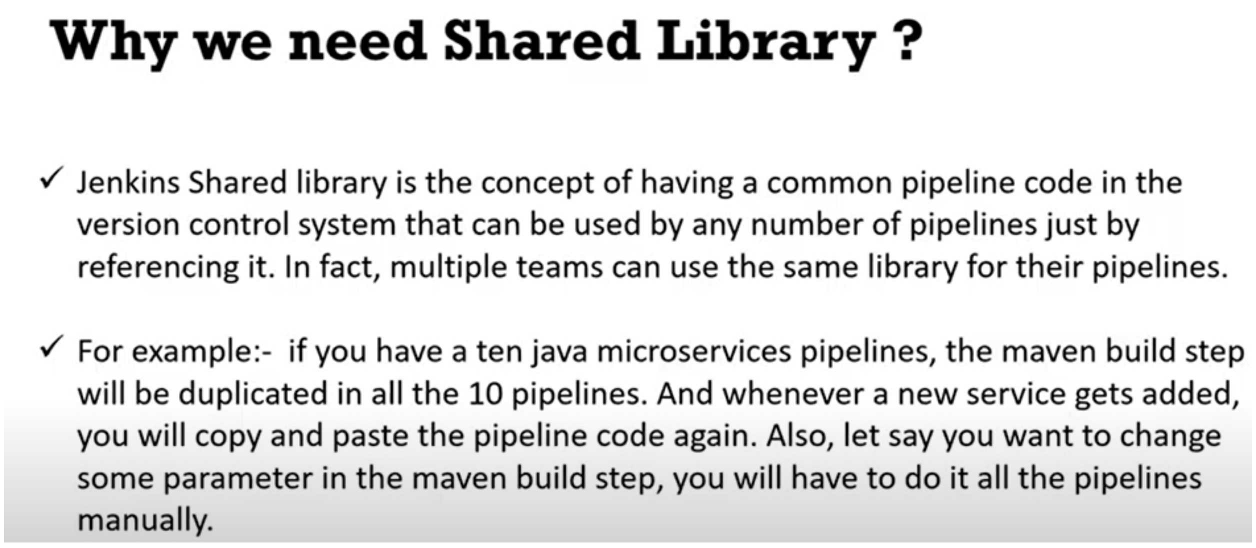
Reusable Jenkins Pipelines

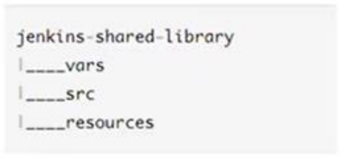
- ❖ As Pipeline is adopted for more and more projects in an organization, common patterns are likely to emerge.
- ❖ Oftentimes it is useful to share parts of Pipelines between various projects to reduce redundancies and keep code "DRY”.
- ❖ Pipeline has support for creating "Shared Libraries" which can be defined in external source control repositories and loaded into existing Pipelines.



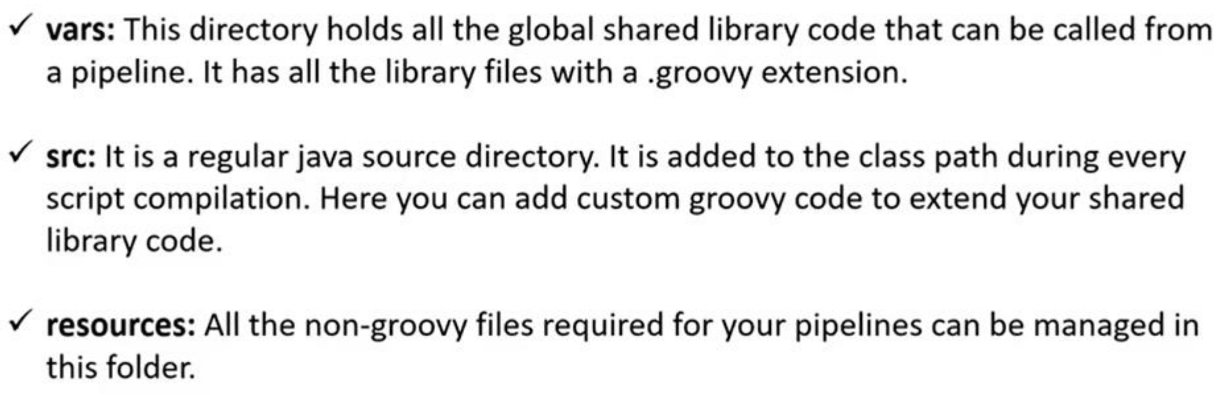
Why we need Shared Library ?

- ✓ Jenkins Shared library is the concept of having a common pipeline code in the version control system that can be used by any number of pipelines just by referencing it. In fact, multiple teams can use the same library for their pipelines.

- ✓ For example:- if you have a ten java microservices pipelines, the maven build step will be duplicated in all the 10 pipelines. And whenever a new service gets added, you will copy and paste the pipeline code again. Also, let say you want to change some parameter in the maven build step, you will have to do it all the pipelines manually.



```
jenkins-shared-library
|____vars
|____src
|____resources
```



- ✓ **vars:** This directory holds all the global shared library code that can be called from a pipeline. It has all the library files with a .groovy extension.

- ✓ **src:** It is a regular java source directory. It is added to the class path during every script compilation. Here you can add custom groovy code to extend your shared library code.

- ✓ **resources:** All the non-groovy files required for your pipelines can be managed in this folder.

Scenario :-

After the maven build if specified word found console, for specified number of times or more, then set build to unstable.

Scenarios #1 - Jenkins Reusable Pipeline

- ❖ When we build with Maven, we always use a reusable portion of the Jenkins pipeline to check on specified words on the Jenkins logs displayed on Jenkins UI. We aim to build software that is stable, secure, and with fewer maintenance issues.
- ❖ A Reusable Jenkins pipeline helps my company check how many WARNINGS appear in Jenkins logs during the build automation process with Maven.
- ❖ We aim to ensure we have one or no WARNING words in our logs.
- ❖ We use the Vars folder to hold groovy code. Our groovy code is written in bash language. We usually keep the groovy code in Jenkins shared library where we can call them frequently and multiple times when building different java-based projects.

Scenario #2

- ❖ We use maven to build because maven provides project information (log document, dependency list, unit test reports etc.).

- ❖ Logs are critical because they help us troubleshoot any build that may fail. Maven is very helpful for projects we run because we can update the central repository of JARs and other dependencies without necessarily engaging in scripting.

- ❖ With the help of the Maven automation tool, we can build any number of projects into output types like the JAR and WAR.

- ❖ We can then package the binaries build made by Jenkins into different applications, such as image applications which we can deploy to the staging area of production for further testing and configuration.

- ❖ We usually download the maven dependencies from a centralized repository. We store the groovy code in the Vars folder.

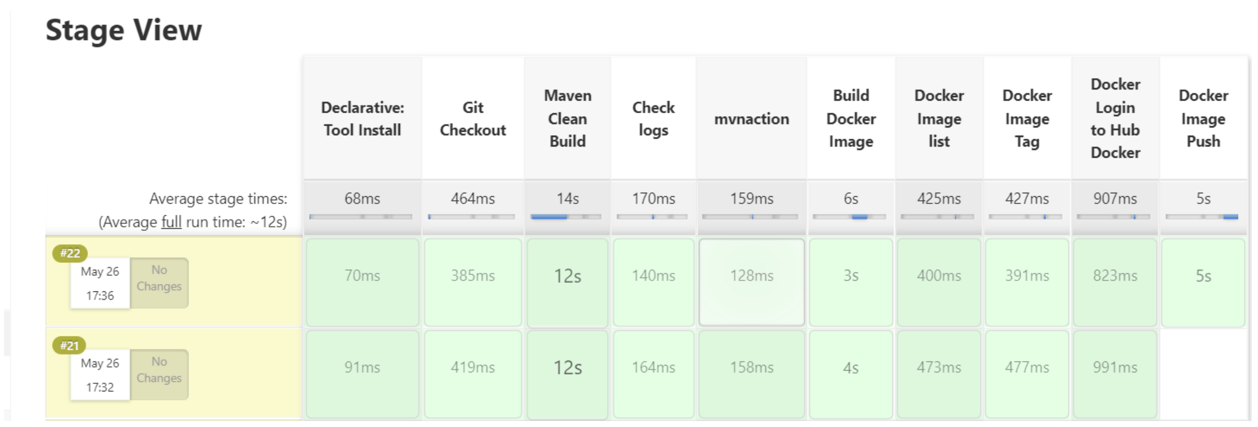
❖ Our groovy code is written in bash language. We have variables that define the Maven tool name in the Global tool configuration, define the tool name as maven, and define the command to run to publish project packages as dependencies from the Maven centralized repositories.

❖ This is important as it helps us reuse the groovy code in multiple pipelines by simply calling the groovy name file. This strategy is also vital because it helps keep our CI/CD declarative pipeline script dry and not mashed with so many lines of code.

The maven dependencies downloaded from the central repository are essential in accomplishing the following tasks:

1. Compiling, connecting, and packaging the code.
2. Converting src to binary code, testing, retesting, compiling, and packaging into a binary artifact.

The groovy code is always in the Jenkins global shared library and can be used by any team member running CI/CD in control Jenkins.



Stage View

	Declarative: Tool Install	Git Checkout	Maven Clean Build	Check logs	mvnaction	Build Docker Image	Docker Image list	Docker Image Tag	Docker Login to Hub Docker	Docker Image Push
Average stage times: (Average full run time: ~12s)	68ms	464ms	14s	170ms	159ms	6s	425ms	427ms	907ms	5s
#22 May 26 17:36 No Changes	70ms	385ms	12s	140ms	128ms	3s	400ms	391ms	823ms	5s
#21 May 26 17:32 No Changes	91ms	419ms	12s	164ms	158ms	4s	473ms	477ms	991ms	

What is POM XML?

A Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project. It contains default values for most projects.

How is POM XML generated?

xml. The POM contains information about the project and various configuration detail used by Maven to build the project(s). Before creating a POM, we should first decide the project group (groupId), its name (artifactId) and its version as these attributes help in uniquely identifying the project in repository.

How does POM XML work?

The pom. xml file contains information of project and configuration information for the maven to build the project such as dependencies, build directory, source directory, test source directory, plugin, goals etc. Maven reads the pom. xml file, then executes the goal.

What is properties in POM XML?

Properties can be defined in a POM or in a Profile. The properties set in a POM or in a Maven Profile can be referenced just like any other property available throughout Maven. User-defined properties can be referenced in a POM, or they can be used to filter resources via the Maven Resource plugin.

package will compile your code and also package it. For example, if your pom says the project is a jar, it will create a jar for you when you package it and put it somewhere in the target directory (by default).

install will compile and package, but it will also put the package in your local repository. This will make it so other projects can refer to it and grab it from your local repository.

```
$ mvn clean install -Dmaven.test.skip=true     OR     $ mvn clean package -Dmaven.test.skip=true
```

How to Include the test

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>${maven-surefire-plugin.version}</version>
    <configuration>
        <includes>
            <include>src/test/java/**/*Test.java</include>
        </includes>
    </configuration>
</plugin>
```