and licenses.xml)
     +- [PLUGINNAME].jpi   (.jpi or .hpi file for the plugin)
 +- jobs           (root directory for all Jenkins jobs)
     +- [JOBNAME]      (sub directory for each job)
         +- config.xml     (job configuration file)
         +- workspace      (working directory for the version control system)
         +- latest         (symbolic link to the last successful build)
         +- builds         (stores past build records)
             +- [BUILD_ID]     (subdirectory for each build)
                 +- build.xml      (build result summary)
                 +- log            (log file)
                 +- changelog.xml  (change log)
     +- [FOLDERNAME]   (sub directory for each folder)
         +- config.xml     (folder configuration file)
         +- jobs           (sub directory for all nested jobs)
```

Backup Strategy

1. Exclude archiving following folders:

- /war (the exploded Jenkins war directory)
- /cache (downloaded tools)
- /tools (extracted tools)

1. Job configuration backup - config.xml for each job
2. Jenkins master configuration - config.xml, *.xml files in root directory
3. Plugins - plugins root directory
4. Secrets

Credentials domains

- Logical separation credentials that are listed for some plugins or hosts
- Simplify configuration for the user
- Don’t restrict access to credentials in any way

Authorization for projects

- Matrix Authorization Strategy Plugin (All projects authorization)
- Authorize Project (Per project authorization)
- Role-based Authorization Strategy plugin (Per role authorization)

**Default mode: “Logged-in users can do anything”*

Best practices for Securing Jenkins

- Segment Jenkins jobs according to credential usage
- Use Non-privileged containers as Jenkins Agents if you can
- Separate identities and secrets for build
- Don’t use executor on master. Treat master as fortress
- Don’t give permissions for configuring jobs if it’s not required