



Dockerizing Jenkins: Protección de contraseñas con docker-compose, docker-secret y Jenkins Credentials Plugin

por Kayan Azimov · 23 de agosto, 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.

This is the third part of Dockerizing Jenkins series. You can find the previous parts here:

Dockerizing Jenkins 2, Part 1: Declarative Build Pipeline With SonarQube Analysis

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

In this part we will look at:

1. **How to use docker-compose to run containers.**
2. **How to use passwords in docker environment with docker-secrets.**
3. **How to hide sensitive information in Jenkins with credentials plugin.**

In Part 1, we created a basic Jenkins Docker image in order to run a Java Maven pipeline with test and SonarQube analysis. In Part 2, we looked at how to perform deployment using the Maven settings file. As you remember, we saved the password in the file without any encryption, which is not something you would ever do, of course.

All the code for this and the previous parts is in my GitHub repo, and I decided to create a branch for every part, as the master branch will change with every part and older articles would refer to the wrong code base. For this part, the code will be in the branch

“dockerizing_jenkins_part_3_docker_compose_docker_secret_credentials_plugin” and you can run the below command to check it out:

```
1 git clone https://github.com/kenych/dockerizing-jenkins && \  
2   cd dockerizing-jenkins && \  
   git checkout dockerizing_jenkins_part_3_docker_compose_docker_secret_credentials_plugi  
3
```

In this part, we will remove the password from the source code and let the credentials plugin apply credentials to the Config File Provider Plugin. But before changing any code, we will need to switch to using docker-compose instead of using the docker run command. This will give us a chance to leverage the docker secrets feature, along with many other features which you will love.

I updated the **runall.sh** script, which we used in the two parts before, and replaced it with the docker-compose and download.sh script, which will just download the minimum stuff we will need in advance. I also removed Java 7 and Java 8 installation in favor of using embedded Java 8 from the Jenkins container, as otherwise our download script takes too long and Java comes for free in the image anyway. You can check it later once our Jenkins container is running.

If you were following part one and two, you should know how to pick up the specific Java version anyway using the Maven tool mechanism, and if you want to play with that, just uncomment these lines in the download script, java.groovy, and in the pipeline as well. Now let's run the download to make sure we have everything we need:

```

1  → ./download.sh
2  2.60.1: Pulling from library/jenkins
3  Digest: sha256:fa62fceb220e7545d1791e6eea6759b4c3bdba246dd839289f2b28b653e72
4  Status: Image is up to date for jenkins:2.60.1
5  6.3.1: Pulling from library/sonarqube
6  Digest: sha256:d5f7bb8a6da054bf28d111e5a27f1378188b427db64cc9fb392e1a8d80a
7  Status: Image is up to date for sonarqube:6.3.1
8  5.4.4: Pulling from jfrog/artifactory-oss
9  Digest: sha256:404a3f0bfdfa0108159575ef74ffd4afaff349b856966ddc49f6401cd2f20d7d
10 Status: Image is up to date for docker.bintray.io/jfrog/artifactory-oss:5.4.4
11 % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
12           %             %             Dload  Upload   Total   Spent    Left   Speed
13 100 8334k  100 8334k    0     0  445k      0  0:00:18  0:00:18 --:--:--  444k

```

Please note that if you haven't ever downloaded the images, it will take some time. Now, while it is downloading the stuff we need, let's look at docker-compose.yml:

```

1  version: "3.1"
2
3  services:
4    myjenkins:
5      build:
6        context: .
7      image: myjenkins
8      ports:
9        - "8080:8080"
10     depends_on:
11       - mysonar
12       - artifactory
13     links:
14       - mysonar
15       - artifactory
16     volumes:
17       - "./jobs:/var/jenkins_home/jobs/"
18       - "./m2deps:/var/jenkins_home/.m2/repository/"
19       - "./downloads:/var/jenkins_home/downloads"

```

```
20     secrets:
21       - artifactoryPassword
22   mysonar:
23     image: sonarqube:6.3.1
24     ports:
25       - "9000"
26   artifactory:
27     image: docker.bintray.io/jfrog/artifactory-oss:5.4.4
28     ports:
29       - "8081"
30
31   secrets:
32     artifactoryPassword:
33       file: ./secrets/artifactoryPassword
```

If you were curious, you would ask, why did I call the file `docker-compose.yml`?

Well, this is a convenient way, as otherwise you can't call `docker-compose` commands without the explicit file argument `-f`, otherwise you would get this error:

```
1  docker-compose up
2  ERROR:
3      Can't find a suitable configuration file in this directory or any
4      parent. Are you in the right directory?
5      Supported filenames: docker-compose.yml, docker-compose.yaml
```

Now let's investigate the compose file. You may have noticed the "services" section; that is where all containers go. Because we are going to build our own Jenkins image, we added a "build" section to it.

Next, "depends on" will wait for other services before running the Jenkins container.

Then "links" will make it possible to refer to other containers by service name from within the container. This is really cool as we don't need to define in our pom file the dynamic IP address for the artifactory URL anymore; instead, we can just write `http://artifactory:8081/artifactory/example-repo-local` and Jenkins will be able to resolve "artifactory" to its IP address.

The very interesting part is secrets. It will bind mount docker secret files, to which we can later refer from container by `"/run/secrets/secret_name_here"` path. You may ask, couldn't we simply bind mount just a password file to refer to from within the Groovy script? Well, we could, but best practices require referring to sensitive password information through docker secrets (in a Swarm environment, and here is why).

In this tutorial, we won't use Swarm where you could create your password through "swarm init" and "echo "your_password_here" | docker secret create artifactoryPassword – " but instead use `docker-compose`. So I created the file "artifactoryPassword" under the secrets folder without the password for artifactory in it; instead, it says "write your password here!" This is to show that you should never save the actual password in the repo. So please update it with the actual password, which is "password," the default password for JFrog artifactory.

Now we are ready to run the containers and later check if the secret file is there.

```
1  docker-compose up
```

```

1 Creating network "dockerizingjenkinspart2_default" with the default driver
2 Building myjenkins
3 Step 1/6 : FROM jenkins:2.60.1
4 ---> f426a52bafa9
5 Step 2/6 : MAINTAINER Kayan Azimov
6 ---> Using cache
7 ---> 760e7bb0f335
8 Step 3/6 : ENV JAVA_OPTS "-Djenkins.install.runSetupWizard=false"
9 ---> Using cache
10 ---> e3dbac0834cd
11 Step 4/6 : COPY plugins.txt /usr/share/jenkins/ref/plugins.txt
12 ---> Using cache
13 ---> 76193d716609
14 Step 5/6 : RUN /usr/local/bin/install-plugins.sh < /usr/share/jenkins/ref/plugins.txt
15 ---> Using cache
16 ---> 2cbf4376a0a9
17 Step 6/6 : COPY groovy/* /usr/share/jenkins/ref/init.groovy.d/
18 ---> 32c36863caef
19 Removing intermediate container 7d0a005ecf02
20 Successfully built 32c36863caef
21 Successfully tagged myjenkins:latest
22 WARNING: Image for service myjenkins was built because it did not already exist. To rebuild
23

```

Pay attention to this warning: “Image for service myjenkins was built because it did not already exist. To rebuild this image you must use `docker-compose build` or `docker-compose up --build`.”

If you don’t get it, that means you have already got that image from the previous tutorial. If this happens, run “docker-compose build --no-cache” first and then “docker-compose up” so it recreates a new updated Jenkins image from this source code. Otherwise, Docker will try to use the cache by figuring out if anything changed and hoping that the COPY command in the Dockerfile could be deterministic. Well, perhaps it can, if the hash of the files is used. But I ran into a problem with that for some reason, as Groovy files, which are copied by “COPY groovy/* /usr/share/jenkins/ref/init.groovy.d/”, were just used from the cache, even after changing them, awkward! But again, if you ever suspect Docker not picking up the updated source code, just use --no-cache.

Now let’s look at the Jenkins’ container secret’s file location :

```

1 docker exec -it dockerizingjenkinspart2_myjenkins_1 cat /run/secrets/artifactoryPassword

```

And you should see the password for Artifactory.

Let’s make sure we have the default Java installation we mentioned before:

```

1 docker exec -it dockerizingjenkins_myjenkins_1 java -version
2 openjdk version "1.8.0_131"
3 OpenJDK Runtime Environment (build 1.8.0_131-8u131-b11-2-b11)
4 OpenJDK 64-Bit Server VM (build 25.131-b11, mixed mode)

```

Time to dockerize Jenkins with the credentials plugin. We will need to add another Groovy script for saving our credentials:

```

1  import jenkins.model.*
2  import com.cloudbees.plugins.credentials.*
3  import com.cloudbees.plugins.credentials.common.*
4  import com.cloudbees.plugins.credentials.domains.*
5  import com.cloudbees.plugins.credentials.impl.*
6  import com.cloudbees.jenkins.plugins.sshcredentials.impl.*
7
8  println("Setting credentials")
9
10 def domain = Domain.global()
11 def store = Jenkins.instance.getExtensionList('com.cloudbees.plugins.credentials.SystemCre
12
13 def artifactoryPassword = new File("/run/secrets/artifactoryPassword").text.trim()
14
15 def credentials=['username':'admin', 'password':artifactoryPassword, 'description':'Irtifi
16
17 def user = new UsernamePasswordCredentialsImpl(CredentialsScope.GLOBAL, 'artifactoryCred
18
19 store.addCredentials(domain, user)

```

As you can see, instead of having the password in the source code, we are reading the password from “/run/secrets/artifactoryPassword.”

Once credentials is created, we can refer to it from Config File Provider Plugin. Amend mvn_settings from the previous part, as follow:

```

1  import jenkins.model.*
2  import org.jenkinsci.plugins.configfiles.maven.*
3  import org.jenkinsci.plugins.configfiles.maven.security.*
4
5  def configStore = Jenkins.instance.getExtensionList('org.jenkinsci.plugins.configfiles.Glc
6
7  println("Setting maven settings xml")
8  def serverCreds = new ArrayList()
9
10
11 //server id as in your pom file
12 def serverId = 'artifactory'
13
14 //credentialId from credentials.groovy
15 def credentialId = 'artifactoryCredentials'
16
17 serverCredentialMapping = new ServerCredentialMapping(serverId, credentialId)

```

```

18 serverCreds.add(serverCredentialMapping)
19
20
21 def configId = 'our_settings'
22 def configName = 'myMavenConfig for jenkins automation example'
23 def configComment = 'Global Maven Settings'
24 def configContent = '''<settings>
25 <!-- your maven settings goes here -->
26 </settings>'''
27
28
29 def globalConfig = new GlobalMavenSettingsConfig(configId, configName, configComment, configContent)
30 configStore.save(globalConfig)
31
32 println("maven settings complete")

```

You may notice we deleted the server section with the password from settings; this is because, in combination with the credentials plugin information, we can now generate it on the fly and apply it to settings. Cool, isn't it?

Now let's rebuild our image. Remember the warning from docker-compose?

```
1 docker-compose up --build
```

Check that the credentials are created:

T	P	Store	Domain	ID	Name
		Jenkins	(global)	artifactoryCredentials	admin/***** (Artifactory OSS Credentials)

Icon: S M L

Stores scoped to Jenkins

P	Store	Domains
	Jenkins	(global)

And settings now doesn't have any sensitive information:


 Config Files

 Add a new Config

Edit Configuration File

description

The configuration

ID	our_settings
Name	myMavenConfig for jenkins automation example
Comment	Global Maven Settings
Replace All	<input checked="" type="checkbox"/>
Server Credentials	ServerId: artifactory
Credentials	admin/***** (Irtifactory OSS Credentials) 

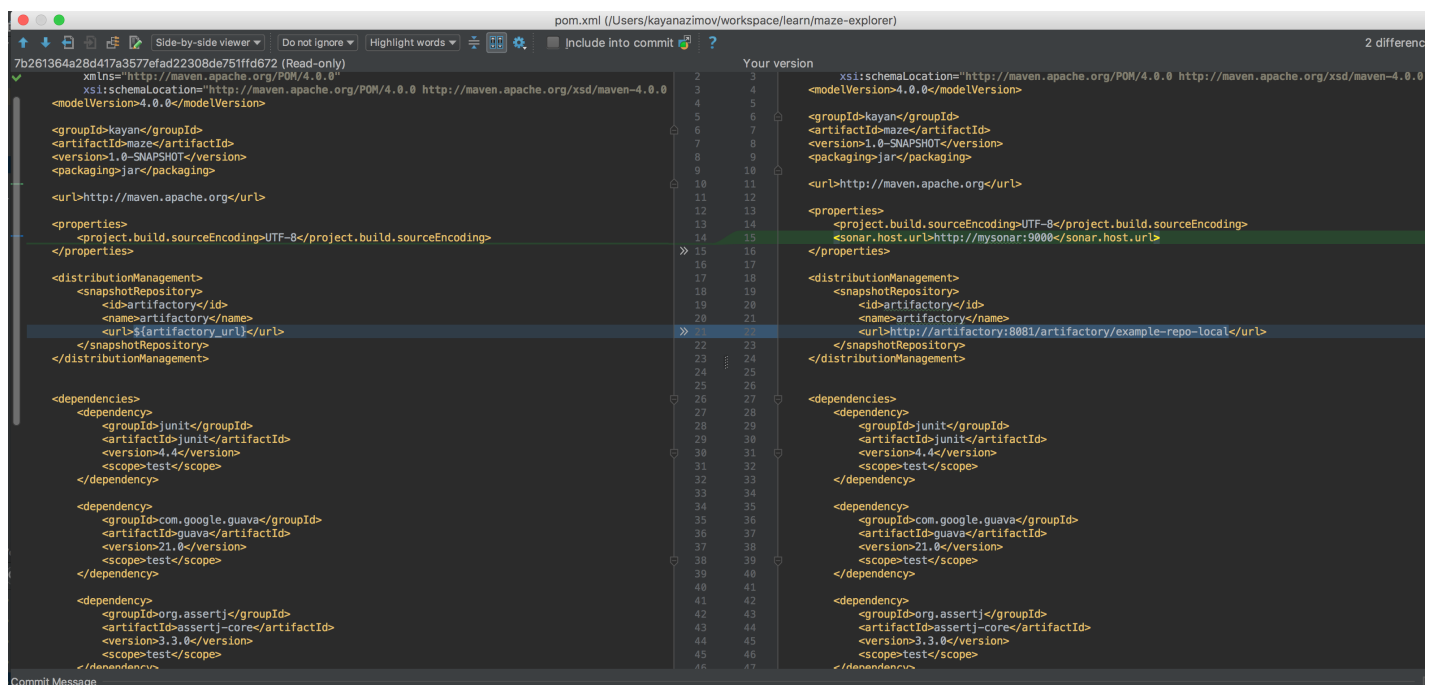
Add

Content

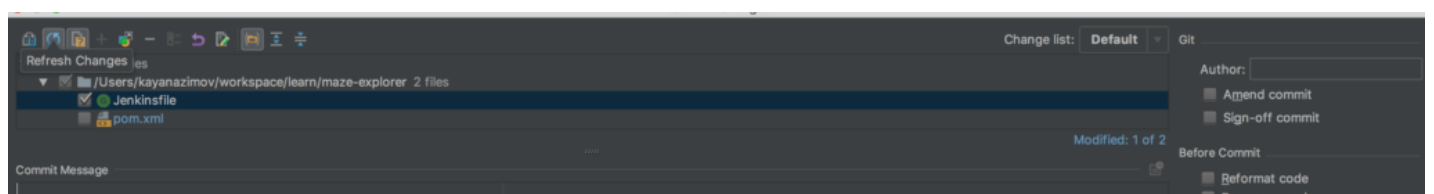
```
1 <settings>
2 <!-- your maven settings goes here -->
3 </settings>
```

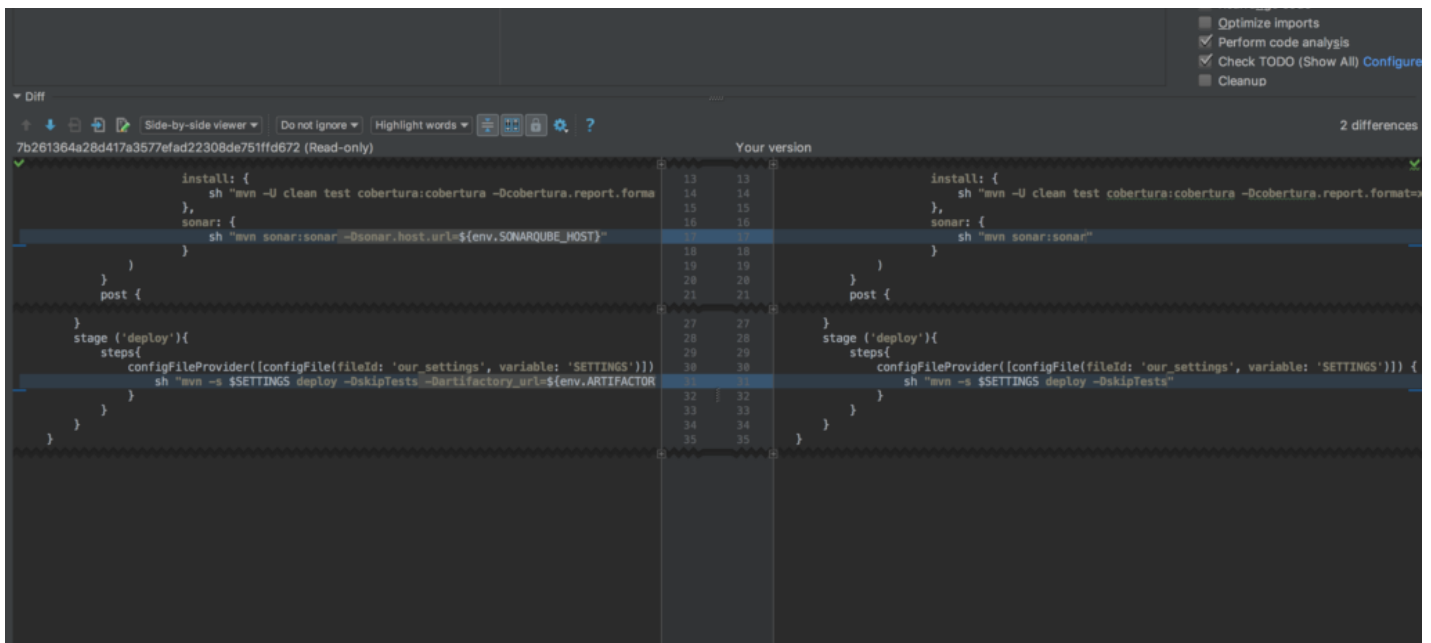
You can make the security guys at your company happy now!

Before running our pipeline, remember that “links” in the docker-compose file has removed the necessity of having IP addresses written in the pom file. You can now remove IP addresses from the pom file in the project:

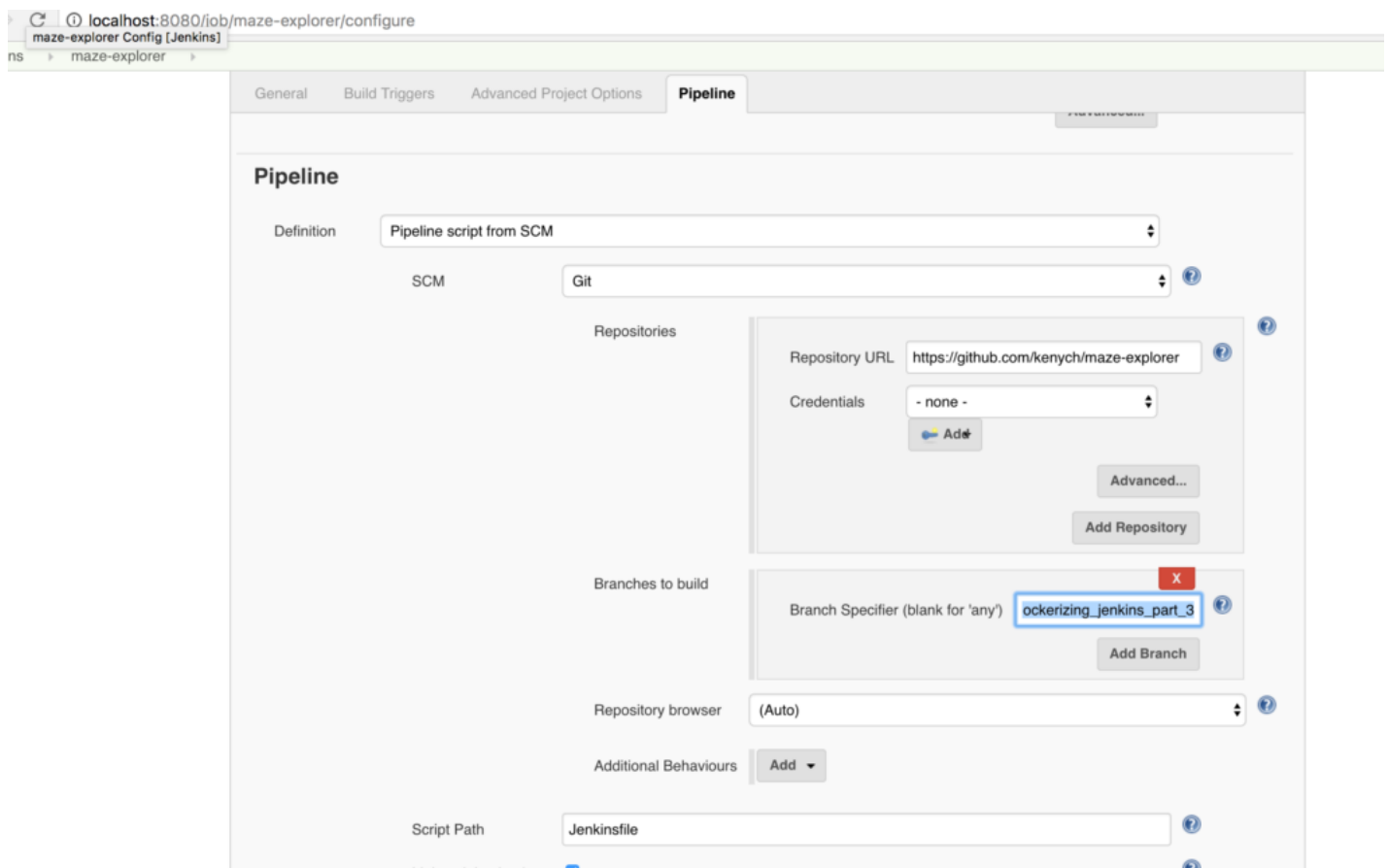


The same is true for the pipeline:





If you are using the code from Part 2, make sure you switch to the “*/dockerizing_jenkins_part_3” branch in the Git repo of the pipeline for maze-explorer project, which has the pom and pipeline changes in it:



And now we have more tidy pipeline:

```

1 pipeline {
2     agent any
3
4     tools {
5         /**      Uncomment if want to have specific java versions installed, otherwise maven tool

```



```

*      you will also need to uncomment java related stuff in java.groovy from dockerize
6
*      in your download folder
7
*/
8
//      jdk 'jdk8'
9
      maven 'maven3'
10
    }
11
12
    stages {
13
        stage('install and sonar parallel') {
14
            steps {
15
                parallel(
16
                    install: {
17
                        sh "mvn -U clean test cobertura:cobertura -Dcobertura.report.f
18
                    },
19
                    sonar: {
20
                        sh "mvn sonar:sonar"
21
                    }
22
                )
23
            }
24
            publicar {
25
                siempre {
26
                    junit '** / target / * - reports / TEST - *. xml'
27
                    step ([ $ class: 'CoberturaPublisher' , coberturaReportFile: 'target
28
                }
29
            }
30
        }
31
    }
32
    stage ( 'deploy' ) {
33
        pasos {
34
            configFileProvider ([ configFile ( fileId: 'our_settings' , variable: 'S
35
            sh "mvn -s $ SETTINGS deploy -DskipTests"
36
        }
37
    }
38
}
39
40
}

```

¡Ejecuta la construcción y cruza los dedos para una construcción verde y exitosa!

Eso es todo, ahora ha implementado otra función genial y ha llevado la dockerización de Jenkins al siguiente nivel más seguro.

Todos los pasos están codificados en el repositorio a continuación; puede verificar y ejecutar todo con un solo comando:

```

1  git clone https://github.com/kenych/dockerizing-jenkins && \
-  cd dockerizing-jenkins && \

```

```
2 cd dockerizing-jenkins && \
3 git checkout dockerizing_jenkins_part_3_docker_component_docker_secret_credentials_plug
4 ./runall.sh
```

Obtenga más información sobre cómo Scalyr creó una base de datos propietaria que no usa la indexación de texto para su herramienta de administración de registros.

Me gusta este artículo? Leer más de DZone



Templating Jenkins Build Environments con Docker Containers



¿Están sus contenedores seguros? Una entrevista con Aqua Security



DevSecOps y GDPR: por qué la gestión de riesgos de código abierto nunca ha sido más importante



Gratis DZone Refcard Comenzando con Docker

Temas: DOCKER, JENKINS, CONTENEDORES, DEVOPS, SEGURIDAD DE CONTENEDORES

Las opiniones expresadas por los contribuidores de DZone son suyas.

Obtenga lo mejor de DevOps en su bandeja de entrada.

Manténgase actualizado con el boletín DevOps quincenal de DZone. [VER UN EJEMPLO](#)

SUSCRIBIR

Recursos para socios de DevOps

Continúe su transformación digital con un Blueprint para entrega continua.

Automic



Automatice la seguridad en su oleoducto DevOps

Sonatype



4 Ways to Improve Your DevOps Testing

xMatters



Jenkins, Docker and DevOps: The Innovation Catalysts

CloudBees



