Share a standard Pipeline across multiple projects with Shared Libraries

Published on 2017-10-02 by Philip Stroh Tweet

- pipeline
- declarative
- microservices

This is a guest post by Philip Stroh, Software Architect at TimoCom.

When building multiple microservices - e.g. with Spring Boot - the integration and delivery pipelines of your services will most likely be very similar. Surely, you don't want to copy-and-paste Pipeline code from one `Jenkinsfile` to another if you develop a new service or if there are adaptions in your delivery process. Instead you would like to define something like a pipeline "template" that can be applied easily to all of your services.

The requirement for a common pipeline that can be used in multiple projects does not only emerge in microservice architectures. It's valid for all areas where applications are built on a similar technology stack or deployed in a standardized way (e.g. pre-packages as containers).

In this blog post I'd like to outline the possibility to create such a pipeline "template" using Jenkins Shared Libraries. If you're not yet familiar with Shared Libraries I'd recommend having a look at the documentation.

The following code shows a (simplified) integration and delivery Pipeline for a Spring Boot application in declarative syntax.

JenkinsFile

```
pipeline {
    agent any
    environment {
        branch = 'master'
        scmUrl = 'ssh://git@myScmServer.com/repos/myRepo.git'
        serverPort = '8080'
        developmentServer = 'dev-myproject.mycompany.com'
        stagingServer = 'staging-myproject.mycompany.com'
        productionServer = 'production-myproject.mycompany.com'
    }
    stages {
        stage('checkout git') {
            steps {
                git branch: branch, credentialsId: 'GitCredentials', url: scmUrl
            }
        }

        stage('build') {
            steps {
                sh 'mvn clean package -DskipTests=true'
            }
        }

        stage ('test') {
            steps {
                parallel (
                    "unit tests": { sh 'mvn test' },
                    "integration tests": { sh 'mvn integration-test' }
                )
            }
        }

        stage('deploy development'){
            steps {
                deploy(developmentServer, serverPort)
            }
        }

        stage('deploy staging'){
            steps {
                deploy(stagingServer, serverPort)
            }
        }

        stage('deploy production'){
            steps {
                deploy(productionServer, serverPort)
            }
        }
    }
    post {
        failure {
            mail to: 'team@example.com', subject: 'Pipeline failed', body: "${env.BUILD_URL}"
        }
    }
}
```

This Pipeline builds the application, runs unit as well as integration tests and deploys the application to several environments. It uses a global variable "deploy" that is provided within a Shared Library. The deploy method copies the JAR-File to a remote server and starts the application. Through the handy REST endpoints of Spring Boot Actuator a previous version of the application is stopped beforehand. Afterwards the deployment is verified via the health status monitor of the application.

vars/deploy.groovy

```
def call(def server, def port) {
    httpRequest httpMode: 'POST', url: "http://${server}:${port}/shutdown", validResponseCodes: '200,408'
    sshagent(['RemoteCredentials']) {
        sh "scp target/*.jar root@${server}:/opt/jenkins-demo.jar"
        sh "ssh root@${server} nohup java -Dserver.port=${port} -jar /opt/jenkins-demo.jar &"
    }
    retry (3) {
        sleep 5
        httpRequest url:"http://${server}:${port}/health", validResponseCodes: '200', validResponseContent: '"status":"UP"'
    }
}
```

The common approach to reuse pipeline code is to put methods like "deploy" into a Shared Library. If we now start developing the next application of the same fashion we can use this method for deployments as well. But often there are even more similarities within projects of one company. E.g. applications are built, tested and deployed in the same way into the same environments (development, staging and production). In this case it is possible to define the whole Pipeline as a global variable within a Shared Library. The next code snippet defines a Pipeline "template" for all of our Spring Boot applications.

vars/myDeliveryPipeline.groovy

```
def call(Map pipelineParams) {

    pipeline {
        agent any
        stages {
            stage('checkout git') {
                steps {
                    git branch: pipelineParams.branch, credentialsId: 'GitCredentials', url: pipelineParams.scmUrl
                }
            }

            stage('build') {
                steps {
                    sh 'mvn clean package -DskipTests=true'
                }
            }

            stage ('test') {
                steps {
                    parallel (
                        "unit tests": { sh 'mvn test' },
                        "integration tests": { sh 'mvn integration-test' }
                    )
                }
            }

            stage('deploy developmentServer'){
                steps {
                    deploy(pipelineParams.developmentServer, pipelineParams.serverPort)
                }
            }

            stage('deploy staging'){
                steps {
                    deploy(pipelineParams.stagingServer, pipelineParams.serverPort)
                }
            }

            stage('deploy production'){
                steps {
                    deploy(pipelineParams.productionServer, pipelineParams.serverPort)
                }
            }
        }
        post {
            failure {
                mail to: pipelineParams.email, subject: 'Pipeline failed', body: "${env.BUILD_URL}"
            }
        }
    }
}
```

Now we can setup the Pipeline of one of our applications with the following method call:

Jenkinsfile

```
myDeliveryPipeline(branch: 'master', scmUrl: 'ssh://git@myScmServer.com/repos/myRepo.git',
                   email: 'team@example.com', serverPort: '8080',
                   developmentServer: 'dev-myproject.mycompany.com',
                   stagingServer: 'staging-myproject.mycompany.com',
                   productionServer: 'production-myproject.mycompany.com')
```

The Shared library documentation mentions the ability to encapsulate similarities between several Pipelines with a global variable. It shows how we can enhance our template approach and build a higher-level DSL step:

vars/myDeliveryPipeline.groovy

```
def call(body) {
    // evaluate the body block, and collect configuration into the object
    def pipelineParams= [:]
    body.resolveStrategy = Closure.DELEGATE_FIRST
    body.delegate = pipelineParams
    body()

    pipeline {
        // our complete declarative pipeline can go in here
        ...
    }
}
```

Now we can even use our own DSL-step to set up the integration and deployment Pipeline of our project:

Jenkinsfile

```
myDeliveryPipeline {
    branch = 'master'
    scmUrl = 'ssh://git@myScmServer.com/repos/myRepo.git'
    email = 'team@example.com'
    serverPort = '8080'
    developmentServer = 'dev-myproject.mycompany.com'
    stagingServer = 'staging-myproject.mycompany.com'
    productionServer = 'production-myproject.mycompany.com'
}
```

The blog post showed how a common Pipeline template can be developed using the Shared Library functionality in Jenkins. The approach allows to create a standard Pipeline that can be reused by applications that are built in a similar way.

It works for Declarative and Scripted Pipelines as well. For declarative pipelines the ability to define a Pipeline block in a Shared Library is official supported since version 1.2 (see the recent blog post on Declarative Pipeline 1.2).

About the Author

Philip Stroh

This author has no biography defined. See social media links referenced below.

- GitHub