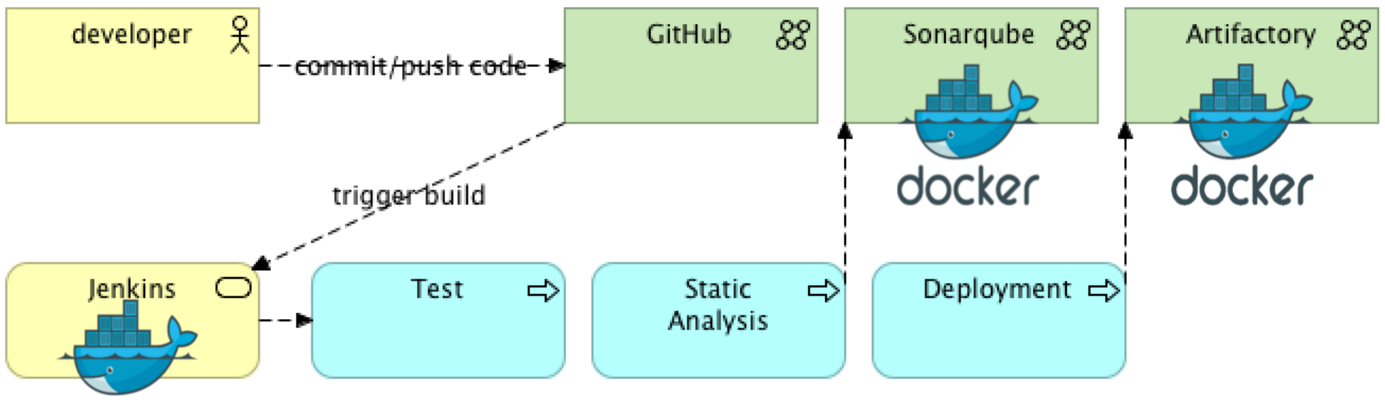
DZone

Dockerizing Jenkins, Part 2: Deployment With Maven and JFrog Artifactory

by Kayan Azimov · Jul. 31, 17 · DevOps Zone

Download the blueprint that can take a company of any maturity level all the way up to enterprise-scale continuous delivery using a combination of Automic Release Automation, Automic's 20+ years of business automation experience, and the proven tools and practices the company is already leveraging.

In the first part of this tutorial, we looked at how to dockerize installation of the Jenkins plugins, Java and Maven tool setup in Jenkins 2, and created a declarative build pipeline for a Maven project with test and SonarQube stages. In this part, we will focus on deployment.



Couldn't we simply add another stage for deployment in part 1, you may ask? Well, in fact, deployment requires quite a few steps to be taken, including Maven pom and settings file configuration, artifact repository availability, repository credentials encryption, etc. Let's add them to the list and then implement them step by step like we did in the previous session.

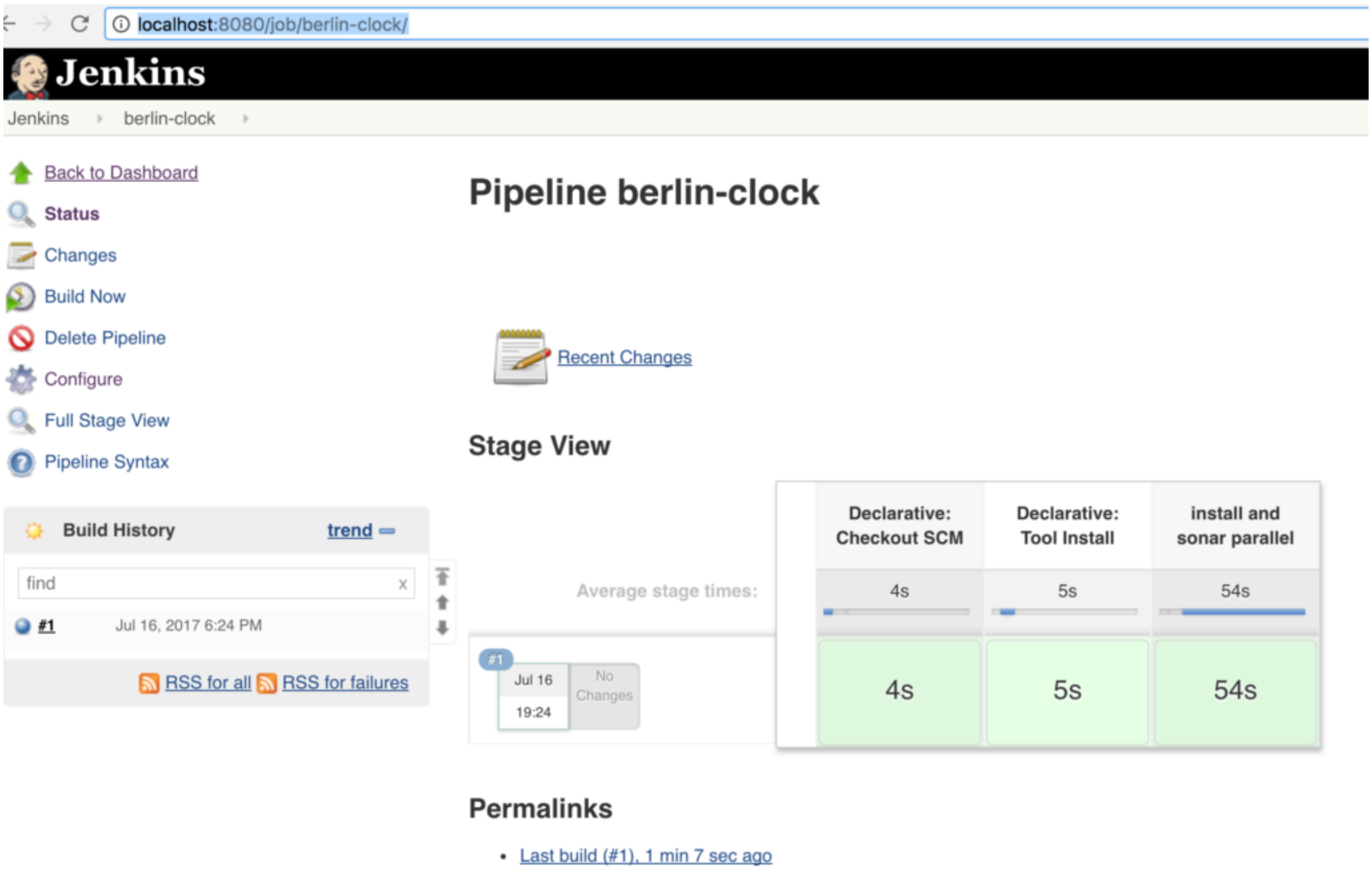
- Running JFrog Artifactory on Docker.
- Configuring the Maven pom file.
- Configuring the Maven settings file.
- Using Config File Provider Plugin for persistence of Maven settings.
- Dockerizing the installation and configuration process.

If you are already familiar with first part of this tutorial, created your project from the scratch and using your own repository, then you can just follow the steps as we go further, otherwise, if you are starting now, you can just clone/fork the work we did in the last example and then add changes as they follow in the tutorial:

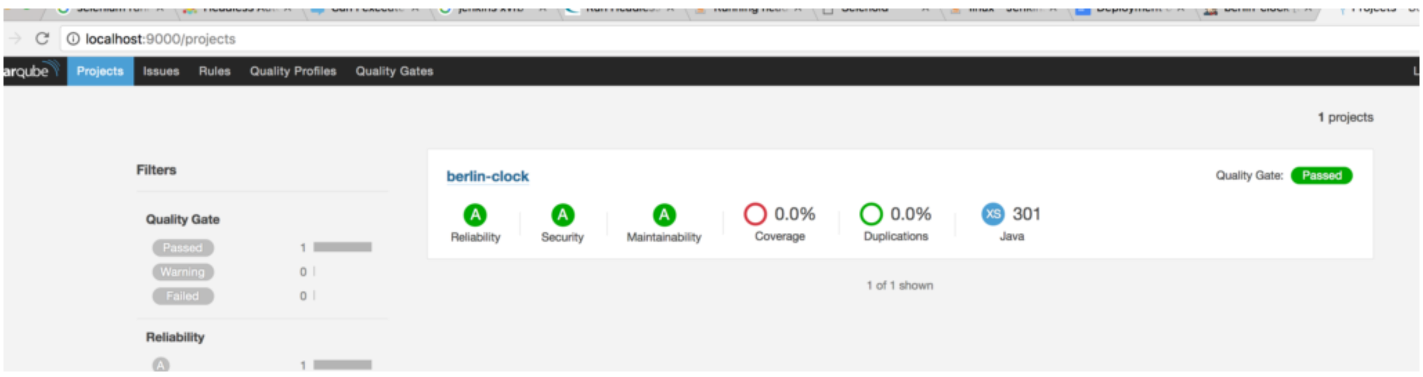
```
git clone https://github.com/kenych/jenkins_docker_pipeline_tutorial1 && cd jenkins_docker
```

Please note all steps have been tested on MacOS Sierra and Docker version 17.05.0-ce and you should change them accordingly if you are using MS-DOS, FreeBSD, etc.

The script above is going to take a while as it is downloading Java 7, Java 8, Maven, SonarQube, and Jenkins Docker images, so please be patient. Once done, you should have Jenkins and Sonar up and running as we created in part 1:



	Declarative: Checkout SCM	Declarative: Tool Install	install and sonar parallel
Average stage times:	4s	5s	54s
#1 Jul 16 19:24 No Changes	4s	5s	54s



If you got errors about some port being busy, just use the free ports from your host, as I explain here. Otherwise, you can use dynamic ports.

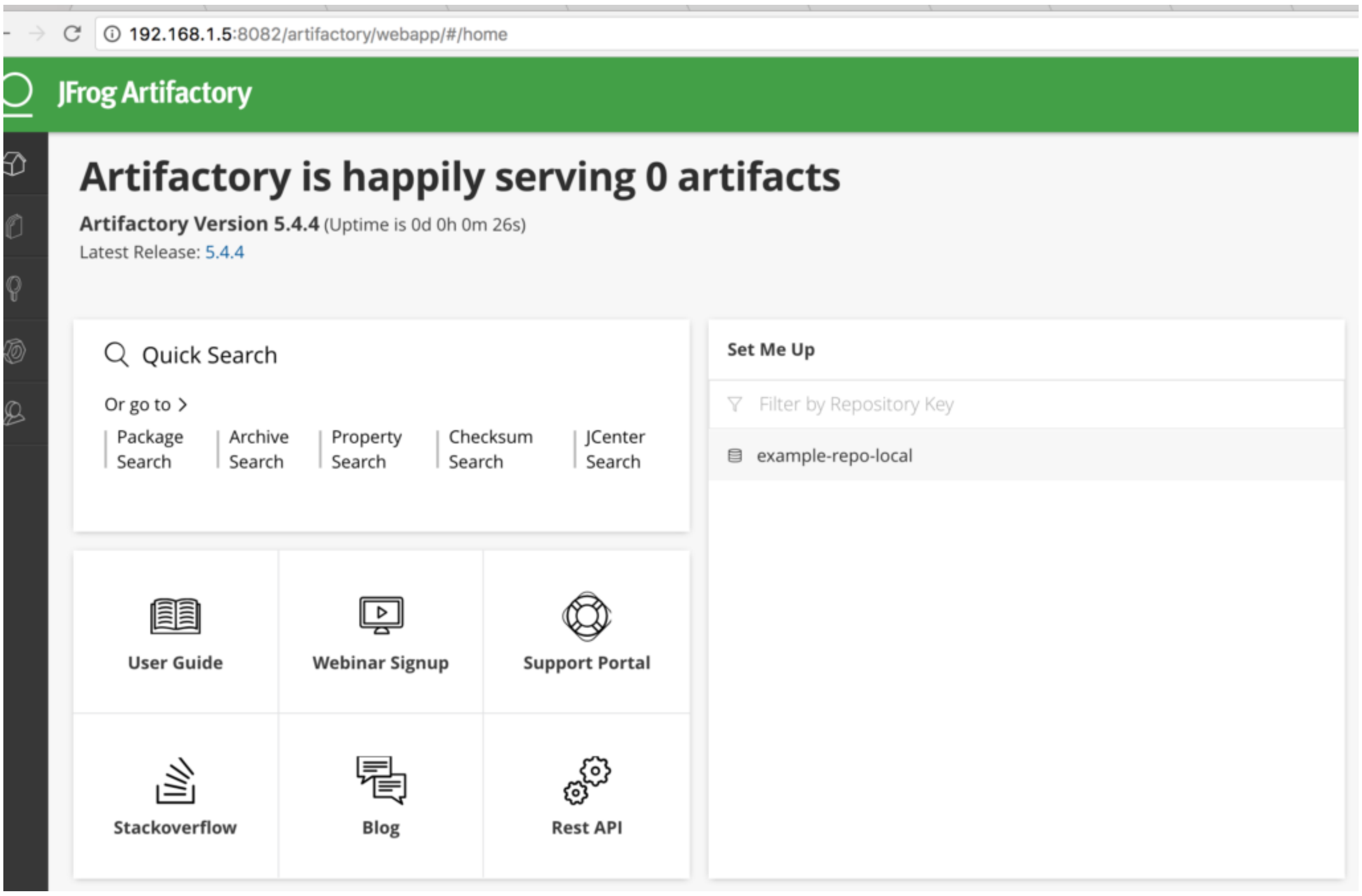
Chapter 1: Running JFrog Artifactory on Docker

So let's look at the first step. Obviously, if we want to test the deployment in our example, we need some place to deploy our artifacts to. We are going to use a limited open source version of JFrog

Artifactory called “artifactory oss.” Let’s run it on Docker to see how easy it is to have your own artifact repo. The port 8081 on my machine was busy, so I had to run it on 8082. You should do so according to free ports available on your machine:

```
docker run --rm -p 8082:8081 --name artifactory docker.bintray.io/jfrog/artifactory-oss:5.
```

Normally you would run Artifactory with volumes to preserve its state (the jar files in our case), but for the sake of our tutorial which is about Dockerizing Jenkins, we will simply run it on “in memory” fashion. Once it is up and running, we should be ready to deploy our artifacts to it.



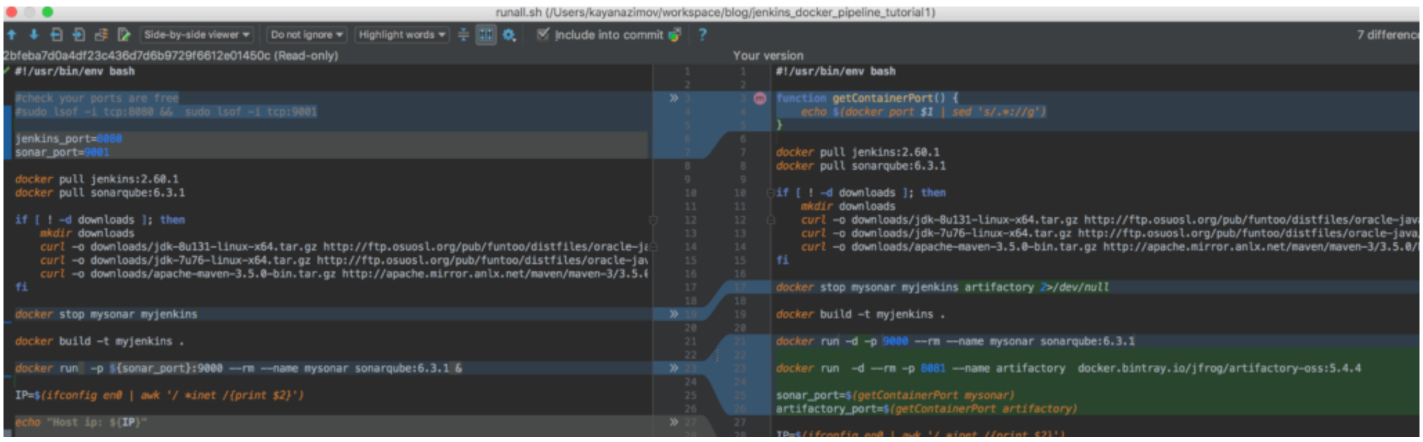
Artifactory is happily serving 0 artifacts

Artifactory Version 5.4.4 (Uptime is 0d 0h 0m 26s)
Latest Release: 5.4.4

example-repo-local

Please pay attention to the name of the path for the default repository it has created; it was example-repo-local in my case, and we will refer to it very soon.

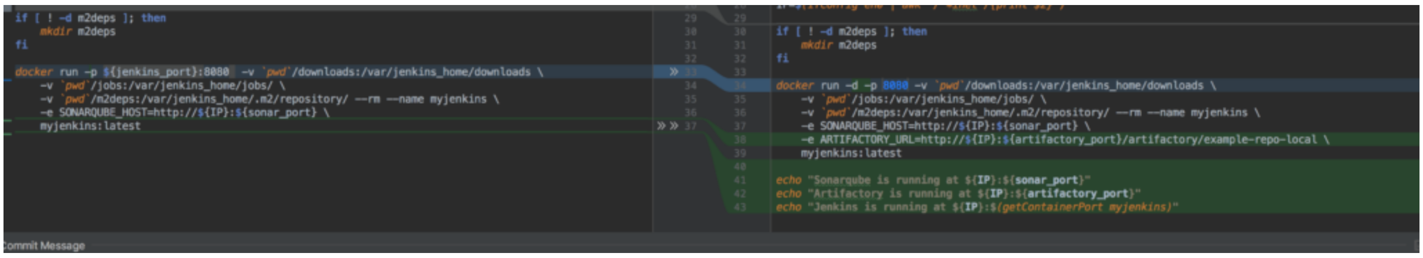
By the time I finished part two, I improved the runall script; now it uses dynamic ports and should run with no issues if some of the ports used by SonarQube, Jenkins or Artifactory were busy (8080, 8081, 9000, etc):



2bfeba7d0a4df23c436d7d6b9729f6612e01450c (Read-only)

```
#!/usr/bin/env bash

#check your ports are free
#sudo lsof -i tcp:8080 &&  sudo lsof -i tcp:9001

jenkins_port=8080
sonar_port=9001

docker pull jenkins:2.60.1
docker pull sonarqube:6.3.1

if [ ! -d downloads ]; then
    mkdir downloads
    curl -o downloads/jdk-8u131-linux-x64.tar.gz http://ftp.osuosl.org/pub/funtoo/distfiles/oracle-ja
    curl -o downloads/jdk-7u76-linux-x64.tar.gz http://ftp.osuosl.org/pub/funtoo/distfiles/oracle-jav
    curl -o downloads/apache-maven-3.5.0-bin.tar.gz http://apache.mirror.anlx.net/maven/maven-3/3.5.
fi

docker stop mysonar myjenkins

docker build -t myjenkins .

docker run  -p ${sonar_port}:9000 --rm --name mysonar sonarqube:6.3.1 &

IP=$(ifconfig en0 | awk '/ *inet /{print $2}')

echo "Host ip: ${IP}"
```

```
#!/usr/bin/env bash

function getContainerPort() {
    echo $(docker port $1 | sed 's/.*://g')
}

docker pull jenkins:2.60.1
docker pull sonarqube:6.3.1

if [ ! -d downloads ]; then
    mkdir downloads
    curl -o downloads/jdk-8u131-linux-x64.tar.gz http://ftp.osuosl.org/pub/funtoo/distfiles/oracle-java
    curl -o downloads/jdk-7u76-linux-x64.tar.gz http://ftp.osuosl.org/pub/funtoo/distfiles/oracle-java
    curl -o downloads/apache-maven-3.5.0-bin.tar.gz http://apache.mirror.anlx.net/maven/maven-3/3.5.0/
fi

docker stop mysonar myjenkins artifactory 2>/dev/null

docker build -t myjenkins .

docker run -d -p 9000 --rm --name mysonar sonarqube:6.3.1

docker run  -d --rm -p 8081 --name artifactory  docker.bintray.io/jfrog/artifactory-oss:5.4.4

sonar_port=$(getContainerPort mysonar)
artifactory_port=$(getContainerPort artifactory)

IP=$(ifconfig en0 | awk '/ *inet /{print $2}')
```



```
if [ ! -d m2deps ]; then
    mkdir m2deps
fi

docker run -p ${jenkins_port}:8080 -v `pwd`/downloads:/var/jenkins_home/downloads \
    -v `pwd`/jobs:/var/jenkins_home/jobs/ \
    -v `pwd`/m2deps:/var/jenkins_home/.m2/repository/ --rm --name myjenkins \
    -e SONARQUBE_HOST=http://${IP}:${sonar_port} \
    myjenkins:latest
```

```
if [ ! -d m2deps ]; then
    mkdir m2deps
fi

docker run -d -p 8080 -v `pwd`/downloads:/var/jenkins_home/downloads \
    -v `pwd`/jobs:/var/jenkins_home/jobs/ \
    -v `pwd`/m2deps:/var/jenkins_home/.m2/repository/ --rm --name myjenkins \
    -e SONARQUBE_HOST=http://${IP}:${sonar_port} \
    -e ARTIFACTORY_URL=http://${IP}:${artifactory_port}/artifactory/example-repo-local \
    myjenkins:latest

echo "Sonarqube is running at ${IP}:${sonar_port}"
echo "Artifactory is running at ${IP}:${artifactory_port}"
echo "Jenkins is running at ${IP}:$(getContainerPort myjenkins)"
```

Commit Message

So let’s switch to updated script. It will just stop all containers, build a Jenkins image, and then rerun with dynamic ports:

```
#!/usr/bin/env bash

function getContainerPort() {
    echo $(docker port $1 | sed 's/.*://g')
}

docker pull jenkins:2.60.1
docker pull sonarqube:6.3.1

if [ ! -d downloads ]; then
    mkdir downloads
    curl -o downloads/jdk-8u131-linux-x64.tar.gz http://ftp.osuosl.org/pub/funtoo/distfile
    curl -o downloads/jdk-7u76-linux-x64.tar.gz http://ftp.osuosl.org/pub/funtoo/distfiles
    curl -o downloads/apache-maven-3.5.0-bin.tar.gz http://apache.mirror.anlx.net/maven/ma
fi

docker stop mysonar myjenkins artifactory 2>/dev/null

docker build -t myjenkins .

docker run -d -p 9000 --rm --name mysonar sonarqube:6.3.1

docker run  -d --rm -p 8081 --name artifactory  docker.bintray.io/jfrog/artifactory-oss:5.

sonar_port=$(getContainerPort mysonar)
artifactory_port=$(getContainerPort artifactory)

IP=$(ifconfig en0 | awk '/ *inet /{print $2}')

if [ ! -d m2deps ]; then
    mkdir m2deps
fi

docker run -d -p 8080 -v `pwd`/downloads:/var/jenkins_home/downloads \
    -v `pwd`/jobs:/var/jenkins_home/jobs/ \
    -v `pwd`/m2deps:/var/jenkins_home/.m2/repository/ --rm --name myjenkins \
    -e SONARQUBE_HOST=http://${IP}:${sonar_port} \
    -e ARTIFACTORY_URL=http://${IP}:${artifactory_port}/artifactory/example-repo-local \
    myjenkins:latest

echo "Sonarqube is running at ${IP}:${sonar_port}"
echo "Artifactory is running at ${IP}:${artifactory_port}"
echo "Jenkins is running at ${IP}:$(getContainerPort myjenkins)"
```

Run it in debug mode to see what is happening:

```
bash +x runall.sh
2.60.1: Pulling from library/jenkins
Digest: sha256:fa62fcebeab220e7545d1791e6eea6759b4c3bdba246dd839289f2b28b653e72
Status: Image is up to date for jenkins:2.60.1
6.3.1: Pulling from library/sonarqube
Digest: sha256:d5f7bb8aecaa46da054bf28d111e5a27f1378188b427db64cc9fb392e1a8d80a
Status: Image is up to date for sonarqube:6.3.1
mysonar
myjenkins
artifactory
Sending build context to Docker daemon  365.1MB
Step 1/6 : FROM jenkins:2.60.1
 ---&amp;amp;gt; f426a52bafa9
Step 2/6 : MAINTAINER Kayan Azimov
 ---&amp;amp;gt; Using cache
 ---&amp;amp;gt; 760e7bb0f335
Step 3/6 : ENV JAVA_OPTS "-Djenkins.install.runSetupWizard=false"
 ---&amp;amp;gt; Using cache
 ---&amp;amp;gt; e3dbac0834cd
Step 4/6 : COPY plugins.txt /usr/share/jenkins/ref/plugins.txt
 ---&amp;amp;gt; Using cache
 ---&amp;amp;gt; 39966bece010
Step 5/6 : RUN /usr/local/bin/install-plugins.sh &amp;amp;lt; /usr/share/jenkins/ref/plugi
 ---&amp;amp;gt; 987bdeca2517
Step 6/6 : COPY groovy/* /usr/share/jenkins/ref/init.groovy.d/
 ---&amp;amp;gt; Using cache
 ---&amp;amp;gt; e5ec6b7f49aa
Successfully built e5ec6b7f49aa
Successfully tagged myjenkins:latest
85d8716d3c7fd6272b6915d977daa37dbe9e4ece0f5c367dd9798fbfca272b2d
5fc3f649d3ba5c47df48a54b761d432611ed18872aa727114cdcf1a0f30cac0c
21e98466cb4f6e78817d9869d39bda57f475d7174f565031f168eb836118d701
Sonarqube is running at 192.168.1.2:32836
Artifactory is running at 192.168.1.2:32837
Jenkins is running at 192.168.1.2:32838
```

We now see the ports for all running containers in the logs, so we can access any of them if we like. As you may have noticed, our shell script has become a bit too complicated; this is just to run three containers, so it means that next time, I should probably switch to using docker compose instead!

We obviously need to update our pipeline which we created in part 1, add a deployment step to it, push the updated Jenkins file, and build the job. If you are using your own repo, alternatively, just amend the existing job as below by using Replthe ay button on the last successful build and then run it:

```
pipeline {
    agent any

    tools {
        jdk 'jdk8'
        maven 'maven3'
    }

    stages {
        stage('install and sonar parallel') {
            steps {
                parallel(
                        install: {
                            sh "mvn -U clean test cobertura:cobertura -Dcobertura.report.f
                        },
                        sonar: {
                            sh "mvn sonar:sonar -Dsonar.host.url=${env.SONARQUBE_HOST}"
                        }
                )
            }
            post {
                always {
                    junit '**/target/*-reports/TEST-*.xml'
                    step([$class: 'CoberturaPublisher', coberturaReportFile: 'target/site/
                }
            }
        }
        stage('deploy') {
            steps {
                    sh "mvn deploy -DskipTests"
            }
        }
    }
}
```

This is what going to happen:

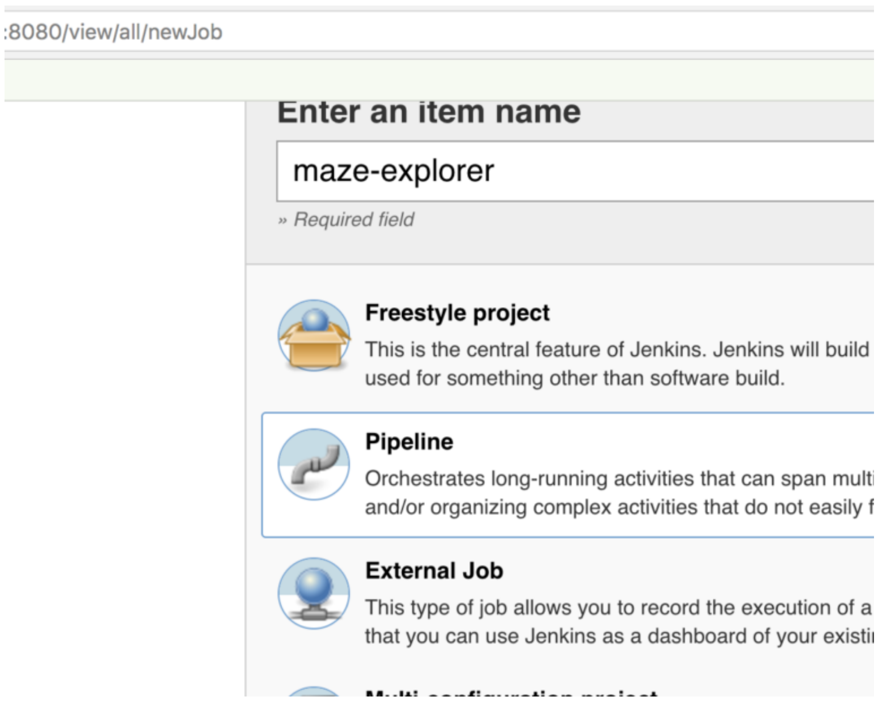
```
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 9.265 s
[INFO] Finished at: 2017-07-16T18:39:51Z
[INFO] Final Memory: 17M/95M
[INFO] ------------------------------------------------------------------------
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-deploy-plugin:2.7:deploy (de
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the foll
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoExecutionException
[Pipeline] }
```

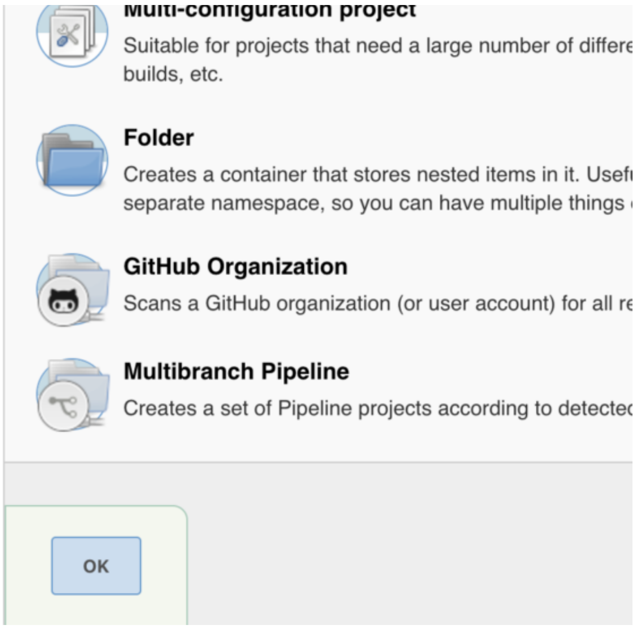
The job fails, but Maven is clever enough to figure out the reason, and it prompted us with a message that the distributionManagement section in the pom file is missing. So the Maven deploy plugin doesn't basically know where to deploy the artifact, would you know?

Chapter 2: Configuring Maven pom File

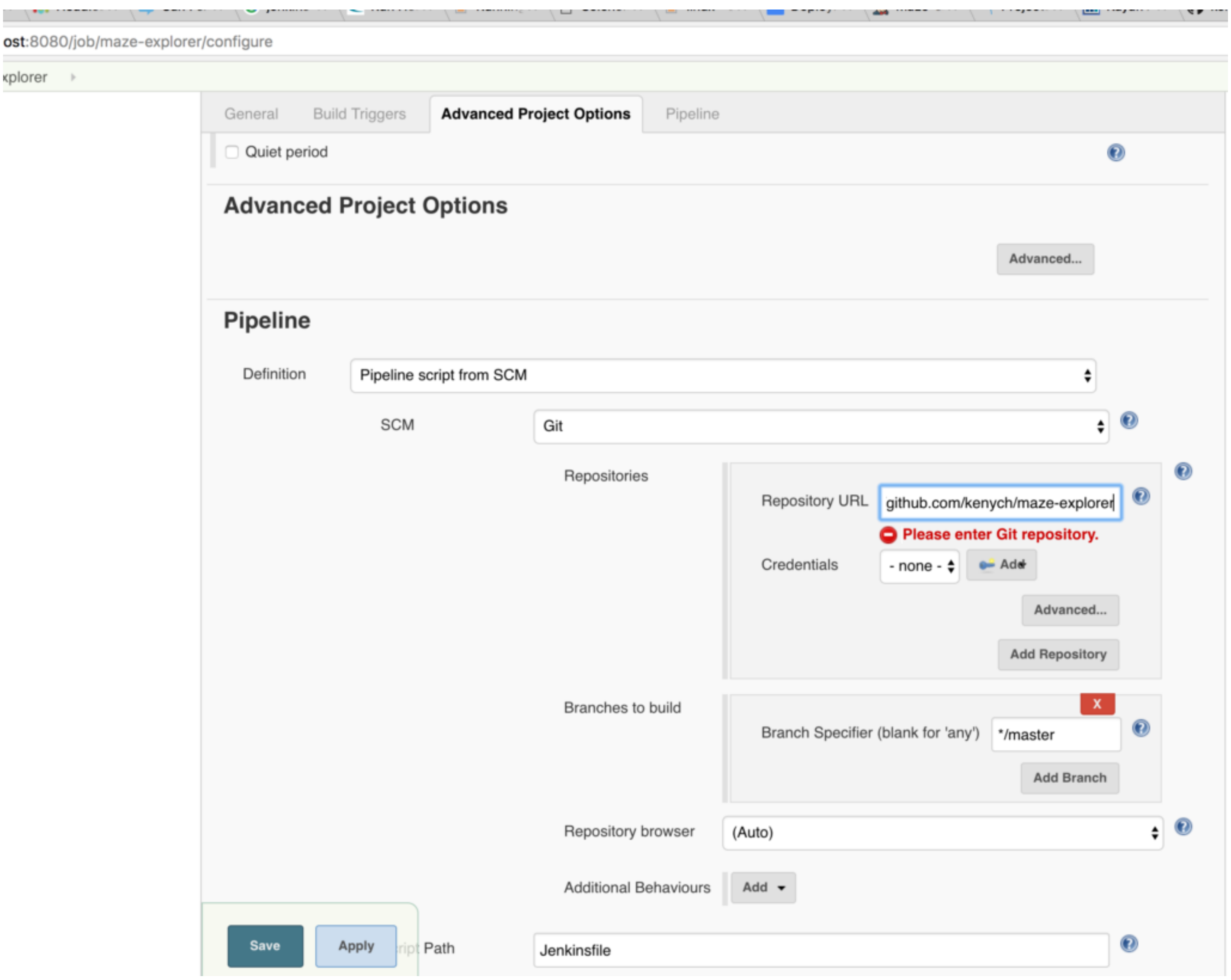
Let's configure the missing part in the pom file inside the project. If you are not using your own git project for this tutorial, you won't be able to amend the POM file, as it is in my repository and you don't have access, unfortunately. But you can switch to another project in that case, which I prepared for you. So please create another pipeline job for the project maze-explorer, which has the necessary pom changes. If you are using your own Git project, just ignore this note. Obviously, I could have used just a branch in the same project, but let's have a couple jobs in Jenkins!

Creating a new pipeline:


:8080/view/all/newJob
Enter an item name
maze-explorer
» Required field
Freestyle project
This is the central feature of Jenkins. Jenkins will build
used for something other than software build.
Pipeline
Orchestrates long-running activities that can span multi
and/or organizing complex activities that do not easily f
External Job
This type of job allows you to record the execution of a
that you can use Jenkins as a dashboard of your existi


Multi-configuration project
Suitable for projects that need a large number of differe
builds, etc.
Folder
Creates a container that stores nested items in it. Usefu
separate namespace, so you can have multiple things
GitHub Organization
Scans a GitHub organization (or user account) for all re
Multibranch Pipeline
Creates a set of Pipeline projects according to detected
OK

Configuring the repository:


ost:8080/job/maze-explorer/configure

General | Build Triggers | Advanced Project Options | Pipeline

Quiet period

Advanced Project Options

Pipeline

Definition: Pipeline script from SCM

SCM: Git

Repositories

Repository URL: github.com/kenych/maze-explorer

Please enter Git repository.

Credentials: - none -

Branches to build

Branch Specifier (blank for 'any'): */master

Repository browser: (Auto)

Additional Behaviours: Add

Script Path: Jenkinsfile

Here is how your changes in the POM will look, We added distributionManagement to the project section:

```
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://maven.apache.
    <modelVersion>4.0.0</modelVersion>
```

```
   <modelVersion>4.0.0</modelVersion>

   <groupId>kayan</groupId>
   <artifactId>maze</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>jar</packaging>

   <url>http://maven.apache.org</url>

   <properties>
       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   </properties>

   <distributionManagement>
       <snapshotRepository>
           <id>artifactory</id>
           <name>artifactory</name>
           <url>${artifactory_url}</url>
       </snapshotRepository>
   </distributionManagement>


(the rest of POM file)
```

The URL should point to the URL of Artifactory, so we need to pass it to Maven through Jenkins as an environment variable, just like we did for Sonar. Obviously it can't be static as the IP of the host can change.

If you run the job now, it is still going to fail, even if it has distributionManagement configured:

```
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4.459 s
[INFO] Finished at: 2017-07-16T20:13:34Z
[INFO] Final Memory: 13M/137M
[INFO] ------------------------------------------------------------------------
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-deploy-plugin:2.7:deploy (de
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
```

This is because we need to have credentials in order to deploy. The default credentials of JFrog are "admin:password." Try to log in and check. In order to pass credentials to the deployment plugin, we need to set them in the Maven settings.xml file.

Chapter 3: Configuring the Maven Settings File

If we have Maven installed locally, then we can run it first just to check that the deployment actually works with our configuration, and then we can start looking at how to configure it with Jenkins. Let's check what we have in the settings file:

```
mvn help:effective-settings
```

You will see something similar if the settings file is absent or empty.

```
<settings xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLS
  <localRepository xmlns="http://maven.apache.org/SETTINGS/1.1.0">/Users/kayanazimov/.m2/r
  <pluginGroups xmlns="http://maven.apache.org/SETTINGS/1.1.0">
    <pluginGroup>org.apache.maven.plugins</pluginGroup>
    <pluginGroup>org.codehaus.mojo</pluginGroup>
  </pluginGroups>
</settings>
```

Now let's go to /Users/YOUR_USER_NAME_HERE/.m2/ and change or create settings.xml as below. In order to pass credentials of the repo, we need to add a server in the servers section of the Maven settings:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.1.0 http://maven.apache.c
<servers>
    <server>
      <id>artifactory</id>
      <username>admin</username>
      <password>password</password>
    </server>
   </servers>
</settings>
```

Please note, the id should be same as the id you used earlier in the pom file for snapshotRepository.

You can check if Maven is picking it up:

```
mvn help:effective-settings
```

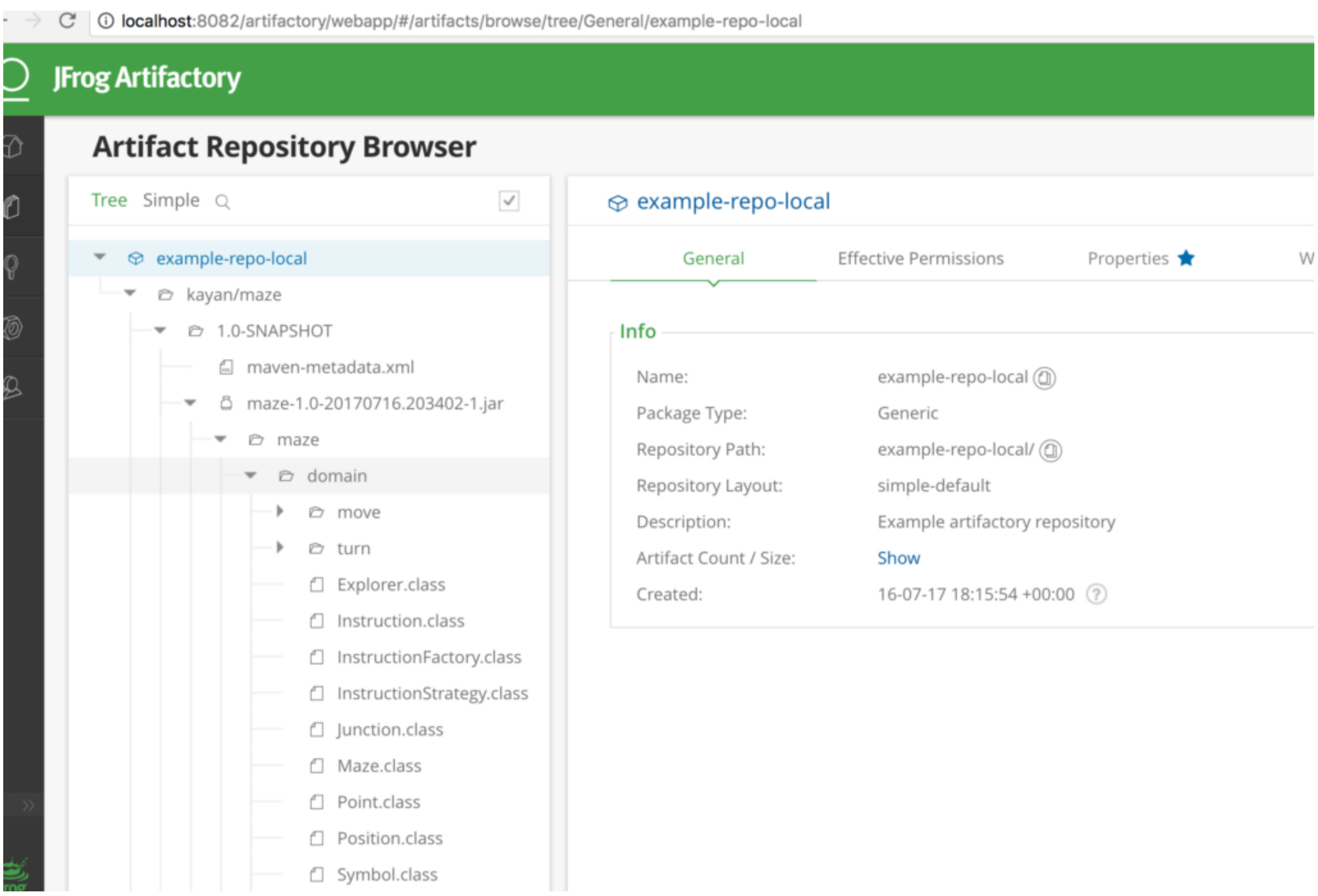
Now clone the project, or if you're using your own project, just run it from the project folder (use the

Now clone the project, or if you're using your own project, just run it from the project folder (use the Artifactory port, which will be shown when running the runall script):

```
mvn deploy -DskipTests -Dartifactory_url=http://localhost:8082/artifactory/example-repo-lo

[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building maze 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ maze ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /Users/kayanazimov/workspace/learn/maze-explore
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ maze ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ maze ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 10 resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:testCompile (default-testCompile) @ maze ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ maze ---
[INFO] Tests are skipped.
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ maze ---
[INFO] Building jar: /Users/kayanazimov/workspace/learn/maze-explorer/target/maze-1.0-SNAP
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ maze ---
[INFO] Installing /Users/kayanazimov/workspace/learn/maze-explorer/target/maze-1.0-SNAPSHO
[INFO] Installing /Users/kayanazimov/workspace/learn/maze-explorer/pom.xml to /Users/kayan
[INFO]
[INFO] --- maven-deploy-plugin:2.7:deploy (default-deploy) @ maze ---
Downloading: http://localhost:8082/artifactory/example-repo-local/kayan/maze/1.0-SNAPSHOT/
Uploading: http://localhost:8082/artifactory/example-repo-local/kayan/maze/1.0-SNAPSHOT/ma
Uploaded: http://localhost:8082/artifactory/example-repo-local/kayan/maze/1.0-SNAPSHOT/maz
Uploading: http://localhost:8082/artifactory/example-repo-local/kayan/maze/1.0-SNAPSHOT/ma
```

```
Uploaded: http://localhost:8082/artifactory/example-repo-local/kayan/maze/1.0-SNAPSHOT/maz
Downloading: http://localhost:8082/artifactory/example-repo-local/kayan/maze/maven-metadat
Uploading: http://localhost:8082/artifactory/example-repo-local/kayan/maze/1.0-SNAPSHOT/ma
Uploaded: http://localhost:8082/artifactory/example-repo-local/kayan/maze/1.0-SNAPSHOT/mav
Uploading: http://localhost:8082/artifactory/example-repo-local/kayan/maze/maven-metadata.
Uploaded: http://localhost:8082/artifactory/example-repo-local/kayan/maze/maven-metadata.x
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.828 s
[INFO] Finished at: 2017-07-18T09:39:52+01:00
[INFO] Final Memory: 13M/207M
[INFO] ------------------------------------------------------------------------

➜  maze-explorer git:(master)
```

Now check the Artifactory:



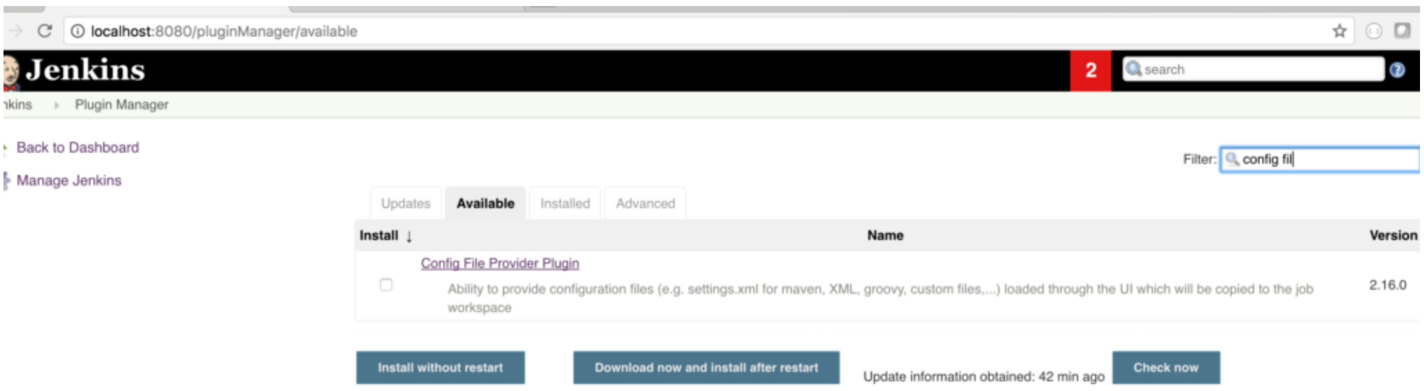
Name: example-repo-local
Package Type: Generic
Repository Path: example-repo-local/
Repository Layout: simple-default
Description: Example artifactory repository
Artifact Count / Size: Show
Created: 16-07-17 18:15:54 +00:00

As you can see we successfully deployed it, Yay! Now it is time to prepare this step in Jenkins.

Chapter 4: Using the Config File Provider Plugin

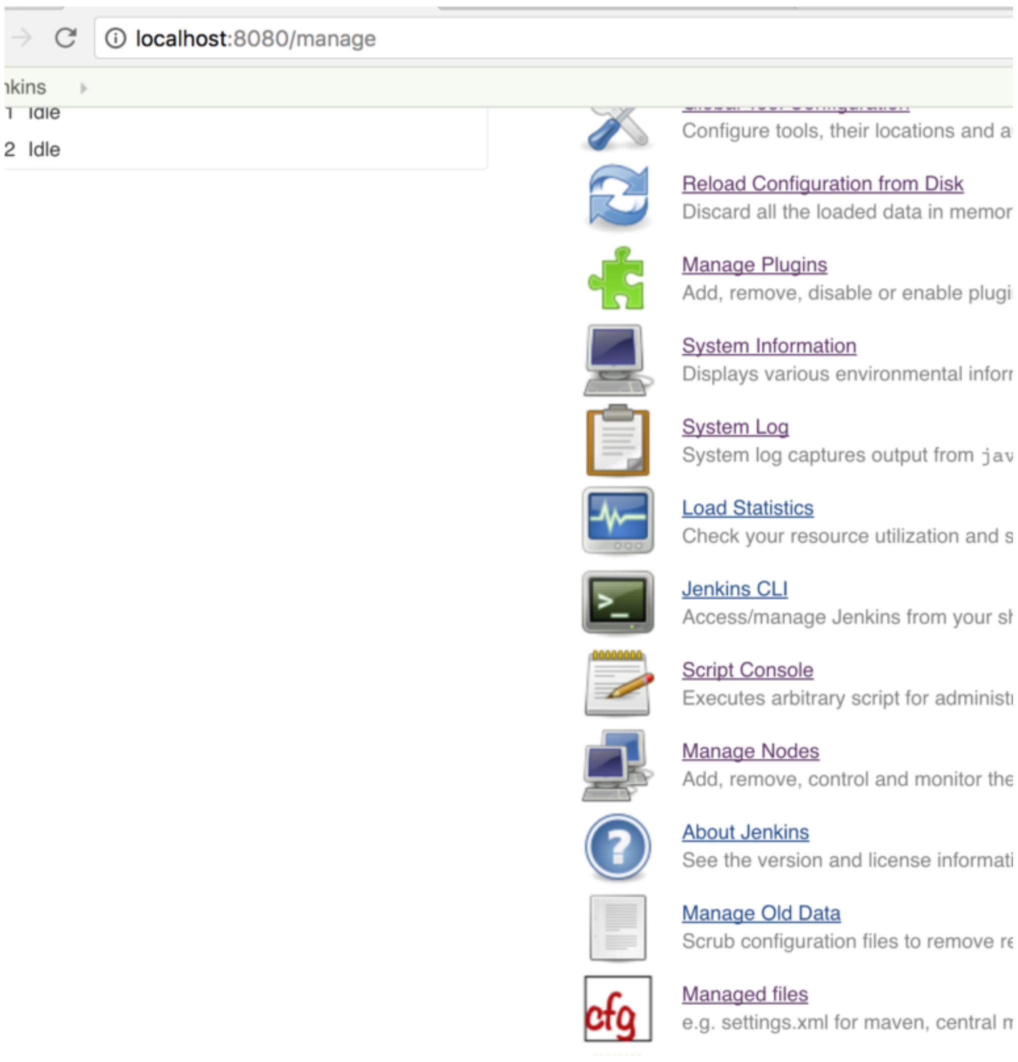
for Persistence of Maven Settings

In order to apply the settings file to Maven in Jenkins, we need the config file provider plugin, which lets us preserve multiple settings files (please be aware, we might need more than one settings depending on a project running the job in real life). Now let’s install the plugin manually first:

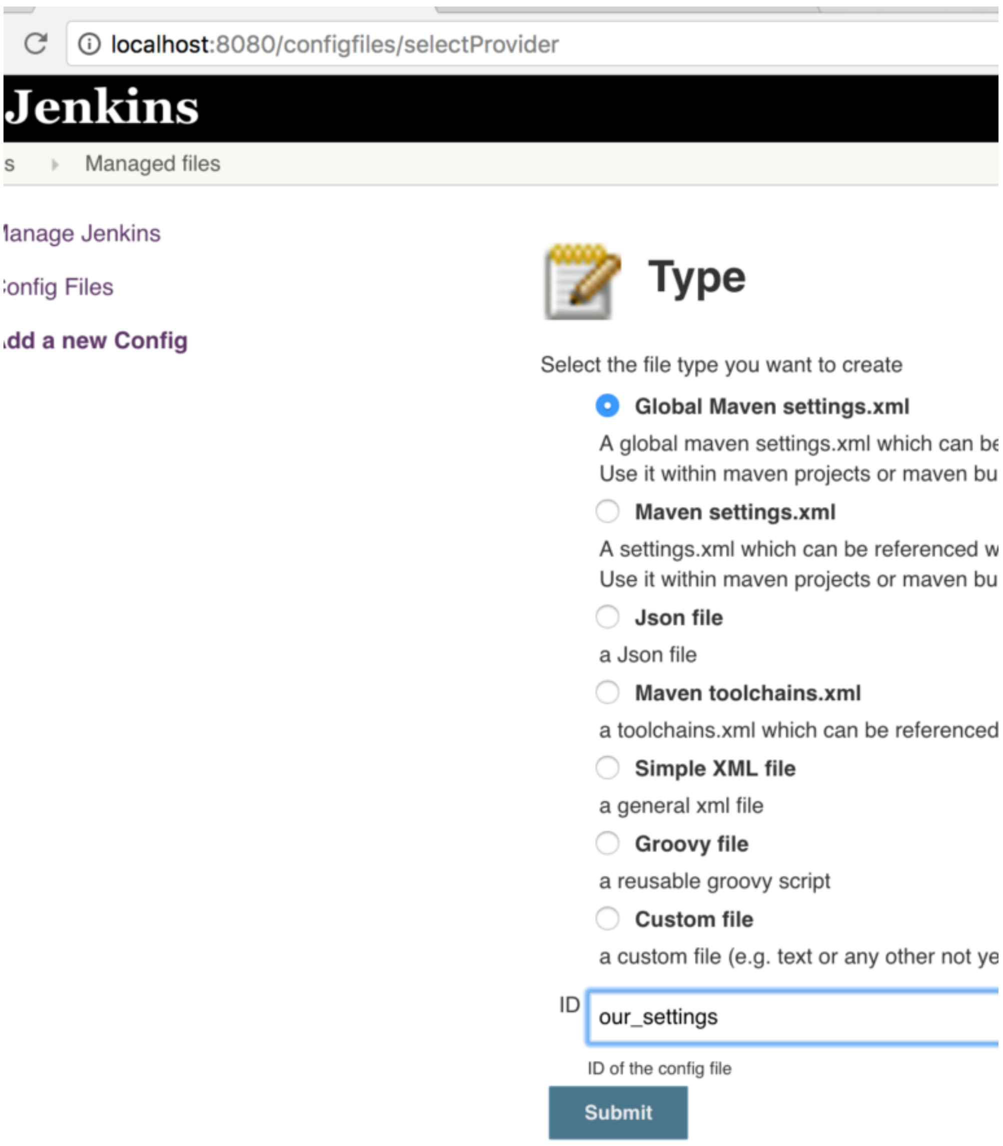

localhost:8080/pluginManager/available
Jenkins
Plugin Manager
Back to Dashboard
Manage Jenkins
Filter: config fil
Updates
Available
Installed
Advanced
Install ↓
Name
Version
Config File Provider Plugin
Ability to provide configuration files (e.g. settings.xml for maven, XML, groovy, custom files,...) loaded through the UI which will be copied to the job workspace
2.16.0
Install without restart
Download now and install after restart
Update information obtained: 42 min ago
Check now

We need to set up the Maven settings file in the plugin. Go to manage Jenkins, managed files, and add new config, set the id to “our_settings” and copy the content from the settings.xml we used before:

Click Managed files:


localhost:8080/manage
1 Idle
2 Idle
Configure tools, their locations and a
Reload Configuration from Disk
Discard all the loaded data in memor
Manage Plugins
Add, remove, disable or enable plugi
System Information
Displays various environmental infor
System Log
System log captures output from jav
Load Statistics
Check your resource utilization and s
Jenkins CLI
Access/manage Jenkins from your sh
Script Console
Executes arbitrary script for administ
Manage Nodes
Add, remove, control and monitor the
About Jenkins
See the version and license informati
Manage Old Data
Scrub configuration files to remove re
Managed files
e.g. settings.xml for maven, central m

Select Global Maven Settings and set id:



localhost:8080/configfiles/selectProvider

Jenkins

s ▸ Managed files

ſanage Jenkins
ːonfig Files
ːdd a new Config

Type

Select the file type you want to create

[✓ Global Maven settings.xml]
A global maven settings.xml which can be
Use it within maven projects or maven bu

Maven settings.xml
A settings.xml which can be referenced w
Use it within maven projects or maven bu

Json file
a Json file

Maven toolchains.xml
a toolchains.xml which can be referenced

Simple XML file
a general xml file

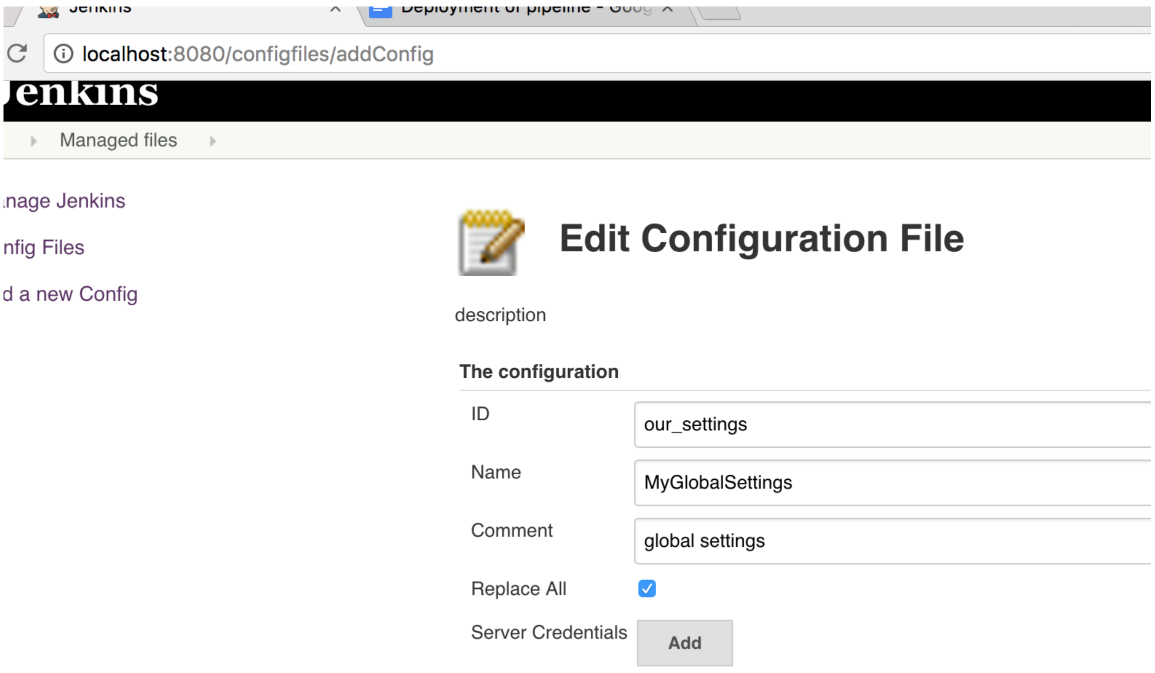
Groovy file
a reusable groovy script

Custom file
a custom file (e.g. text or any other not ye

ID: our_settings
ID of the config file

Submit

And set content:



localhost:8080/configfiles/addConfig

enkins

▸ Managed files ▸

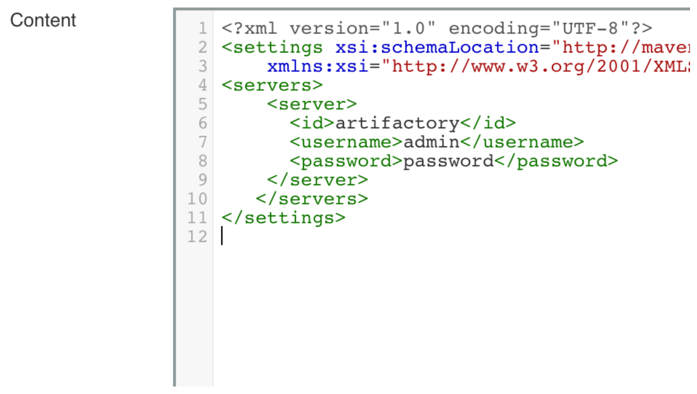
ːnage Jenkins
nfig Files
d a new Config

Edit Configuration File

description

The configuration

ID	our_settings
Name	MyGlobalSettings
Comment	global settings
Replace All	✓
Server Credentials	Add

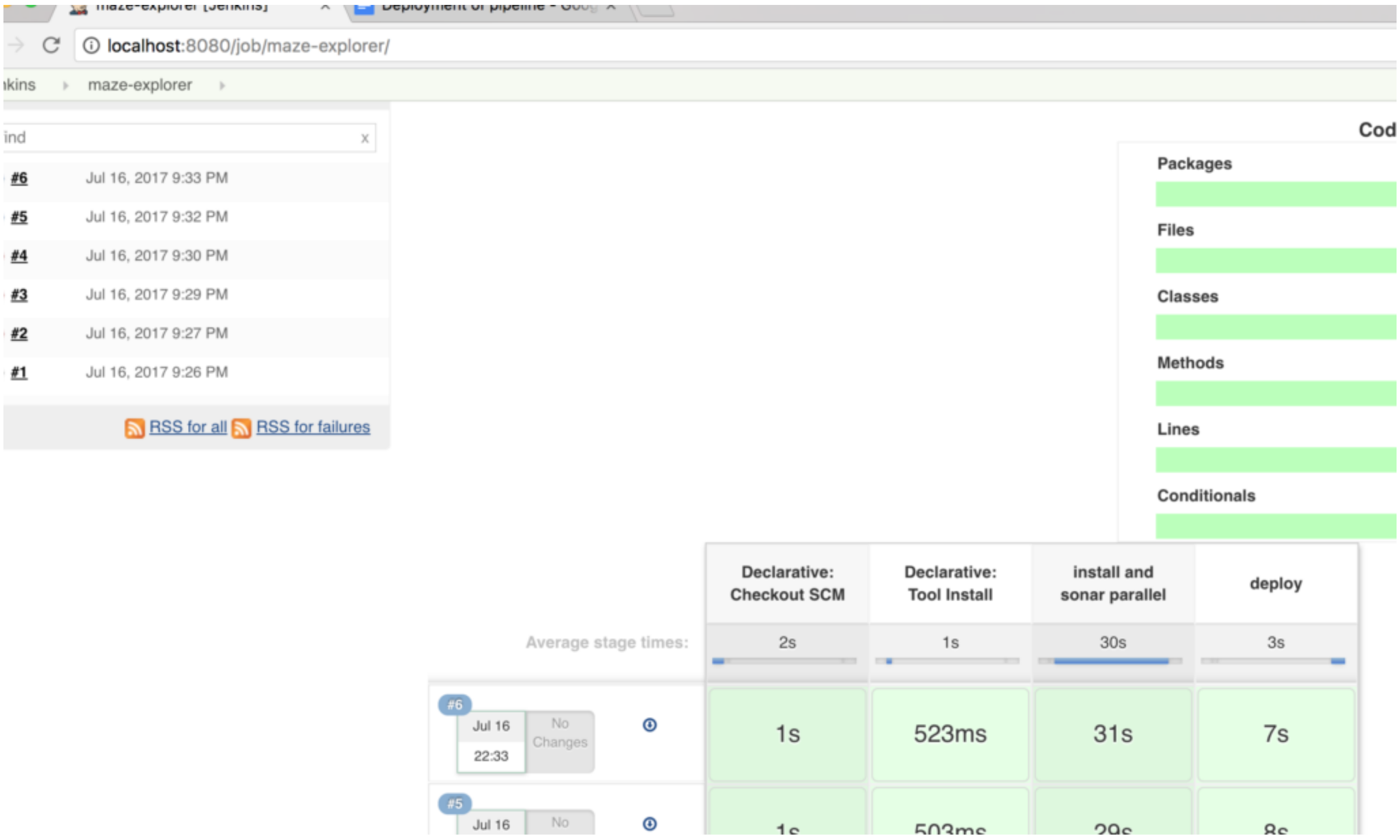


```xml
<?xml version="1.0" encoding="UTF-8"?>
<settings xsi:schemaLocation="http://maven
    xmlns:xsi="http://www.w3.org/2001/XML
<servers>
    <server>
      <id>artifactory</id>
      <username>admin</username>
      <password>password</password>
    </server>
   </servers>
</settings>
```

Now we can use the Config File Provider plugin in our pipeline. Please Replay the job and update the pipeline as below:

```
pipeline {
    agent any

    tools {
        jdk 'jdk8'
        maven 'maven3'
    }

    stages {
        stage('install and sonar parallel') {
            steps {
                parallel(
                        install: {
                            sh "mvn -U clean test cobertura:cobertura -Dcobertura.report.f
                        },
                        sonar: {
                            sh "mvn sonar:sonar -Dsonar.host.url=${env.SONARQUBE_HOST}"
                        }
                )
            }
            post {
                always {
                    junit '**/target/*-reports/TEST-*.xml'
                    step([$class: 'CoberturaPublisher', coberturaReportFile: 'target/site/
                }
            }
        }
        stage ('deploy'){
            steps{
                configFileProvider([configFile(fileId: 'our_settings', variable: 'SETTINGS
                    sh "mvn -s $SETTINGS deploy -DskipTests -Dartifactory_url=${env.ARTIF
                    }
                }
            }
        }
    }
}
```

Let's run the build:



If you are lucky, you will get the screen above. Otherwise, read what can go wrong with containers when running out of memory.

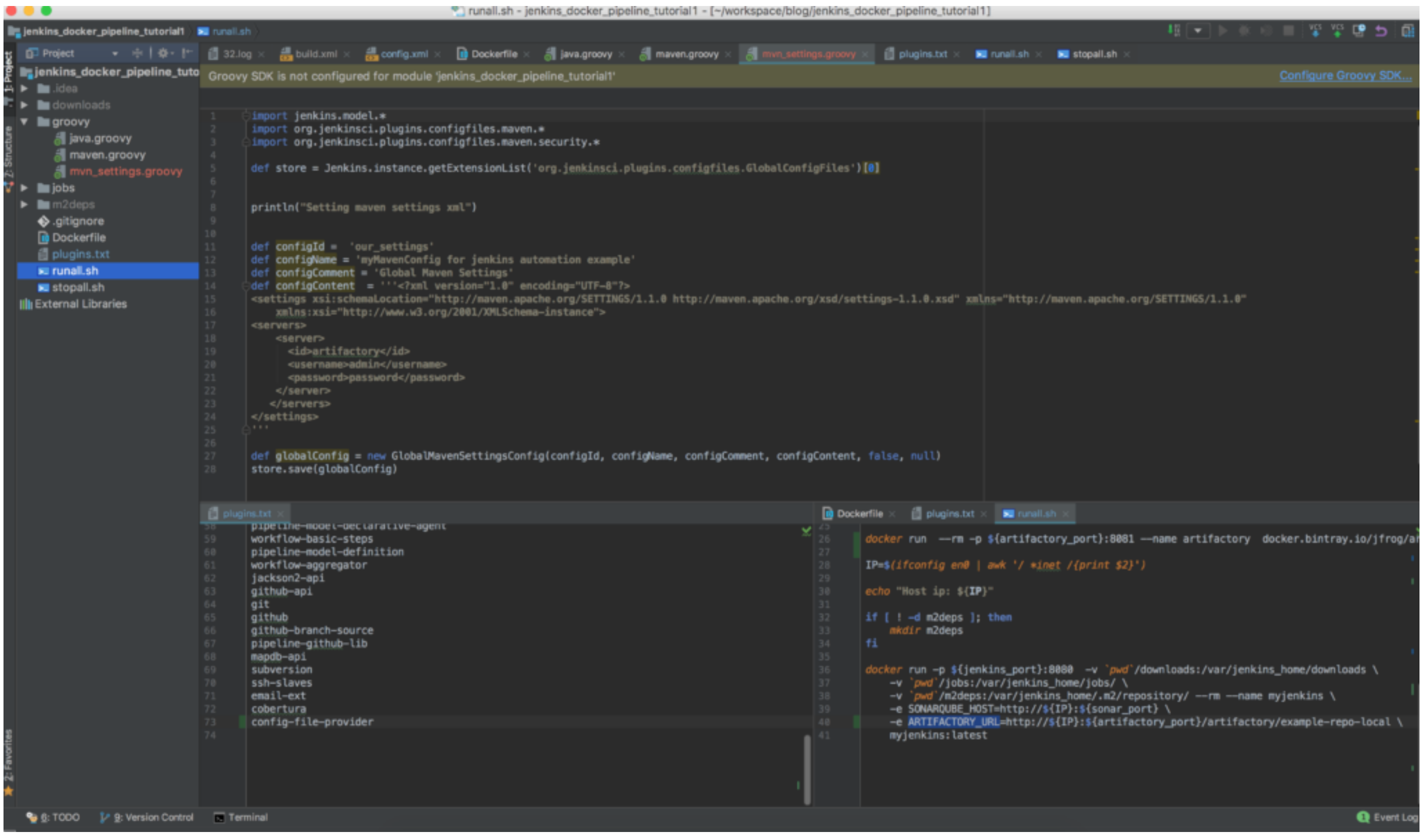
Chapter 5: Dockerizing the Installation and Configuration Process

Now let's dockerize the plugin installation and configuration. Create mvn_settings.groovy file, copy it to the groovy folder that we created in the first part of the tutorial, and set content as below:

```
import jenkins.model.*
import org.jenkinsci.plugins.configfiles.maven.*
import org.jenkinsci.plugins.configfiles.maven.security.*

def store = Jenkins.instance.getExtensionList('org.jenkinsci.plugins.configfiles.GlobalCor



println("Setting maven settings xml")


```

```
def configId =  'our_settings'
def configName = 'myMavenConfig for jenkins automation example'
def configComment = 'Global Maven Settings'
def configContent  = '''<?xml version="1.0" encoding="UTF-8"?>
<settings xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.1.0 http://maven.apache.c
<servers>
   <server>
     <id>artifactory</id>
     <username>admin</username>
     <password>password</password>
   </server>
  </servers>
</settings>
'''

def globalConfig = new GlobalMavenSettingsConfig(configId, configName, configComment, conf
store.save(globalConfig)
```

And don't forget to install the "config-file-provider" plugin, just add it to the plugins.txt like in this screenshot:



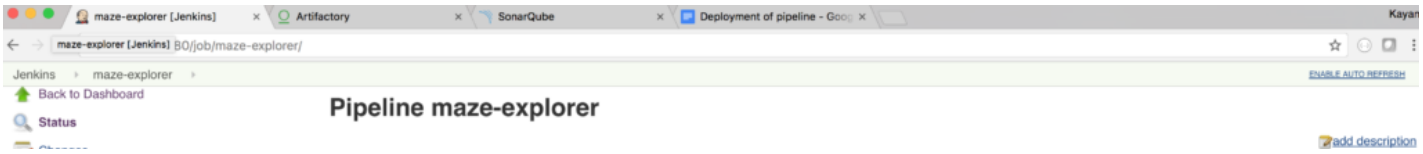
Groovy SDK is not configured for module 'jenkins_docker_pipeline_tutorial1'

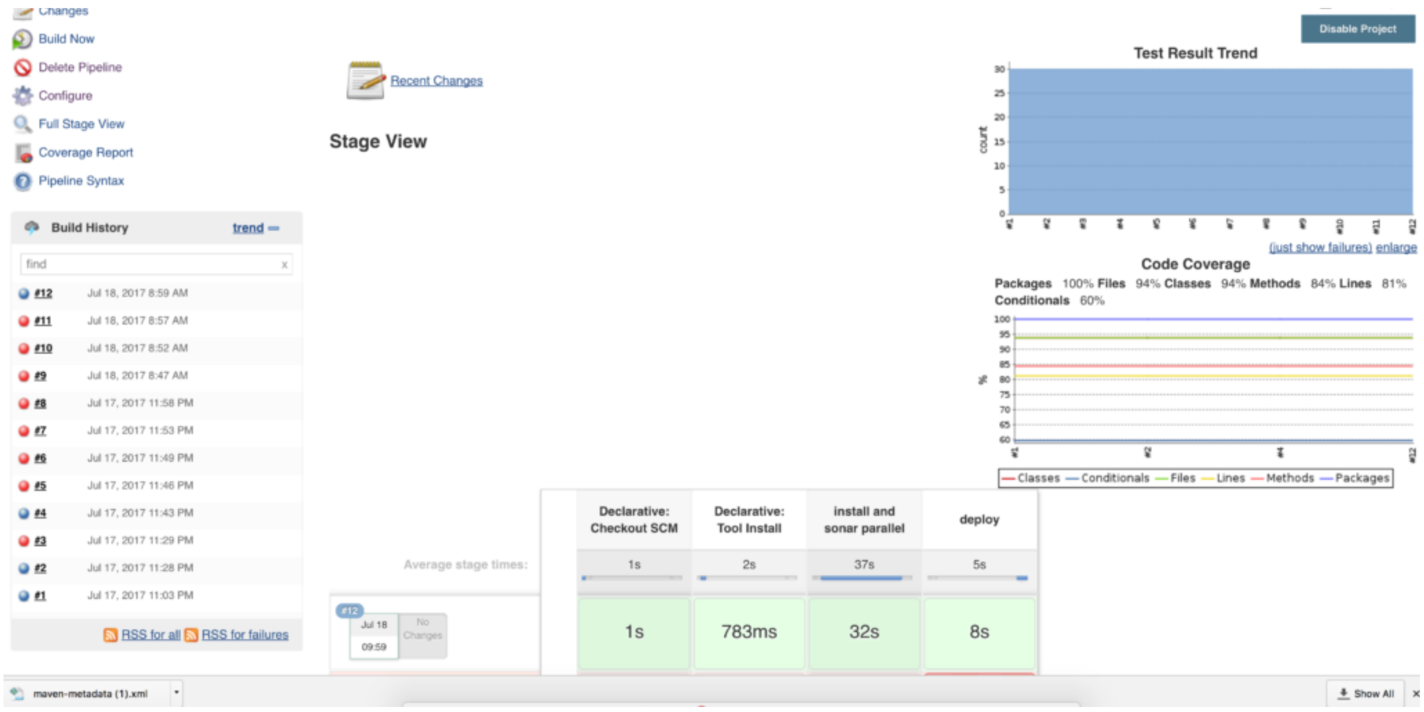
```
import jenkins.model.*
import org.jenkinsci.plugins.configfiles.maven.*
import org.jenkinsci.plugins.configfiles.maven.security.*

def store = Jenkins.instance.getExtensionList('org.jenkinsci.plugins.configfiles.GlobalConfigFiles')[0]


println("Setting maven settings xml")


def configId =  'our_settings'
def configName = 'myMavenConfig for jenkins automation example'
def configComment = 'Global Maven Settings'
def configContent  = '''<?xml version="1.0" encoding="UTF-8"?>
<settings xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.1.0 http://maven.apache.org/xsd/settings-1.1.0.xsd" xmlns="http://maven.apache.org/SETTINGS/1.1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<servers>
    <server>
      <id>artifactory</id>
      <username>admin</username>
      <password>password</password>
    </server>
   </servers>
</settings>
'''

def globalConfig = new GlobalMavenSettingsConfig(configId, configName, configComment, configContent, false, null)
store.save(globalConfig)
```

```
workflow-basic-steps
pipeline-model-definition
workflow-aggregator
jackson2-api
github-api
git
github
github-branch-source
pipeline-github-lib
mapdb-api
subversion
ssh-slaves
email-ext
cobertura
config-file-provider
```

```
docker run  --rm -p ${artifactory_port}:8081 --name artifactory  docker.bintray.io/jfrog/ar

IP=$(ifconfig en0 | awk '/ *inet /{print $2}')

echo "Host ip: ${IP}"

if [ ! -d m2deps ]; then
    mkdir m2deps
fi

docker run -p ${jenkins_port}:8080  -v `pwd`/downloads:/var/jenkins_home/downloads \
    -v `pwd`/jobs:/var/jenkins_home/jobs/ \
    -v `pwd`/m2deps:/var/jenkins_home/.m2/repository/ --rm --name myjenkins \
    -e SONARQUBE_HOST=http://${IP}:${sonar_port} \
    -e ARTIFACTORY_URL=http://${IP}:${artifactory_port}/artifactory/example-repo-local \
    myjenkins:latest
```

Time to rebuild and rerun your Jenkins, just run the runall script. Once Jenkins is ready, run the build:





Build History trend

#12	Jul 18, 2017 8:59 AM
#11	Jul 18, 2017 8:57 AM
#10	Jul 18, 2017 8:52 AM
#9	Jul 18, 2017 8:47 AM
#8	Jul 17, 2017 11:58 PM
#7	Jul 17, 2017 11:53 PM
#6	Jul 17, 2017 11:49 PM
#5	Jul 17, 2017 11:46 PM
#4	Jul 17, 2017 11:43 PM
#3	Jul 17, 2017 11:29 PM
#2	Jul 17, 2017 11:28 PM
#1	Jul 17, 2017 11:03 PM

Packages 100% Files 94% Classes 94% Methods 84% Lines 81% Conditionals 60%

	Declarative: Checkout SCM	Declarative: Tool Install	install and sonar parallel	deploy
Average stage times:	1s	2s	37s	5s
#12 Jul 18 09:59 No Changes	1s	783ms	32s	8s

We now have the CI pipeline for our project and almost fully automated Jenkins. Why almost?

Because we need to hide the passwords firstly. But I am running late, and my girlfriend has already started complaining...

But I promise we will look at how to use encrypted passwords in Jenkins very soon in the next session.

Again, If you were lazy and don't like "hands on" stuff, just clone the complete code for the second part of the tutorial, run it, enjoy it, and share it if you like:

```
git clone https://github.com/kenych/dockerizing-jenkins-part-2 && cd dockerizing-jenkins-p
```

Hope you managed to run everything without any issues and enjoyed this tutorial. Should you have any issue don't hesitate to comment and I will try to help.

Like This Article? Read More From DZone

Unleash the DevOps!

Setting Up Jenkins to Deploy to Heroku

Orchestrating Workflows With Jenkins and Docker

Free DZone Refcard
Getting Started With Docker

Topics: DOCKER , DEPLOY , ARTIFACTORY , JENKINS

DevOps Partner Resources

DDD, Part 2: DDD Building Blocks

by M Yauri Maulana at-Tamimi · Dec 30, 17 · DevOps Zone

Learn more about how CareerBuilder was able to resolve customer issues 5x faster by using Scalyr, the fastest log management tool on the market.

It's been a while since I wrote the first part of my DDD series. In this second part, I will continue with one of the most important things to know about DDD before you go further with your implementation. First things first: you must know the DDD building blocks below:

1. Entities
2. Value objects
3. Aggregate roots
4. Repositories
5. Factories
6. Services

Fasten your seatbelts!. We're going through the details of those blocks now.

1. Entities

An entity is a plain object that has an identity (ID) and is potentially mutable. Each entity is uniquely identified by an ID rather than by an attribute; therefore, two entities can be considered equal (identifier equality) if both of them have the same ID even though they have different attributes. This means that the state of the entity can be changed anytime, but as long as two entities have the same ID, both are considered equal regardless what attributes they have.

2. Value Objects

Value objects are immutable. They have no identity (ID) like we found in entity. Two value objects can be considered equal if both of them have the same type and the same attributes (applied to all of its attributes).

There are often uses for a thing like message passing and in fact, this is particularly useful in the API layer within an onion architecture to expose your domain concepts without necessarily exposing the immutable aspect.

Some benefits of value objects:

1. The compound of value objects can swallow lots of computational complexity.

2. Entities can be released from logic complexity.

3. Improve extensibility, especially for testability and concurrency issues if using correctly.

3. Aggregate Roots

Aggregate root is an entity that binds together with other entities. Moreover, aggregate root is actually a part of aggregate (collection/cluster of associated objects that are treated as a single unit for the purpose of data changes). Thus, each aggregate actually consists of an aggregate root and a boundary. For example, the relationship between Order and OrderLineItem within SalesOrderDomain can be considered as an aggregate where Order acts as the aggregate root, while the OrderLineItem is the child of Order within SalesOrder boundary.

One of the key features of an aggregate root is that the external objects are not allowed to holds a reference to an aggregate root child entities. Thus, if you need access to one of the aggregate root child entities (AKA aggregate), then you must go through the aggregate root (i.e. you can't access the child directly).

The other thing is that all operations within the domain should, where possible, go through an aggregate root. Factories, repositories, and services are some exceptions to this, but whenever possible, if you can create or require that an operation goes through the aggregate root, that's going to be better.

4. Repositories

Repositories are mostly used to deal with the storage. They are actually one of the most important concepts on the DDD because they have abstracted away a lot of the storage concerns (i.e. some form/mechanism of storage).

The repository implementation could be a file-based storage, or database (SQL-/NoSQL-based), or any other thing that is related to storage mechanism, such as caching. Any combination of those is also possible.

A repository should not be confused with the data store. A repository job is to store aggregate roots. Underneath that, repositories implementation may actually have to talk to multiple different storage locations in order to construct the aggregates. Thus, a single aggregate root might be drawn from a REST API, as well as a database or files. You may wrap those in something called the data store, but the repository is sort of a further layer of an abstraction on top of all those individual data stores. Usually, I implement the repository as an interface within the domain/domain services layer within onion architecture, and then the implementation logic of the repository interface is going to be defined in the infrastructure layer.

5. Factories

The factories are used to give an abstraction to the object construction (see factory design pattern from GOF).

A factory can also potentially return an aggregate root or an entity, or perhaps a value object. Often times, when you need a factory method for an aggregate root, it will be rolled into the repository. So, your repository might have a finder create method on it.

Usually, factories also implemented as an interface within the domain/domain services layer with the implementation logic will be defined in the infrastructure layer.

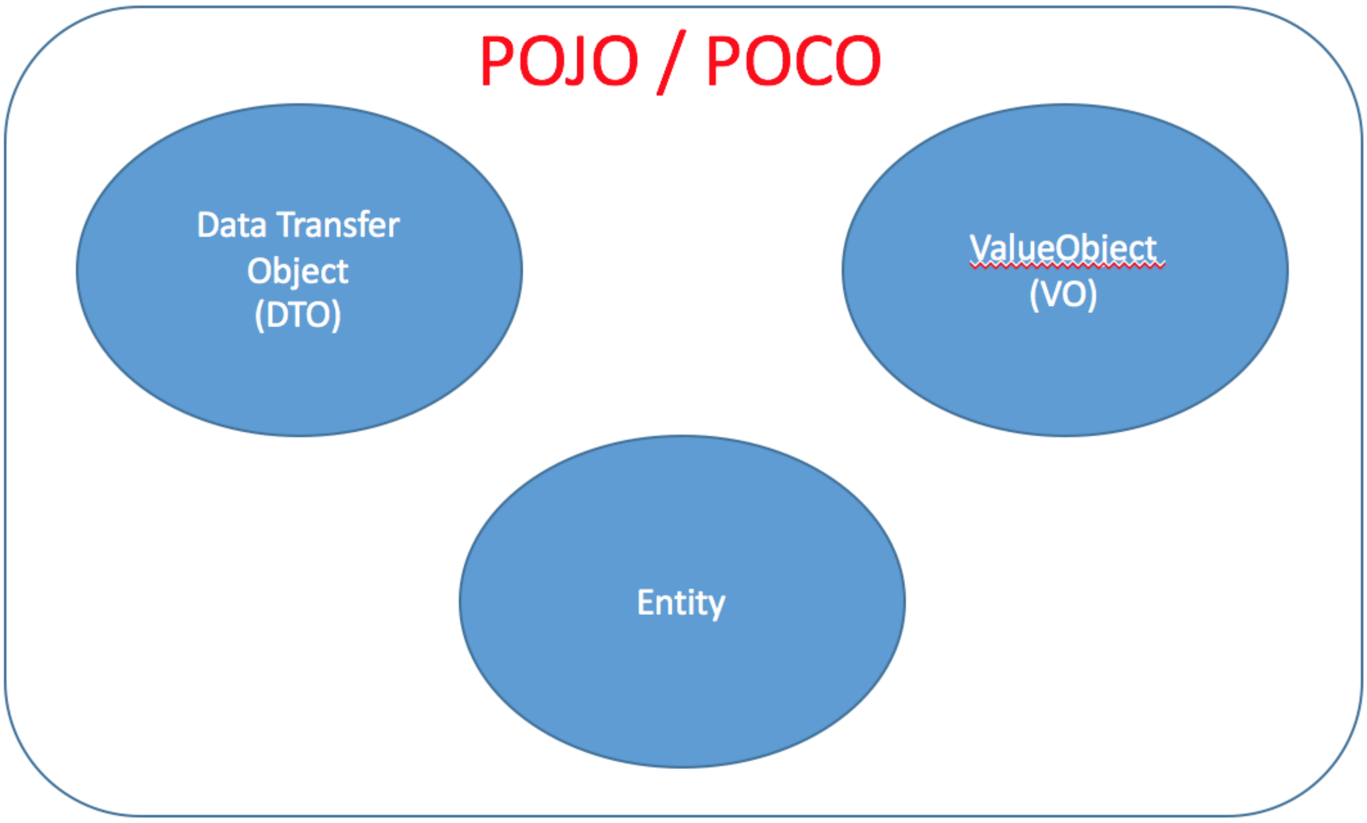
6. Services

A service basically exists to provide a home for operations that don't quite fit into an aggregate root. As an example, when you have an operation and don't know which aggregate root it goes into, perhaps it operates on multiple aggregate roots or maybe it doesn't belong to any existing aggregate root. Then, you can put the logic into a service. However, don't be rushed to put everything into a service. First and foremost, it's better to carefully analyze whether the operation fits into one of the existing aggregate roots. If you couldn't find the aggregate root, it's subsequently better to ask yourself if you have missed one aggregate root, or perhaps there are domain concepts that you haven't considered that should be brought into your domain before you put the operation into a service.

Other Important Things

I often found that many developers use term VO (value objects) and DTO (data transfer object) interchangeably. They think both are just the same. This is quite annoying for me. I'd like to clarify here that both refer to the different things.

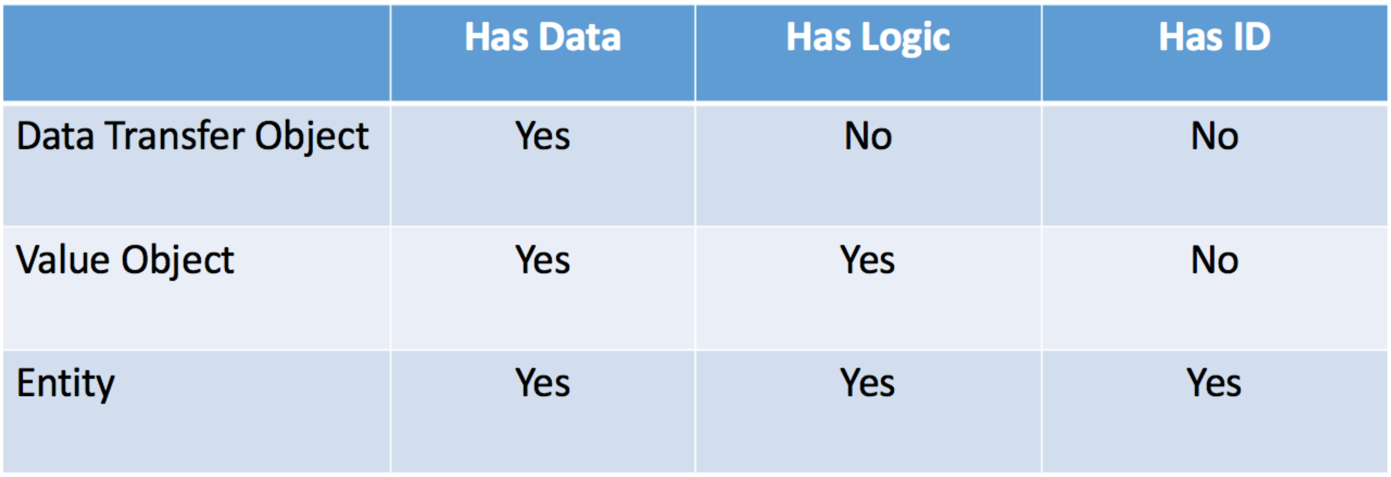
As depicted in the picture below, VO and DTO are subsets of a POJO/POCO. An entity is also a subset of POJO/POCO.



In the above depiction, POJO and POCO can be interchangeably used. Both are referring to similar things. Both are just domain objects that mostly represent the domain/business object within the business application.

The term POJO (plain old Java object) was coined by Martin Fowler and very popular in Java community, while POCO (plain old CLR object/plain old class object) is widely used in the dotNet.

As mentioned earlier, DTO, VO, and entity are just a subset of POJO/POCO. However, they are really different things as described below:



	Has Data	Has Logic	Has ID
Data Transfer Object	Yes	No	No
Value Object	Yes	Yes	No
Entity	Yes	Yes	Yes

DTO is merely a stupid data container (only holds data without any logic). Thus, it's anemic in general (only contains attributes and getter/setter). DTO is absolutely immutable. Usually, we use DTO to transfer the object between layers and tiers in one single application or between application to application or JVM to JVM (mostly useful between networks to reduce multiple network call).

VO is also immutable, but what makes it different than DTO is that VO also contains logic.

That's all for now. Read the next part here.