How To - Modular Pipeline Library Jenkins (Shared Library)

Overview:

Jenkins, automation platform or CI platform, recently received some updates to Pipeline, which is a suite of plugins that support the integration and implementation of pipelines into Jenkins. Pipeline is built with Groovy, and has a simple syntax and interface, so it's very easy to use. These updates allow us to create one Jenkinsfile that describes the entire pipeline, and the steps that need to be executed with a series of self-explanatory scripts. This increases the visibility of CI/CD automation processes for end users, and improves supportability by DevOps teams.

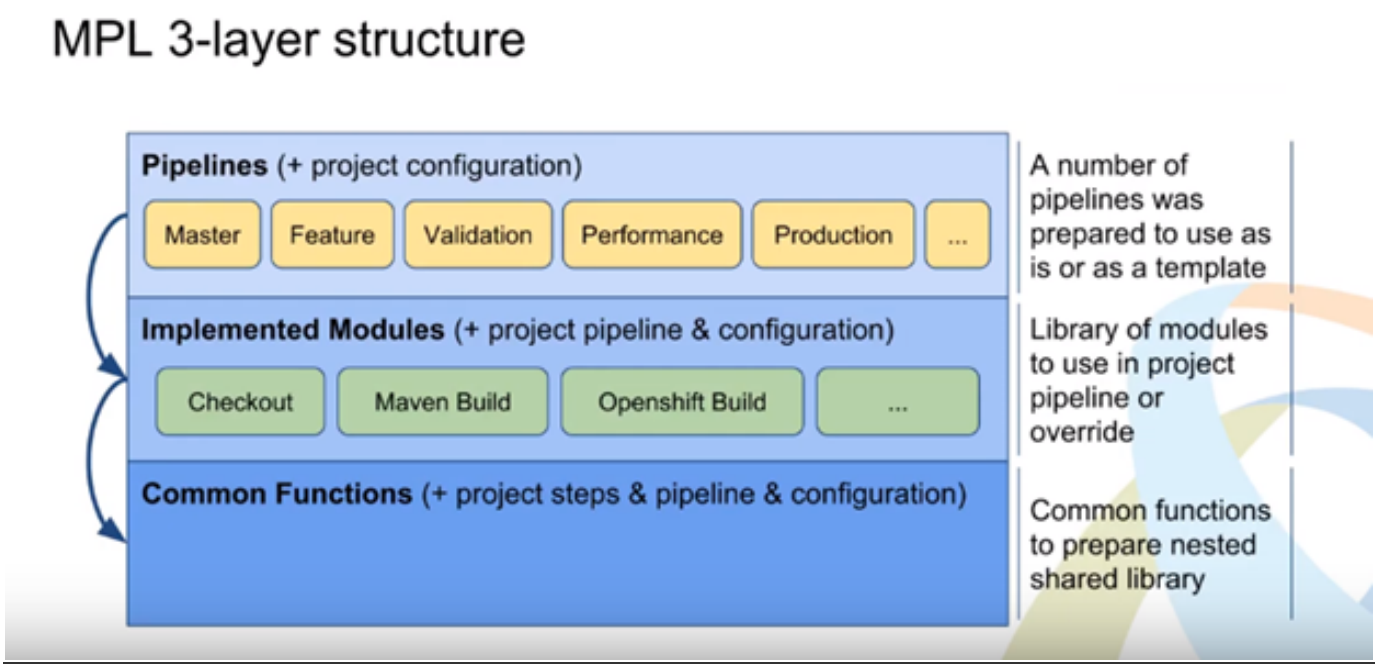
However, there's a large issue with Pipeline: it's hard to support multiple Jenkinsfiles (and therefore multiple projects) with unique pipelines. We need to store the common logic somewhere, which is where Jenkins Shared Libraries come in. They are included in the Jenkinsfile, and allow the use of prepared interfaces to simplify automation and store common pieces.

While shared libraries allow you to store logic and manipulate Jenkins, they don't provide a good way to utilize all the common information. The MPL (Modular Pipeline Library), therefore, optimizes the pipeline and shared libraries by allowing users to create easy-to-follow descriptions for processes, which are then stored for later use by other teams.

MPL provides a simple way to:

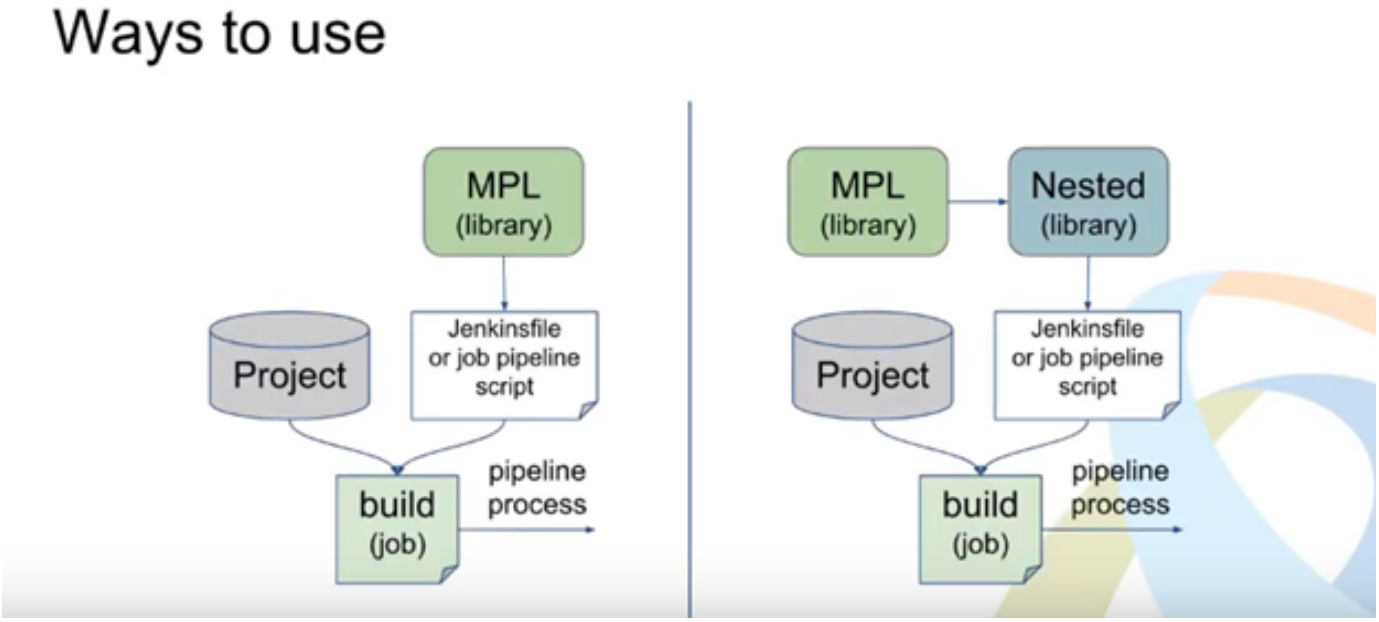
1. Separate pipelines and steps by introducing modules
2. Describe steps in the modules with an easy configuration interface
3. Test the described modules and share the results with other pipelines and projects

Layers of MPL:

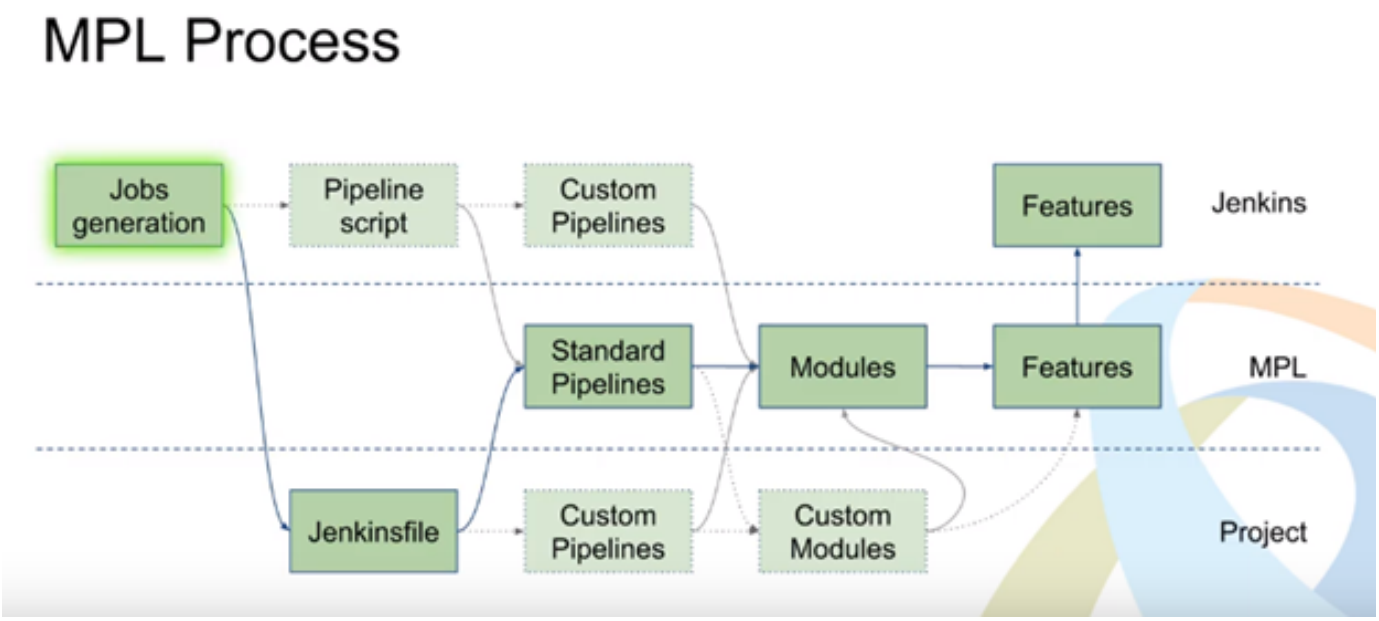


Way to Use:

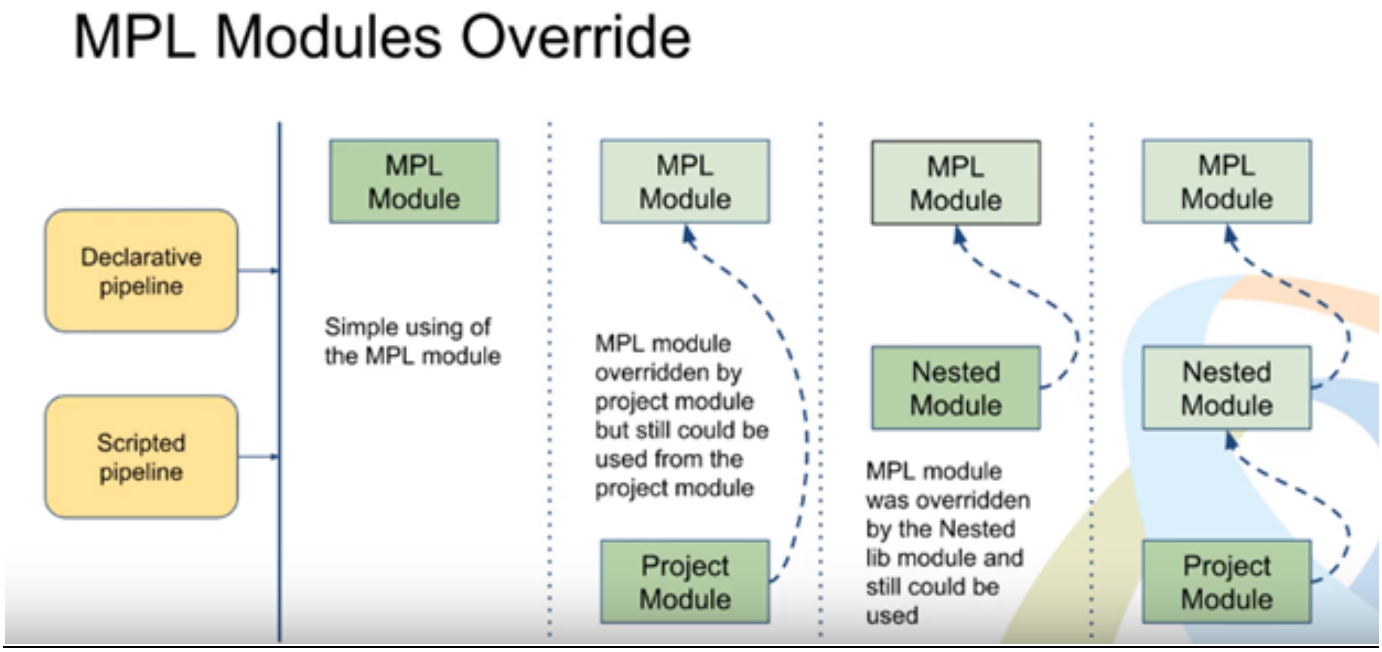
MPL can be used in a simple way or nested way-



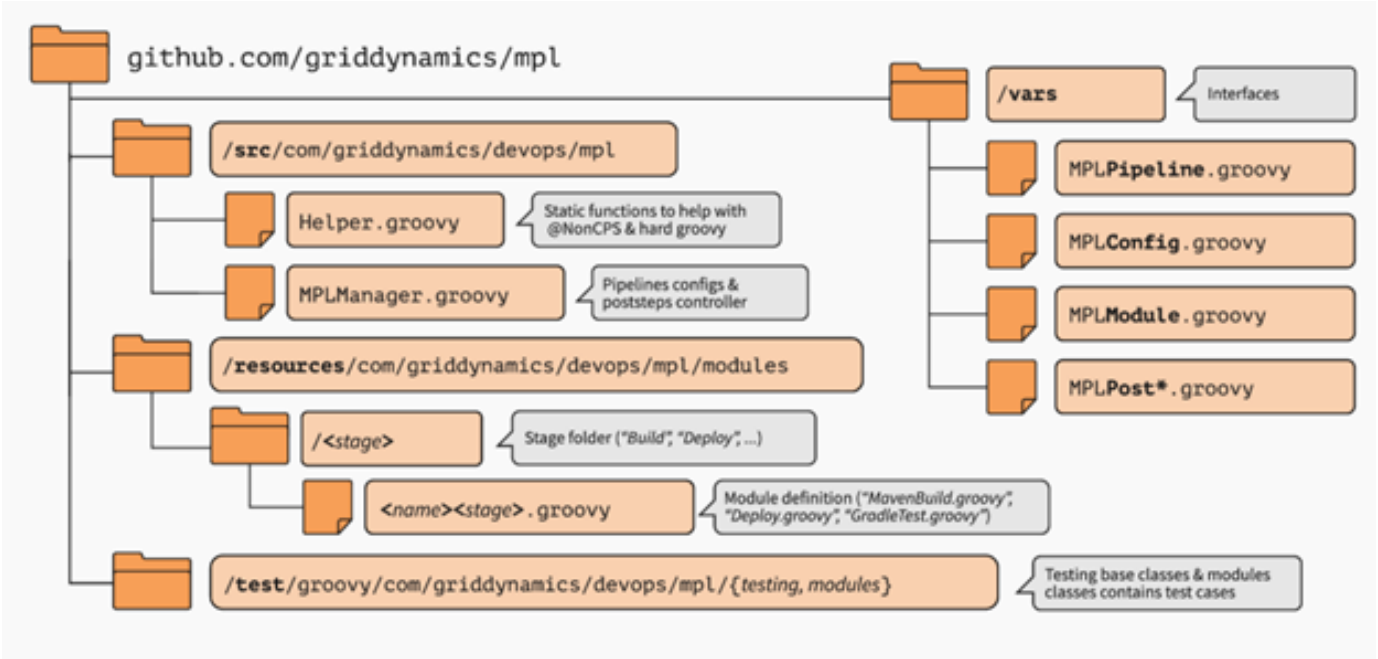
MPL Process:



MPL Modules Override:



A MPL Repository Structure:



- src/, for your classes/methods. Most of your code goes here. A lot of guides on the internet have most, if not all, of theirs in vars/, either because they're using declarative pipelines (which have limitations) or because that's how they got it to work. Inside the src/ directory the structure is [io|org|com|other tld]/companyname/, and then your .groovy files. This convention comes from Java, and it's pretty important to follow.
- vars/ is for, well, global variables. I haven't seen a whole lot of examples of them in practice, though, and it never seems to be a regular var=value type of variable. The most common examples I've seen start with def call.
- resources/ are your non-Groovy files, if you have them. I haven't needed to add any extra files yet, but if you need to, that's where.

MPL Configuration In Jenkins:

- Need to add that repository to Jenkins, under Manage Jenkins » Configure System » Global Pipeline Libraries.
- Choosing to 'Load Implicitly' means you don't have to use the @Library tag in your pipelines to access your library, but also means your library will be loaded in every single pipeline whether you want it or not (you can see the SCM checkout and last commit of your shared library towards the beginning of your pipeline).

- The ‘Allow default version to be overridden’ option is useful — it lets you specify a different branch (or tag, or other identifier) than the defaults so you can try out changes in one place before pushing them on everything, which is important when any change you make to the library has the chance to affect every pipeline that uses it.

MPL Example:

Traditional declarative Jenkinsfile-

```
node {
stage('build') {
//build code (ex: mvn deploy)
}
stage('deploy to qa') {
//deploy code (ex: aws cloudformation create-stack)
}
stage('test') {
//test code (ex: sh /path/to/test/script.sh)
}
stage('deploy to staging') {
//deploy code (ex: aws cloudformation create-stack)
}
stage('test') {
//test code (ex: sh /path/to/test/script.sh)
}
stage('deploy to prod') {
//deploy code (ex: aws cloudformation create/update-stack)
}
```

Groovy code to create MPL:

```
src/io/abc/pipeline.groovy
#!/usr/bin/groovy
package io.abc;

def build() {
  mvn clean deploy -U
}

def test(name) {
  sh "/usr/local/bin/${name}"
}

def deploy(env,app) {
  aws cloudformation create-stack \
  --stack-name ${env}_{app)
  --parameters \
    ParameterKey=Env,ParameterValue=${env}
  }
```

Module Based Jenkins File:

```
Jenkinsfile// import
@Library("jenkins-shared-libraries@release")
import io.abc.*// instantiate
pl = new pipeline()// work
pl.build()
pl.deploy(qa,api)
pl.test(regression)
pl.deploy(staging,api)
pl.test(load)
pl.deploy(prod,api)
```

The annotation can be anywhere in the script where an annotation is permitted by Groovy. When referring to class libraries (with src/ directories), conventionally the annotation goes on an import statement:

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

For Shared Libraries which only define Global Variables (vars/), or a Jenkinsfile which only needs a Global Variable, the annotation pattern @Library('my-shared-library') _ may be useful for keeping code concise. In essence, instead of annotating an unnecessary import statement, the symbol _ is annotated.

It is not recommended to import a global variable/function, since this will force the compiler to interpret fields and methods as static even if they were intended to be instance. The Groovy compiler in this case can produce confusing error messages.

Libraries are resolved and loaded during compilation of the script, before it starts executing. This allows the Groovy compiler to understand the

meaning of symbols used in static type checking, and permits them to be used in type declarations in the script, for example:

```
@Library('somelib')
import com.mycorp.pipeline.somelib.Helper

int useSomeLib(Helper helper) {
helper.prepare()
return helper.count()
}

echo useSomeLib(new Helper('some text'))
```

Library Version:

The "Default version" for a configured Shared Library is used when "Load implicitly" is checked, or if a Pipeline references the library only by name, for example

```
@Library('my-shared-library') _
```

If a "Default version" is not defined, the Pipeline must specify a version, for example -

```
@Library('my-shared-library@master') _
/* Using a version specifier, such as branch, tag, etc */
@Library('my-shared-library@1.0') _
/* Accessing multiple libraries with one statement */
@Library(['my-shared-library', 'otherlib@abc1234']) _
```

If "Allow default version to be overridden" is enabled in the Shared Library's configuration, a @Library annotation may also override a default version defined for the library. This also allows a library with "Load implicitly" to be loaded from a different version if necessary.

When using the library step you may also specify a version:

```
library 'my-shared-library@master'
```

Since this is a regular step, that version could be computed rather than a constant as with the annotation; for example:

```
library "my-shared-library@$BRANCH_NAME"
```

would load a library using the same SCM branch as the multibranch Jenkinsfile. As another example, you could pick a library by parameter:

```
properties([parameters([string(name: 'LIB_VERSION', defaultValue:
'master')])])
library "my-shared-library@${params.LIB_VERSION}"
```

Note that the library step may not be used to override the version of an implicitly loaded library. It is already loaded by the time the script starts, and a library of a given name may not be loaded twice.