



# Dockerizing Jenkins 2, Parte 1: Canal de compilación declarativo con SonarQube Analysis

por Kayan Azimov · 27 y 17 de diciembre · DevOps Zone

Descargue el plan que puede llevar a una compañía de cualquier nivel de madurez hasta la entrega continua a escala empresarial utilizando una combinación de Atomic Release Automation, los más de 20 años de experiencia en automatización de negocios de Atomic y las herramientas y prácticas comprobadas que la compañía ya está aprovechando .

---

En este artículo, voy a demostrar:

- Ejecutando a Jenkins en Docker.
- Automatización de la instalación del complemento Jenkins en Docker.
- Configurar las herramientas de Java y Maven en Jenkins, primero manualmente y luego a través de los scripts de Groovy.
- Automatizando el paso anterior con Docker.
- Ejecutando SonarQube en Docker.
- Configuración de una interconexión Java Maven con pruebas unitarias, cobertura de prueba y pasos de análisis de SonarQube.

La próxima vez, en la Parte 2, voy a demostrar todo lo que necesita para la implementación:

- Cómo ejecutar un repositorio de Artifactory en Docker.
- Cómo configurar un archivo POM para la implementación.
- Cómo configurar las configuraciones de Maven para la implementación.
- Usando el plugin de despliegue Maven.

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

Al final, debería poder ejecutar la tubería en un contenedor Jenkins Docker completamente automatizado.

Tenga en cuenta que parte del código del artículo no funcionará como lo está en el futuro, cosas como la imagen base de jenkins y la lista de complementos, el URI para descargar maven y java cambiarán y podrían romper el código, pero el código proporcionado en el repositorio de git debería funcionar ya que lo actualizo basado en los comentarios de los revisores.

Como ya sabrá, con Jenkins 2, puede tener su canalización de compilación directamente dentro de su proyecto Java, por lo que puede usar su propio proyecto Maven Java para seguir los pasos de este artículo, siempre que esté alojado en un Repositorio de Git

Obviamente, todo se ejecutará en Docker, ya que es la forma más fácil de implementarlos y ejecutarlos.

Entonces, veamos cómo ejecutar Jenkins en Docker:

```
1 docker pull jenkins: 2.60.1
```

Mientras se descarga en segundo plano, veamos qué vamos a hacer con él una vez que esté listo.

Jenkins por defecto aparece completamente desnudo y muestra un asistente de instalación de plugins sugerido. Lo elegiremos, luego capturaremos todos los complementos instalados y luego automatizaremos este paso manual en una imagen Docker y seguiremos esta sencilla regla en todos los pasos:

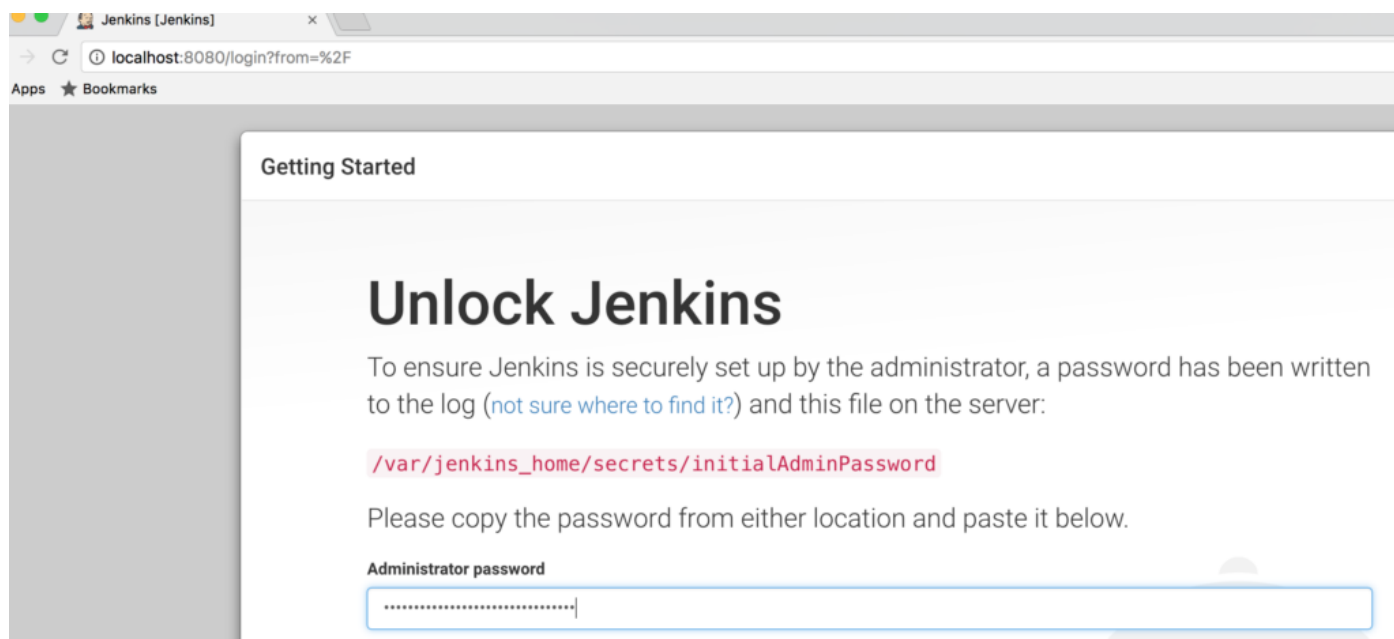
1. Configurar manualmente.
2. Configurar programáticamente.
3. Automatiza con Docker.

```
1  estibador ejecutar -p 8080 : 8080 --rm --name myjenkins Jenkins: 2.60.1
```

Tenga en cuenta que utilicé una etiqueta específica; No estoy usando la última etiqueta, que es la predeterminada si no especifica una, ya que no quiero que se rompa nada en el futuro.

También tenga en cuenta que nombramos el contenedor por lo que es más fácil referirse a él más tarde, ya que de lo contrario, Docker lo nombrará al azar, y añadimos el indicador `-rm` para eliminar el contenedor una vez que lo *detengamos* ; esto asegurará que estamos ejecutando Jenkins de forma inmutable y todo se configura sobre la marcha, y si queremos preservar cualquier dato, lo haremos de forma explícita.

Si recibe un error que *indica* que el puerto `8080` está ocupado, simplemente elija uno que sea gratuito para la primera parte de `8080 : 8080` . El primero es el puerto de host y el segundo es el puerto donde Jenkins correrá dentro del contenedor. Vamos a abrir *localhost: 8080* (o lo que sea que haya elegido como puerto de host). Debería ver la siguiente captura de pantalla para ingresar una contraseña de administrador:



```
*****
*****
*****
Jul 03, 2017 8:17:17 PM hudson.model.UpdateSite updateData
INFO: Obtained the latest update center data file for UpdateSource default
Jul 03, 2017 8:17:18 PM hudson.model.DownloadService$Downloadable load
INFO: Obtained the updated data file for hudson.tasks.Maven.MavenInstaller
Jul 03, 2017 8:17:20 PM hudson.model.DownloadService$Downloadable load
```

Debería ver un asistente de instalación de complementos ahora:

#### Getting Started

## Customize Jenkins

Plugins extend Jenkins with additional features to support many different needs.

### Install suggested plugins

Install plugins the Jenkins community finds most useful.

### Select plugins to install

Select and install plugins most suitable for your needs.

Elija la opción de **complementos sugeridos** y Jenkins comenzará la instalación de complementos. Una vez hecho esto, debemos capturar los complementos instalados y luego automatizarlos en Docker.

Abra `http://localhost:8080/script` y pegue este script Groovy y ejecútelo:

```
1 Jenkins . instancia . pluginManager . plugins . cada {
```

```

pipeline-build-step
pipeline-model-extensions
pipeline-stage-view
junit
docker-workflow
build-timeout
pipeline-stage-tags-metadata
branch-api
token-macro
Result: [Plugin:matrix-project, Plugin:git-client, Plugin:workflow-st
Plugin:cobertura, Plugin:idevops-data]

```

Ahora, copie los complementos que se muestran en la lista, es decir, todo el texto de la parte superior, hasta la sección "Resultado:" y cree un archivo plugins.txt y pegue allí el texto copiado.

Hay dos formas alternativas de obtener los complementos instalados:

Gracias a este método :

```

1 curl "http://192.168.1.7:8080/pluginManager/api/json?depth=1" | jq -r '.plugins []. shortName'

```

O una forma aún más extrema que se me ocurrió, desde el contenedor:

```

1 docker exec -it myjenkins ls / var / jenkins_home / plugins / | grep -v jpi

```

También puedes usar CLI; primero, descárguelo de <http://localhost:8080/jnlpJars/jenkins-cli.jar>, luego ejecute:

```

1 java -jar jenkins-cli.jar -s http://localhost:8080/ list-plugins

```

He escogido deliberadamente la versión de script de Groovy, ya que necesitará este **script** bastante para probar diferentes guiones que usaremos para configurar Jenkins programáticamente.

Detengamos el contenedor y automaticemos este paso:

```
11 COPY plugins.txt /usr/share/jenkins/ref/plugins.txt
12 EJECUTAR /usr/local/bin/install-plugins.sh </usr/share/jenkins/ref/plugins.txt
```

Asegúrese de tener una nueva línea vacía al final del archivo plugins.txt, ya que de lo contrario, Jenkins se quejará durante la instalación de los complementos. Tiempo para construir nuestra imagen:


```
1 estibador construir -t myjenkins.
```

Presta atención al punto al final, no te lo pierdas. Va a tomar un tiempo ya que ahora la imagen de Jenkins tiene instrucciones para descargar e instalar los complementos que definimos en plugins.txt. Una vez que esté listo, podemos ejecutarlo:

```
1 estibador ejecutar -p 8080 : 8080 --rm --name myjenkins myjenkins: últimas
```


Tenga en cuenta que esta vez estamos ejecutando el contenedor no desde la imagen predeterminada, sino la que acabamos de hornear. Puede verificar los complementos instalados ejecutando el mismo script que nosotros o solo en la interfaz de usuario:


← → ↻ 🏠 ⓘ localhost:8080/pluginManager/installed



# Jenkins

Jenkins > Plugin Manager

 [Back to Dashboard](#)

 [Manage Jenkins](#)

Updates Available **Installed** Advanced

Enabled

Name ↓

☒

[Ant Plugin](#)  
Adds Apache Ant support to Jenkins

☒

[Authentication Tokens API Plugin](#)  
This plugin provides an API for converting credentials into authentication tokens in Jenkins.

☒

[bouncycastle API Plugin](#)  
This plugin provides an stable API to Bouncy Castle related tasks.

☒

[Branch API Plugin](#)  
This plugin provides an API for multiple branch based projects.

☐

[build timeout plugin](#)

debemos tener Java instalado, no lo vamos a usar. En su lugar, deberíamos poder ejecutar nuestro proyecto con cualquier versión de Java. Pero primero, necesitamos tener todas las herramientas necesarias a mano. Vamos a descargar un par de versiones de Java. Encontré un lugar donde puedes encontrar cualquier versión de JDK; si no funciona, puedes buscarlo en Google.

```
1 curl -O http://ftp.osuosl.org/pub/funtoo/distfiles/oracle-java/jdk-8u131-linux-x64.tar.gz
2 curl -O http://ftp.osuosl.org/pub/funtoo/distfiles/oracle-java/jdk-7u76-linux-x64.tar.gz
```

Y luego descarga Maven:

```
1 curl -O http://apache.mirror.anlx.net/maven/maven-3/3.5.0/binaries/apache-maven-3.5.0-bin
```

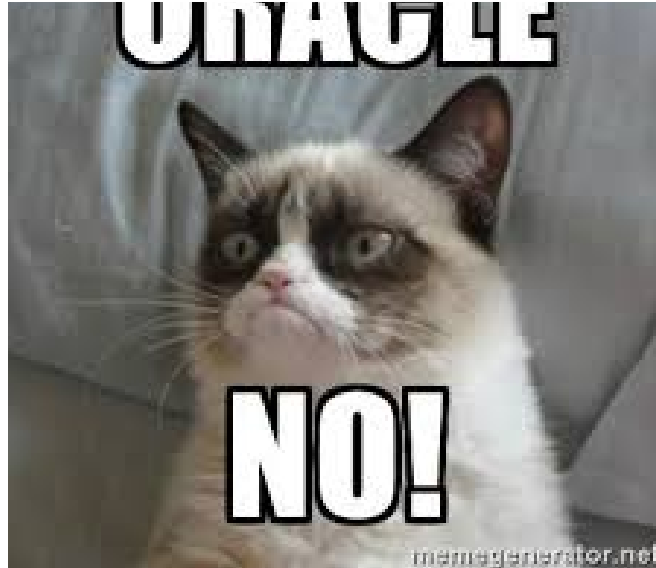
Crea una carpeta de *descargas* y cópiala allí. Su directorio de *descargas* debería verse así:

```
1 → my_jenkins tree descargas
2 descargas
3 ├── apache-maven-3.5.0-bin.tar.gz
4 ├── jdk-7u76-linux-x64.tar.gz
5 └── jdk-8u131-linux-x64.tar.gz
6
7 0 directorios, 3 archivos
8 → my_jenkins
```

Una vez hecho esto, estamos listos para ejecutar el contenedor nuevamente. Esta vez monté un directorio de host "*descargas*" como un volumen dentro del contenedor, por lo que puede acceder a los archivos en la carpeta de host para descargar Maven y Java más tarde:

```
1 estibador ejecutar -p 8080 : 8080 -v `pwd` / descargas: / var / jenkins_home / descarga:
```

Asegurémonos de que podamos ver el contenido de la carpeta desde el contenedor:



su/configuretools/

ion

## Global Tool Configuration

### Maven Configuration

Default settings provider

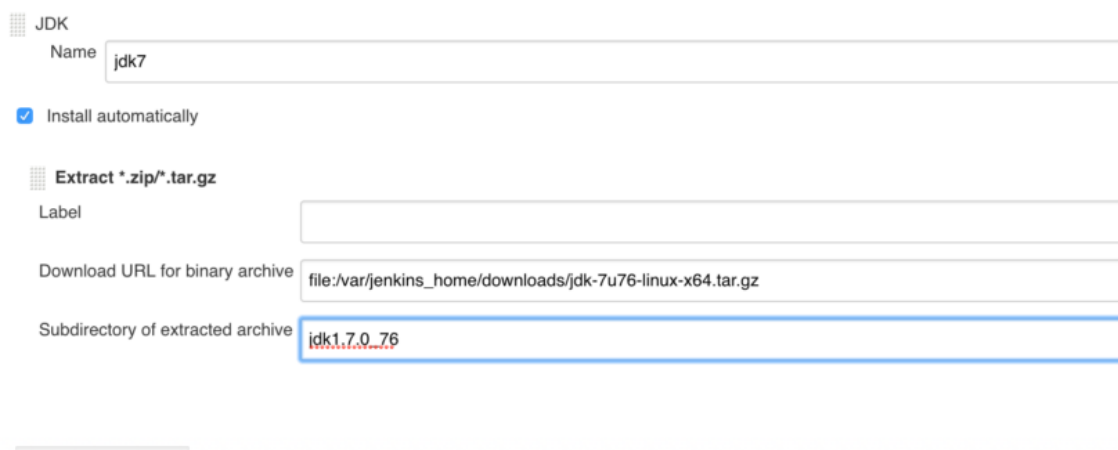
Default global settings provider

### JDK

JDK installations

JDK
<div>Name <input type="text"/></div> <div><span>Required</span></div> <div><input checked="" type="checkbox"/> Install automatically <span>?</span></div> <div><div><div>Install from <a href="http://java.sun.com">java.sun.com</a></div><div>Version <input type="text" value="Java SE Development Kit 8u131"/></div><div><input type="checkbox"/> I agree to the Java SE Development Kit License Agreement</div><div><span>Installing JDK requires Oracle account. <a href="#">Please enter your username/password</a></span></div></div><div><div>Add Installer</div><div>Delete Installer</div><div>Delete JDK</div></div></div>





JDK

Name

☒ Install automatically

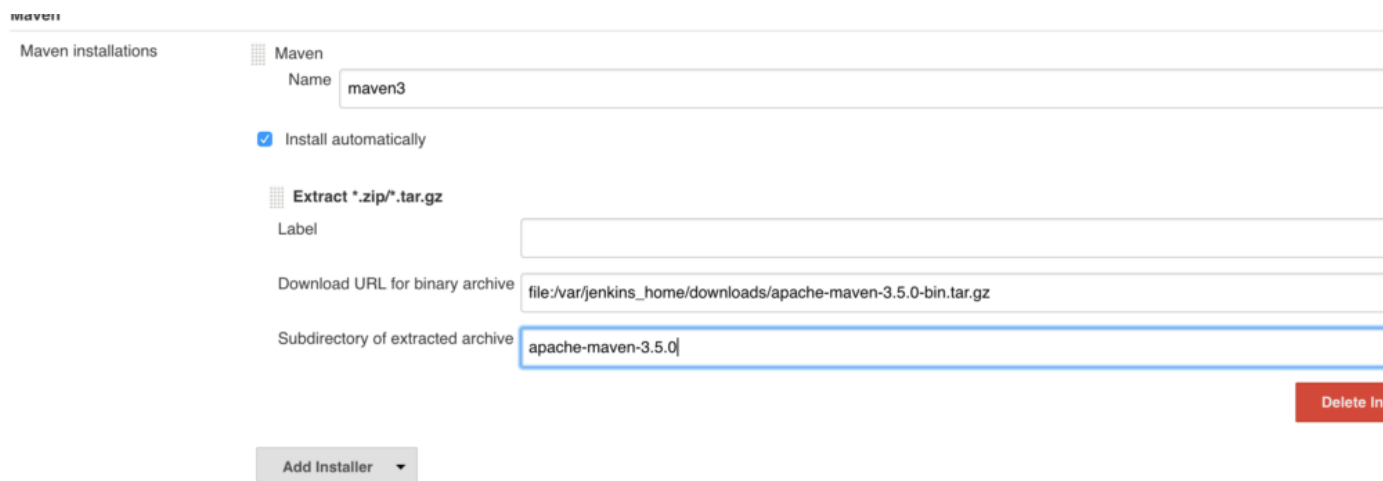
Extract \*.zip/\*.tar.gz

Label

Download URL for binary archive

Subdirectory of extracted archive

Entonces Maven:



maven

Maven installations

Maven

Name

☒ Install automatically

Extract \*.zip/\*.tar.gz

Label

Download URL for binary archive

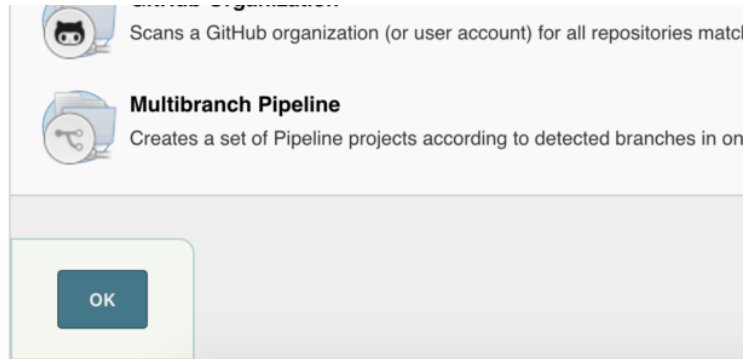
Subdirectory of extracted archive

Delete

Add Installer

Ahora podemos verificar si nuestras herramientas están configuradas correctamente dentro de Jenkins. Vamos a crear un trabajo y configurar su tipo a la tubería y comenzar a escribir nuestro código:

Enter an item name



```
1 tubería {
2     agente cualquier
3     herramientas {
4         jdk 'jdk7'
5         maven 'maven3'
6     }
7     etapas {
8         stage ( 'prueba de instalación de Java' ) {
9             pasos {
10                 sh 'java -version'
11                 sh 'which java'
12             }
13         }
14         stage ( 'instalación de prueba de prueba' ) {
15             pasos {
16                 sh 'mvn -version'
17                 sh 'which mvn'
18             }
19         }
20     }
21 }
```

```

2  importar hudson . herramientas . InstallSourceProperty
3  importar Hudson . herramientas . ZipExtractionInstaller
4
5  def descriptor = nuevo JDK . DescriptorImpl ();
6
7
8  def List < JDK > installations = []
9
10 javaTools = [[ 'nombre' : 'jdk8' , 'url' : 'archivo: /var/jenkins_home/downloads/jdk-8u131
11 [ 'nombre' : 'jdk7' , 'url' : 'archivo: /var/jenkins_home/downloads/jdk-7u76-linux->
12
13 javaTools . cada uno { javaTool ->
14
15     println ( "Herramienta de configuración: $ { javaTool . nombre }" )
16
17     def installer = new ZipExtractionInstaller ( javaTool . label como String , java
18     def jdk = nuevo JDK ( javaTool . name como String , null , [ new InstallSourceF
19     instalaciones . agregar ( jdk )
20
21 }
22 descriptor . setInstallations ( instalaciones . toArray ( nuevos JDK [ instalaciones . t
23 descriptor . guardar ()

```

### Y Maven:

```

1  importar Hudson . tareas . Maven
2  importar Hudson . tareas . Maven . MavenInstalación ;

```

```

18  def installer = new ZipExtractionInstaller ( mavenTool . label como String , mavenTool . url como String )
19
20
21  DescribableList . add ( nueva InstallSourceProperty ([ instalador ]))
22
23  instalaciones . add ( nueva MavenInstallation ( mavenTool . name como String , "" , descargar ) )
24
25  extensiones . setInstallations ( instalaciones . toArray ( nuevo MavenInstallation [ instalador ] ) )
26  extensiones . guardar ()

```

Verifica las herramientas; todo lo que creamos antes está allí de nuevo. ¡Hurra!

Probémoslo y asegúrese de que realmente funciona creando y ejecutando la misma línea de prueba. Oh, chasquido! Por supuesto, se ha ido, ya que eliminamos el contenedor cuando lo detengamos.

Así que asegurémonos de que la próxima vez que destruyamos el contenedor, los trabajos no se pierdan. Detenga el contenedor, nuevamente, cree trabajos de directorio en el directorio actual y ejecútelo con el siguiente comando:

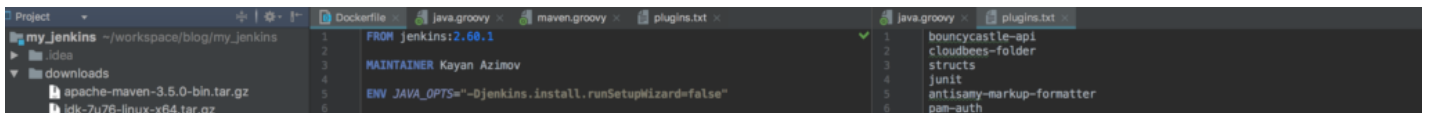
```

1  estibador ejecutar -p 8080 : 8080 -v `pwd` / descargas : / var / jenkins_home / descargas

```

Ahora estamos guardando todos los trabajos en el directorio de trabajos en el host, así que la próxima vez que destruya el contenedor, estará allí cuando lo ejecutemos nuevamente.

Hagamos Dockerize la automatización ahora. Cree archivos Groovy y colóquelos en la carpeta Groovy del proyecto.



El siguiente Dockerfile debe verse a continuación.

```
1  #esta es la imagen base que usamos para crear nuestra imagen desde
2  DE jenkins: 2.60.1
3
4  #just información sobre quién creó esto
5  MANTENEDOR Kayan Azimov (correo electrónico)
6
7  # deshacerse de la configuración de contraseña de administrador
8  ENV JAVA_OPTS = "-Djenkins.install.runSetupWizard = false"
9
10 #instalando automáticamente todos los complementos
11 COPY plugins.txt /usr/share/jenkins/ref/plugins.txt
12 EJECUTAR /usr/local/bin/install-plugins.sh </usr/share/jenkins/ref/plugins.txt
13
14 #agregación de scripts
15 COPY groovy / * /usr/share/jenkins/ref/init.groovy.d/
```

Hicimos esto porque cuando Jenkins comienza, ejecuta todos los scripts en un directorio llamado **init.groovy**.

Ahora estamos listos para construir una nueva imagen y ejecutarla:

```
1  → my_jenkins docker build -t myjenkins.
2
3  Enviar contexto de construcción a Docker daemon   336 .7MB
4  Paso 1 /6: a Jenkins: 2.60.1
5    - - > 0b4d4d677a26
6  Paso 2 /6: MAINTAINER Kayan Azimov
7    - - > Usando caché
8    - - > 67b933684219
9  Paso 3 /6: ENV JAVA_OPTS "-Djenkins.install.runSetupWizard = false"
```

1

Debería ver la instalación en los registros:

```

1 Jul 05 , 2017 8 : 27: 50 PM jenkins.util.groovy.GroovyHookScript ejecutar
2 INFORMACIÓN: Ejecutando /var/jenkins_home/init.groovy.d/java.groovy
3 Jul 05 , 2017 8 : 27: 50 PM hudson.model.AsyncPeriodicWork $ 1 plazo
4 INFO: Comenzó Descargar metadatos
5 Herramienta de configuración: jdk8
6 Herramienta de configuración: jdk7
7 Jul 05 , 2017 8 : 27: 50 PM jenkins.util.groovy.GroovyHookScript ejecutar
8 INFORMACIÓN: Ejecutar /var/jenkins_home/init.groovy.d/maven.groovy
9 Herramienta de configuración: maven3
10 Jul 05 , 2017 8 : 27: 51 PM jenkins.util.groovy.GroovyHookScript ejecutar
11 INFORMACIÓN: Ejecutando /var/jenkins_home/init.groovy.d/tcp-slave-agent-port.g

```

Verifique el conducto de prueba (debería estar allí esta vez) y ejecútelo.

Si todo está bien y nuestra línea de prueba está funcionando, es hora de crear una tubería para nuestro proyecto Java Maven.

Cree un nuevo trabajo de interconexión, pero esta vez, seleccione secuencia de comandos de secuencia de SCM y especifique la ruta a su proyecto de Git Maven.


## Pipeline

Definition

SCM

Repositories

Repository URL

 Please enter Git repository.

En consecuencia, siempre puedes actualizar un elemento en Jenkins en lugar de crear un nuevo archivo de Jenkins, simplemente haciendo clic en la flecha al lado del último trabajo y seleccionando Reproducir y actualizando el script:

The screenshot displays the Jenkins Pipeline interface. On the left, the 'Build History' panel shows a list of builds, with build #13 selected. A context menu is open over build #13, showing options like 'Changes', 'Console Output', 'Edit Build Information', 'Delete Build', 'Git Build Data', 'No Tags', 'Test Result', 'Replay' (highlighted), and 'Pipeline Steps'. Above this panel, there are buttons for 'Build Now', 'Delete Pipeline', 'Configure', 'Full Stage View', and 'Pipeline Syntax'. To the right, the 'Stage View' panel shows a timeline of stages. The 'Declarative: Checkout SCM' stage is highlighted with a green bar and a duration of 1s. Below it, the 'Average stage times' section shows a bar chart with a duration of 1s. The 'Stage View' panel also displays a list of builds with their timestamps and 'No Changes' status.

De lo contrario, crea un archivo llamado Jenkins en tu proyecto de Git con el contenido siguiente, inserta tus cambios y ejecuta la compilación:

```
1 tubería {
2     agente cualquier
```

```
1  tubería {
2      agente cualquier
3
4      herramientas {
5          jdk 'jdk8'
6          maven 'maven3'
7      }
8
9      etapas {
10         stage ( 'Instalar' ) {
11             pasos {
12                 sh "mvn prueba limpia"
13             }
14             publicar {
15                 siempre {
16                     junit '** / target / * - reports / TEST - *. xml'
17                 }
18             }
19         }
20     }
21 }
```

## Test Result

0 failures

40 tests

[Took 0.27 sec.](#)[add description](#)

## All Tests

Package	Duration	Fail	(diff) Skip	(diff) Pass	(diff) Total	(diff)
<a href="#">ken.kata.techtest</a>	0.26 sec	0	0	9 +9	9 +9	
<a href="#">ken.kata.techtest.clock</a> ▼	7 ms	0	0	29 +29	29 +29	
<a href="#">ken.kata.techtest.parser</a>	0 ms	0	0	2 +2	2 +2	



```

14         publicar {
15             siempre {
16                 junit '** / target / * - reports / TEST - *. xml'
17             }
18         }
19     }
20 }
21 }

```

En los registros de trabajo, debería ver esto ahora:

```

1  en net.sourceforge.cobertura.reporting.xml.XMLReportFormatStrategy.save (XMLReportFormatSt
2  en net.sourceforge.cobertura.reporting.NativeReport.export (NativeReport.java:31)
3  en net.sourceforge.cobertura.reporting.CompositeReport.export (CompositeReport.java:19
4  en net.sourceforge.cobertura.reporting.ReportMain.parseArgumentsAndReport (ReportMain.
5  en net.sourceforge.cobertura.reporting.ReportMain.generateReport (ReportMain.java:141)
6  en net.sourceforge.cobertura.reporting.ReportMain.main (ReportMain.java:151)
7
8  [INFO] La generación de Cobertura Report fue exitosa.
9  [INFO] - - - - -
10 [INFO] CONSTRUIR ÉXITO

```

Si revisa los registros de compilación, ignore algunos problemas de Cobertura si ejecuta un proyecto de Java 8, ya que Cobertura no es compatible con Java 8; Probablemente lo actualice para usar Jacoco en la próxima sesión.

Para poder ver los resultados de Cobertura en la página de Jenkins, necesita el complemento, vamos a instalarlo primero en `http://localhost:8080/pluginManager/advanced`.

```

7  Progreso (1): 7 .7 / 11 kB
8  Progreso (1): 10 /11 kB
9  Progreso (1): 11 kB
10
11 Descargada: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/24
12 Descargando: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/23/maven-p
13 Progreso (1): 2 .1 / 33 kB
14 Progreso (1): 4 .9 / 33 kB
15 Progreso (1): 7 .7 / 33 kB
16 Progreso (1): 10 /33 kB
17 Progreso (1): 13 /33 kB
18 Progreso (1): 16 /33 kB
19 Progreso (1): 19 /33 kB
20 Progreso (1): 21 /33 kB
21 Progreso (1): 24 /33 kB
22 Progreso (1): 27 /33 kB
23 Progreso (1): 30 /33 kB
24 Progreso (1): 32 /33 kB
25 Progreso (1): 33 kB

```

Eso es porque Maven resuelve las dependencias del proyecto y las guarda en **var / jenkins\_home / .m2 / repository** y si ejecuta el trabajo nuevamente, no las descargará de nuevo. Lo que pasa es que se pierde cuando se destruye el contenedor. Así que vamos a agregarlos a través del volumen tal como lo hicimos para los trabajos. Para hacerlo, simplemente ejecute el contenedor con el siguiente comando:

```

1  mkdir m2deps
   estibador ejecutar -p 8080 : 8080 -v `pwd` / descargas: / var / jenkins_home / descarg
2

```

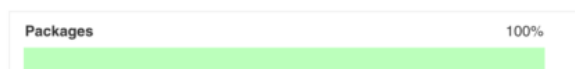
```
1  tubería {
2      agente cualquier
3
4      herramientas {
5          jdk 'jdk8'
6          maven 'maven3'
7      }
8
9      etapas {
10         stage ( 'Instalar' ) {
11             pasos {
12                 sh "mvn -U cobertura de prueba limpia: cobertura -Dcobertura.report.forma
13             }
14             publicar {
15                 siempre {
16                     junit '** / target / * - reports / TEST - *. xml'
17                     step ([ $ class: 'CoberturaPublisher' , coberturaReportFile: 'target
18                 }
19             }
20         }
21     }
22 }
```

Si ejecuta el trabajo ahora, debe obtener estos buenos informes:

### Code Coverage

#### Cobertura Coverage Report

Trend



Ahora es el momento para un análisis de SonarQube. Vamos a usar el plugin Maven Sonar. No voy a explicar aquí qué es SonarQube, pero en resumen, mantiene la pista de los problemas de calidad del código; puedes leer sobre esto aquí .

Tire y luego ejecute el contenedor:

```
1 docker pull sonarqube
```

Otra imagen grande, alrededor de 700M, así que es hora de un descanso ... y una vez que termine, ejecútelo:

```
1 estibador ejecutar -p 9000 : 9000 --rm --name mysonar sonarqube
```

Abra `http://localhost:9000` e inicie sesión como admin: admin y compruebe qué hay si tiene curiosidad; de lo contrario, detenga su contenedor Jenkins y vuelva a ejecutarlo con el servidor Sonar. Tendrá que usar la dirección IP del host y no el host local, ya que eso apuntará al host local del contenedor Docker. La forma más fácil de encontrar su IP (en Mac):

```
1 ~ ifconfig | grep "inet" | grep -v 127 .0.0.1
2
3     inet 192 .168.1.7 netmask 0xffffffff broadcast 192 .168.1.255
4 → ~
```

Ahora ejecute el contenedor Jenkins con la variable de entorno establecida:

```
1 estibador ejecutar -p 8080 : 8080 -v `pwd` / descargas: / var / jenkins_home / descarg
```

Una vez que Jenkins esté levantado, actualice la tubería con la etapa Sonar:

```
1 tubería {
2     agente cualquier
3
4     herramientas {
```

```
20  
21     stage ( 'Sonar' ) {  
22         pasos {  
23             sh "mvn sonar: sonar -Dsonar.host.url = $ { env . SONARQUBE_HOST }"  
24         }  
25     }  
26  
27 }  
28 }
```

Los registros de trabajo se verán así:

```
1  [INFO] Indicador de bloqueo del CPD del sensor (hecho) | tiempo = 82 ms  
2  [INFO] El proveedor de SCM para este proyecto es: git  
3  [INFO] 33 archivos para analizar  
4  
5  
6  [INFO] 33 /33 archivos analizaron  
7  [INFO] Cálculo de CPD para 4 archivos  
8  [INFO] Cálculo de CPD terminado  
9  
10  
11 [INFO] Informe de análisis generado en 1781ms, dir tamaño = 88 KB  
12  
13  
14 [INFO] Informes de análisis comprimidos en 8120 ms, tamaño de archivo comprimido = 72 KB  
15  
16  
17 [INFO] Informe de análisis cargado en 794 ms  
18 [INFO] ANALYSIS EXITOSO, puede navegar http://192.168.1.7:9000/dashboard/index/ken:berlin-  
19
```

Issues	Effort
<b>Type</b>	
Bug	0
Vulnerability	0
<b>Code Smell</b>	<b>23</b>
<b>Resolution</b>	
Unresolved	23
False Positive	0
Removed	0
<b>Severity</b>	
<b>Status</b>	
<b>Creation Date</b>	
<b>Rule</b>	
<b>Tag</b>	
<b>Module</b>	
<b>Directory</b>	
<b>File</b>	
<b>Assignee</b>	

**src/main/java/ken/kata/techtest/BerlinClock.java**

**Annotate the "BerlinClock" interface with the @FunctionalInterface annotation** ...

Code Smell Critical Open Not assigned 2min effort

**src/.../java/ken/kata/techtest/clock/TimeUnit.java**

**Annotate the "TimeUnit" interface with the @FunctionalInterface annotation** ...

Code Smell Critical Open Not assigned 2min effort

**src/.../java/ken/kata/techtest/clock/TimeUnitPart.java**

**Add the "@Override" annotation above this method signature** ...

Code Smell Major Open Not assigned 5min effort

**Remove the parentheses around the "index" parameter** ...

Code Smell Minor Open Not assigned 2min effort

**Replace this lambda with a method reference.** ...

Code Smell Minor Open Not assigned 2min effort

**Remove the parentheses around the "lamp" parameter** ...

Code Smell Minor Open Not assigned 2min effort

Finalmente, nuestra tubería está lista:

localhost:8080/job/berlin-clock/

Jenkins > berlin-clock

Back to Dashboard  
Status  
Changes  
Build Now  
Delete Pipeline  
Configure  
Full Stage View  
Coverage Report  
Pipeline Syntax

### Pipeline berlin-clock

Recent Changes

Stage View

Average stage times:

Declarative: Checkout SCM	Declarative: Tool Install	Install	Sonar
1s	5s	24s	28s

Build History

Test Result Trend

Code Coverage


(just show failures) enlarge

```
9      etapas {
10          stage ( 'instalar y sonar paralelo' ) {
11              pasos {
12                  paralelo ( instalar: {
13                      sh "mvn -U cobertura de prueba limpia: cobertura -Dcobertura.report.f
14                      }, sonar: {
15                      sh "mvn sonar: sonar -Dsonar.host.url = $ { env . SONARQUBE_HOST }"
16                  })
17              }
18              publicar {
19                  siempre {
20                      junit '** / target / * - reports / TEST - *. xml'
21                      step ([ $ class: 'CoberturaPublisher' , coberturaReportFile: 'target
22                  }
23              }
24          }
25      }
26  }
```

Eso es todo, ahora tiene una imagen Jenkins Docker completamente automatizada para esta línea de construcción con algunas etapas importantes. La próxima vez, mostraré cómo implementar un proyecto Maven en el repositorio de artefactos y eso incluirá algunos pasos interesantes, incluyendo un par de complementos de Jenkins y la automatización de la configuración de los complementos.

Finalmente, si fuiste demasiado perezoso para seguir los pasos, simplemente puedes ejecutar el siguiente comando; no requiere ninguna configuración aparte de tener instalado Docker. Espero que hayas encontrado este artículo útil, y si lo hiciste, sigue las actualizaciones mías ya que la parte de implementación es más interesante.

Temas: DOCKER, JENKINS, SONARQUBE, TUBERÍA COMO CÓDIGO, JENKINS CI, ARTEFACTORY, CONSTRUCCIÓN PARALELA

Publicado en DZone con el permiso de Kayan Azimov. [Vea el artículo original aquí.](#)   
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

## DevOps Partner Resources

Continue your digital transformation with a Blueprint for Continuous Delivery.

Automic



Automate Security in Your DevOps Pipeline

Sonatype



4 Ways to Improve Your DevOps Testing

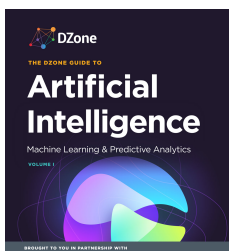
xMatters



Jenkins, Docker and DevOps: The Innovation Cata



CloudBees



## La Guía de Inteligencia Artificial 2017: Aprendizaje automático y análisis predictivo

- Descubra patrones en Analytics usando el poder del aprendizaje automático
- Aprenda sobre las redes neuronales usando las bibliotecas de Java
- Vea cómo el proyecto de código abierto de Google puede enriquecer las aplicaciones empresariales

**Descargar My Free PDF**