Jenkins Shared Library, Jenkins pipeline'larınızda kullanabileceğiniz ve pipeline'ların tekrar kullanılabilirliğini artıran önceden tanımlanmış işlevleri ve global değişkenleri içeren bir kod deposudur. Bu sayede, birden fazla pipeline içinde aynı işlevleri tekrar tekrar tanımlamak yerine, bu işlevleri tek bir yerde tanımlayabilir ve tüm pipeline'larınızda kullanabilirsiniz.

Maven ve Groovy dili kullanarak Jenkins Shared Library oluşturmak için aşağıdaki adımları takip etmemiz gerekir:

- Jenkins sunucusunuza (Jenkinsin barındırıldığı server) SSH veya RDP protokolü ile erişelim.
- Shared Library oluşturmak için Jenkins ana dizininde bir dizin oluşturalım. Örneğin, "my-shared-library" adında bir dizin oluşturulabilir yada ÖrnekHVLProje-SharedLib örneğin.
- Oluşturduğumuz dizine gidip ve "src" adlı bir alt dizin oluşturalım. Bu dizin içinde Groovy scriptlerini tutacağız.
- "vars" adlı bir alt dizin oluşturalım. Bu dizin içinde pipeline işlevlerini tanımlayacağız.
- vars dizininde, her bir işlev için bir Groovy betiği oluşturun. Örneğin, "myFunction.groovy" adında bir betik oluşturabiliriz. (Bu ideal yaklaşım standart akışlar için toplu bir groovy de kullanılabilir şunu akılda tutalım yapacağımız her işlem için ör: build bir script ve bir de functionımız olacak, groovyde function python ile aynı şekilde tanımlanıyor)
- İşlevleri tanımladıktan sonra, Maven ile bir proje oluşturalım. Proje ayarları için pom.xml adında bir dosya ihtiyacımız olacak bildiğimiz, hatırlayacağımız gibi.
- pom.xml dosyasında, Jenkins Shared Library plugin'ini eklemek önemli. Ayrıca, Shared Library'yi otomatik olarak Jenkins sunucusuna yüklemek için Jenkins sunucusunun URL'sini ve kimlik (token-user) bilgilerini de eklemek gerekebilir.
- Maven projesini derleyin (mvn package, mvn build, mvn install vs.). Derleme işlemi tamamlandığında, target dizininin altında bir JAR dosyası oluşturulacaktır.
- Son olarak, Jenkins sunucusundaki Jenkins arayüzünde "Manage Jenkins" > "Configure System" sayfasına gidelim ve "Global Pipeline Libraries" bölümünde "Add" butonuna tıklayarak oluşturduğunuz shared library'i Jenkins'a ekleyelim. Burada, oluşturduğunuz JAR dosyasının yolunu ve işlevleri yüklemek için kullanılacak bir workspace alanı belirleyebiliriz.

Bu adımları tamamladıktan sonra, pipeline'larımızzda oluşturduğunuz işlevleri kullanabiliriz. Örneğin, aşağıdaki kod, "myFunction" adlı işlevi kullanarak bir örnek pipeline tanımı göstermektedir, bu bir şablon burada build olabilir örneğin myfunction yerine öyle düşünelim.

```
@Library('my-shared-library') _
pipeline {
  agent any
  stages {
    stage('Example') {
      steps {
        myFunction()
      }
    }
  }
}
```

HAVELSAN'da genel örneğin TDLYM için uyguladığımız akışı içeren bir groovy pipeline şablonu yazalım;

```
pipeline {
  agent any
  // Git Checkout
  stages {
    stage('Checkout') {
      steps {
        git branch: 'main', url: 'https://github.com/your-hvlrepo.git'
      }
    }
  }
  // SonarQube Code Analysis
  stages {
    stage('SonarQube Analysis') {
      steps {
        withSonarQubeEnv('My SonarQube Server') {
          sh 'mvn sonar:sonar'
        }
      }
    }
  }
  // Build and Package
  stages {
    stage('Build and Package') {
      steps {
        sh 'mvn clean package'
      }
    }
  }
  // Deploy
  stages {
    stage('Deploy') {
      steps {
        sh 'mvn deploy'
      }
    }
  }
  // Git Tag
  stages {
    stage('Tag Git Repo') {
      steps {
        sh 'git tag -a v1.0 -m "Version 1.0"'
        sh 'git push origin v1.0'
      }
    }
  }
}
```

Yukarıdaki Groovy script'i, aşağıdaki adımları gerçekleştirir:

git komutunu kullanarak, Jenkins sunucusu tarafından yürütülen pipeline için belirtilen GitHub deposunu main branch üzerinde kontrol eder.

withSonarQubeEnv komutu ile, SonarQube sunucusuna kod analizi için gerekli olan çevre değişkenleri tanımlanır ve mvn sonar:sonar komutu ile SonarQube'da kod analizi gerçekleştirilir. Burada sonar environment ve proje bilgisi tanımlanmalıdır.

mvn clean package komutu ile Maven projesi, bir JAR dosyasına paketlenir.

mvn deploy komutu ile paketlenmiş JAR dosyası, depolama hizmetine dağıtılır. (Nexus)

git tag ve git push komutları ile, projenin son sürümü için bir etiket oluşturulur ve GitHub deposuna yüklenir.

Bunu Shared Library yapısına çevirmemiz gerektiğinde şunları göz önünde bulunduralım;

my-shared-library adında bir shared library oluşturarak, yukarıdaki Jenkins pipeline'i yeniden yazalım:

```groovy
// vars/myJenkinsPipeline.groovy

def call() {
  pipeline {
    agent any

    stages {
      stage('Checkout') {
        steps {
          checkoutGit()
        }
      }

      stage('SonarQube Analysis') {
        steps {
          sonarqubeAnalysis()
        }
      }

      stage('Build and Package') {
        steps {
          buildAndPackage()
        }
      }

      stage('Deploy') {
        steps {
          deploy()
        }
      }

      stage('Tag Git Repo') {
        steps {
          tagGitRepo()
```

```
        }
      }
    }
  }
}

def checkoutGit() {
  git branch: 'main', url: 'https://github.com/your-hvlrepo.git'
}

def sonarqubeAnalysis() {
  withSonarQubeEnv('My SonarQube Server') {
    sh 'mvn sonar:sonar'
  }
}

def buildAndPackage() {
  sh 'mvn clean package'
}

def deploy() {
  sh 'mvn deploy'
}

def tagGitRepo() {
  sh 'git tag -a v1.0 -m "Version 1.0"'
  sh 'git push origin v1.0'
}
```

Bu Groovy script'i, **myJenkinsPipeline** adında bir fonksiyon olarak tanımladık. Bu fonksiyon, pipeline'i yürütmek için gerekli adımları içeriyor. Bu fonksiyon ayrıca içerisinde yapacağımız işlemlerin tanımlandığı deploy, tagGitRepo gibi fonksiyonları da içeriyor.

Hadi şimdi bu shared library'i kullanmak için, Jenkinsfile'ımızda aşağıdaki gibi bir kod parçasını ekleyelim:

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

myJenkinsPipeline()
```

Yukarıdaki kod parçası, **my-shared-library** adındaki shared library'yi yükler ve **myJenkinsPipeline** adlı fonksiyonu çağırır. Bu fonksiyon, pipeline'i yürütmek için gereken adımları çalıştırır.

Hadi şimdi daha doğru bir yapı kuralım ve her fonksiyonu adımı ayrı groovy scriptler olarak tanımlayalım, ve bu scriptler çağrıldığı projede her adımda kullanılabilir olsun;

Bunun için;

Her adımı ayrı ayrı Groovy scriptler olarak tanımlayacağımız bir shared library yapısı için, öncelikle her adım için bir **vars** dizini içinde bir Groovy script oluşturmamız gerekiyor. Bu script'ler, adımları gerçekleştiren fonksiyonları içerecek ve Jenkinsfile'ımızda bu fonksiyonları çağıracağız.

Yukarıdaki yapıyı ele alırsak;

checkoutGit.groovy, **sonarqubeAnalysis.groovy**, **buildAndPackage.groovy**, **deploy.groovy** ve **tagGitRepo.groovy** adında beş farklı Groovy scriptimiz olacak.

1. **checkoutGit.groovy** dosyası:

```
// vars/checkoutGit.groovy

def call(String gitUrl, String gitBranch) {
  git branch: gitBranch, url: gitUrl
}
```

2. **sonarqubeAnalysis.groovy** dosyası:

```
// vars/sonarqubeAnalysis.groovy

def call(String sonarqubeServerName) {
  withSonarQubeEnv(sonarqubeServerName) {
    sh 'mvn sonar:sonar'
  }
}
```

3. **buildAndPackage.groovy** dosyası:

```
// vars/buildAndPackage.groovy

def call() {
  sh 'mvn clean package'
}
```

4. **deploy.groovy** dosyası:

```
// vars/deploy.groovy

def call() {
  sh 'mvn deploy'
}
```

5. **tagGitRepo.groovy** dosyası:

```
// vars/tagGitRepo.groovy

def call(String tagName) {
  sh "git tag -a ${tagName} -m 'Version ${tagName}'"
  sh "git push origin ${tagName}"
}
```

Bu script'leri oluşturduktan sonra, bunları kullanmak için bir Jenkinsfile'da çağırmamız gerekiyor. Aşağıdaki kod parçasını bunun için kullanacağız;

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

pipeline {
  agent any

  stages {
    stage('Checkout') {
      steps {
        checkoutGit(gitUrl: 'https://github.com/your-hvlrepo.git', gitBranch:
'main')
      }
    }

    stage('SonarQube Analysis') {
      steps {
        sonarqubeAnalysis(sonarqubeServerName: 'My SonarQube Server')
      }
    }
    stage('Build and Package') {
      steps {
        buildAndPackage()
      }
    }
    stage('Deploy') {
      steps {
        deploy()
      }
    }
    stage('Tag Git Repo') {
      steps {
        tagGitRepo(tagName: 'v1.0')
      }
    }
  }
}
```

Buradaki dosya yapımız, filetree olarak görürsek de şöyle olmalıdır:

/var/lib/jenkins/my-shared-library

└── vars

  ├── buildAndPackage.groovy

  ├── checkoutGit.groovy

  ├── deploy.groovy

  ├── sonarqubeAnalysis.groovy

  └── tagGitRepo.groovy

Shared library, Jenkins sunucusunda herhangi bir dizinde olabilir, ancak genellikle Jenkins'ın çalıştığı ana dizindeki **vars** veya **src** dizinleri altında yer alır, biz de bunu kullanacağız.

Örneğin, Jenkins sunucusunda **/var/lib/jenkins** dizininde yer alıyorsanız, **my-shared-library** adlı shared library'yi yukarıdaki gibi bir dizin yapısında tutmalıyız.

Ayrıca Bu Jenkinsfile, shared library'yi kullanarak pipeline'ın her aşamasını ayrı ayrı çağırır. **@Library** yönergesi, **my-shared-library** adlı shared library'yi çağırmak için kullanılır. **agent any** ifadesi, herhangi bir Jenkins ajanı üzerinde çalışabilecek bir pipeline oluşturur yada burada spesifik belirtebiliriz. Pipeline'ın her aşaması, shared library'deki ilgili Groovy script'ini çağırmak için **steps** bloğu kullanır.

Şimdi hadi bu projeyi uçtan uca nasıl gerçekleştireceğimizi anlatalım ve adım adım resmi toparlayalım;

1. Jenkins sunucusu üzerinde shared library dizini oluşturuyoruz.

Jenkins sunucusu üzerinde, shared library'yi barındırmak için bir dizin oluşturmak gerekiyor. Bu dizinde, tüm Groovy script'lerini ve diğer dosyaları saklayacağız.

Örneğin, **/var/lib/jenkins/shared-library** dizinini oluşturabilirsiniz.

2. shared library içinde vars ve src dizinlerini oluşturuyoruz.

vars ve src dizinlerini shared library dizininin altında oluşturacağız. Bu dizinlerin altında, pipeline'ımızda çağrılacak Groovy script'lerini tutacağız. Şöyle ki;

/var/lib/jenkins/shared-library

├── vars

└── src

3. Groovy script'leri shared library'nin **vars** dizinine yerleştirelim.

Pipeline'ımızda kullanacağımız her adım için bir Groovy script oluşturuyoruz. Bu script'leri, **vars** dizininin altına kaydedeceğiz.

Örneğin, **checkoutGit.groovy**, **sonarqubeAnalysis.groovy**, **buildAndPackage.groovy**, **deploy.groovy** ve **tagGitRepo.groovy** adlı beş script oluşturalım bizim proje yapımızda ve tüm script'leri **vars** dizininin altına yerleştirelim.

Yapıyı şöyle gösterelim;

/var/lib/jenkins/shared-library

└── vars

  ├── buildAndPackage.groovy

  ├── checkoutGit.groovy

  ├── deploy.groovy

  ├── sonarqubeAnalysis.groovy

  └── tagGitRepo.groovy

4. Diğer dosyaları shared library'nin **src** dizinine yerleştirmemiz gerekir.

Shared library'nin diğer dosyalarını, örneğin util script'lerini, **src** dizininin altında saklayabiliriz util script'ler, birden çok pipeline'ınızda kullanılabilen ve farklı adımlar için genel bir çözüm sağlayabilen genel amaçlı fonksiyonlar olabilir. Örneğin, bir util script, verilen bir dosya yolundaki dosyanın var olup olmadığını kontrol edebilir veya bir HTTP isteği göndermek için genel bir fonksiyon olabilir. Yada pipeline için bir test.groovy dosyamız varsa onu burada saklayabiliriz. Burada yapı şu hale evrilecektir:

/var/lib/jenkins/shared-library

├── src

│ ├── Util.groovy

│ └── ...

└── vars

  ├── buildAndPackage.groovy

  ├── checkoutGit.groovy

  ├── deploy.groovy

  ├── sonarqubeAnalysis.groovy

  └── tagGitRepo.groovy

5. Şimdi en güzel yerine geldik. Jenkinsfile oluşturalım ve Git reposuna ekleyelim.

Pipeline'ınızı tanımlamak için bir Jenkinsfile oluşturmalıyız ve bu Jenkinsfile, Git deposunun ana(kök) dizininde yer almalıdır. Alt bir foldera koymamaya dikkat etmeliyiz.

Örneğin, **/var/lib/jenkins/my-project/Jenkinsfile** adlı bir dosya oluşturun ve pipeline'ımızı tanımlayan Groovy script'i içine yerleştirelim, burada yukarıdaki `'my-shared-library'` isimli scripti kullanabiliriz.

6. Sonra gidelim Jenkinsfile'ı Jenkins üzerinde tanımlayalım.

   Jenkins sunucusunda, pipeline'ımızı oluşturmak için Jenkinsfile'ı tanımlamamız gerekir. Bunun için Jenkins'ın `Manage Jenkins > Configure System` sayfasına gidelim ve "Global Pipeline Libraries" altında "Add" butonuna tıklayalım. Açılan pencerede aşağıdaki ayarları yapıyoruz:

- Name: my-shared-library
- Default version: master
- Retrieval method: Modern SCM
- Select the SCM: Git
- Project Repository: <Git repository URL>
- Credentials: <jenkins credentials for Git repository

BONUS: İlgili versiyona giren Jira issuelarını Pipeline içerisinde kapatacak bir fonksiyon yazalım SharedLibrary düzeninde;

Bunu yapmak için öncelikle **JiraHelper.groovy** dosyamızda **closeIssues** adında bir method oluşturmamız gerekiyor. Bu method, Jira API'si kullanarak git tag'ı altında resolved durumunda olan tüm issueları kapatacak.

Kod içeriği GitHub/fbildirici/SharedLibraryCourseFB reposunda vars dizini altında.

Bu dosyayı oluşturduktan sonra, Jenkinsfile içinde ilgili adımları ekleyebiliriz. Örneğin, **stage('Close Jira Issues')** adımları arasına aşağıdaki kod bloğunu ekleyebiliriz. Jira auth variableları da jenkinse embed etmediysek environment variable olarak vermemiz gerekir.

Kod içeriği GitHub/fbildirici/SharedLibraryCourseFB reposunda ana dizindeki jenkinsfile.

Kodlar için anahtar:

Burada **myJiraHelper** adında bir nesne kullanılmaktadır. Bu nesne, **my-shared-library** adlı paylaşılan kütüphane aracılığıyla sağlanmaktadır. Bu kütüphane, Jira ile etkileşime geçmek için **JiraHelper.groovy** dosyasını içerir.

Pipeline, **JiraHelper.groovy** dosyasını yüklemek ve nesneyi oluşturmak için **@Library** deyimi kullanır. **JiraHelper.groovy** dosyasında, **getResolvedIssues()** ve **closeIssue(issueKey)** gibi fonksiyonlar Jira ile etkileşim kurmak için kullanılır.

Jenkinsfile dosyasında, **JIRA_USERNAME** ve **JIRA_PASSWORD** gibi gizli bilgiler de tanımlanır. Bu bilgiler, Jenkins'teki gizli bilgi yönecisi (secret) aracılığıyla saklanır ve environment değişkenleri olarak Jenkinsfile'da kullanılır.

Burada kullandığımız **myJiraHelper** bir Jenkins Shared Library'sidir ve içerisinde Jira REST API'larını kullanarak Jira ile etkileşim kurmak için gerekli yardımcı fonksiyonları içeren bir Groovy classıdır. **myJiraHelper** classı, Jira ile etkileşim kurmak için REST API çağrıları yapar ve bu çağrıların sonucunda elde edilen verileri işleyerek, örneğin Jira'dan hedeflenen issues'ların verilerini çekerek bizim örneğimizde ve sonra yukarıdaki fonksiyonlarda görüldüğü üzere kapatarak bu işlemi yapar.