Práctica de laboratorio 6b: Construya una canalización CI/CD usando Jenkins

Versión en inglés:

https://ccna7.org/6-3-6-lab-build-a-ci-cd-pipeline-using-jenkins-answers/

Objetivos

Parte 1: Iniciar la Máquina Virtual (VM) DEVASC

Parte 2: Confirmar la aplicación de muestra a Git

Parte 3: Modificar la aplicación de muestra y enviar cambios a Git

Parte 4: Descargar y ejecutar la imagen de Jenkins Docker

Parte 5: Configurar Jenkins

Parte 6: Usar Jenkins para ejecutar una compilación de su aplicación

Parte 7: Usar Jenkins para probar una compilación

Parte 8: Crear una tubería o canalización en Jenkins

Aspectos básicos/Situación

En este laboratorio, se confirmará el código de la aplicación de muestra en un repositorio de GitHub, modificará el código localmente y, a continuación, confirmará los cambios. Igualmente, se instalará un contenedor Docker que incluye la última versión de Jenkins. Configure Jenkins y luego use Jenkins para descargar y ejecutar su programa de aplicación de muestra. Además, se creará un trabajo de prueba dentro de Jenkins que verificará que el programa de aplicación de muestra se ejecuta correctamente cada vez que lo compile. Finalmente, integrará su aplicación de ejemplo y el trabajo de prueba en una canalización de integración continua/desarrollo constante que verificará que su aplicación de muestra está lista para implementarse cada vez que cambie el código.

Recursos necesarios

- 1 Computadora con sistema operativo de su elección
- Virtual Box o VMWare
- Máquina virtual (Virtual Machine) DEVASC

Instrucciones

Parte 1: Iniciar la Máquina virtual (Virtual Machine) de DEVASC

Si no se ha completado el laboratorio **- Instale el Entorno de Laboratorio de la Máquina Virtual**, hacerlo ahora. Si se ha completado ya, iniciar la máquina virtual DEVASC.

Parte 2: Confirmar la aplicación de muestra a Git

En esta parte, creará un repositorio de GitHub para confirmar los archivos de aplicación de muestra que creó en un laboratorio anterior. Se creó una cuenta de GitHub en un laboratorio anterior. Si aún no lo ha hecho, visite github.com ahora y cree una cuenta.

Paso 1: Iniciar sesión en GitHub y cree un nuevo repositorio.

a. Iniciar sesión en https://github.com/ con sus credenciales.

b. Seleccionar el botón "**Nuevo repositorio**" o haga click en el ícono "**+**" en la esquina superior derecha y seleccione "**Nuevo repositorio**".

c. Crear un repositorio usando la siguiente información:

Nombre del repositorio: **sample-app**

Descripción: **Explorar CI/CD con GitHub y Jenkins**

Público/Privado: **Privado**

d. Seleccionar: **Crear repositorio**

Paso 2: Configurar sus credenciales de Git localmente en la máquina virtual (VM).

Abrir una ventana de terminal **con VS Code** en la máquina virtual de DEVASC. Utilizar su nombre en lugar de "Usuario de ejemplo" para el nombre entre comillas "". Utilice @example .com para su dirección de correo electrónico.

```
devasc @labvm: ~$ git config —global user.name «Usuario de ejemplo»
devasc @labvm: ~$ git config —global user.email sample@example.com
```

Paso 3: Inicializar un directorio como el repositorio de Git.

Usar los archivos de aplicación de ejemplo que creó en un laboratorio anterior. Sin embargo, esos archivos también se almacenan para su comodidad en el directorio **/labs/devnet-src/jenkins/sample-app**. Navegue al directorio **jenkins/sample-app** e inicialice como un repositorio de Git.

```
devasc @labvm: ~$ cd labs/devnet-src/jenkins/sample-app/
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$ git init
Repositorio Git vacío inicializado en /home/devasc/labs/devnet-src/jenkins/sample-app/.git/
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$
```

Paso 4: Apuntar el repositorio Git a GitHub repository.

Utilizar el comando **git remote add** para agregar una URL de Git con un alias remoto de «origen» y apunte al repositorio recién creado en GitHub. Usando la URL del repositorio de Git que creó en el paso 1, solo debe reemplazar el ***github-username*** en el siguiente comando con su nombre de usuario de GitHub.

Nota: Su nombre de usuario de GitHub distingue entre mayúsculas y minúsculas.

```
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$ git remote add origin https://github.com/github-username/sample-app.git
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$
```

Paso 5: Almacenar, confirmar e insertar los archivos de aplicación de muestra en el repositorio de GitHub.

a. Use el comando **git add** para almacenar los archivos en el directorio **jenkins/sample-app**. Utilice el argumento asterisco (*) para almacenar todos los archivos en el directorio actual.

```
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$ git add *
```

b. Utilizar el comando git status para ver los archivos y directorios que están almacenados y listos para ser confirmados en su repositorio de GitHub.

```
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$ git status
En rama maestra

Aún no hay confirmaciones

Cambios que deben confirmarse:
  (use «git rm —cache <gile>...«para quitarlo de almacenamiento)
        nuevo archivo: sample-app.sh
        nuevo archivo: sample_app.py
        nuevo archivo: static/style.css
        nuevo archivo: templates/index.html

devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$
```

c. Utilice el comando **git commit** para confirmar los archivos almacenados y comenzar a rastrear los cambios. Agregar un mensaje de su elección o utilizar el que se proporciona aquí.

```
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$ git commit -m
"Confirmando archivos de aplicación de ejemplo."
[master 4030ab6] Confirmar archivos de aplicación de muestra
 4 archivos cambiados, 46 inserciones (+)
 modo de creación 100644 sample-app.sh
 modo de creación 100644 sample_app.py
 modo de creación 100644 static/style.css
 modo de creación 100644 templates/index.html
```

d. Usar el comando **git push** para enviar sus archivos locales de aplicación de muestra a su repositorio de GitHub.

```
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$ git push origin master
Nombre de usuario para 'https://github.com': username
Contraseña para 'https://AllJohns@github.com': contraseña
Enumeración de objetos: 9, hecho.
Contando objetos: 100% (9/9), hecho.
Compresión Delta utilizando hasta 2 threads
Compresión de objetos: 100% (5/5), hecho.
Escribiendo objetos: 100% (8/8), 1,05 KiB | 1,05 Mib/s, hecho.
Total 8 (delta 0), reutilizado 0 (delta 0)
A https://github.com/AllJohns/sample-app.git
   d0ee14a.. 4030ab6 maestro -> maestro
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$
```

Nota: Si en lugar de una solicitud de su nombre de usuario, recibe un mensaje de VS Code con el mensaje, La extensión 'Git' quiere iniciar sesión usando GitHub, entonces configuró mal sus credenciales de GitHub en el paso 2 y/o el URL de GitHub en el paso 4. La dirección URL debe tener el nombre de usuario correcto que distingue entre mayúsculas y minúsculas y el nombre del repositorio que creó en el paso 1. Para revertir su comando anterior git add, use el comando **git remote rm origin**. A continuación, vuelva al paso 2 asegurándose de introducir las credenciales correctas y, en el paso 4, introduzca la dirección URL correcta.

Note: Si después de introducir el nombre de usuario y la contraseña, se obtiene un mensaje de error fatal (fatal error) indicando "repositorio no encontrado (repository not found)", probablemente una URL incorrecta fue enviada. Necesitará revertir su comando **git add** con el comando **git remote rm origin.**

Parte 3: Modificar la aplicación de ejemplo y enviar cambios a Git

En la Parte 4, instalar una imagen de Jenkins Docker que utilizará el puerto 8080. Recordar que los archivos de aplicación de ejemplo también especifican el puerto 8080. El servidor Flask y el servidor Jenkins no pueden usar el puerto 8080 al mismo tiempo.

En esta parte, cambiará el número de puerto utilizado por los archivos de aplicación de muestra, ejecutará la aplicación de muestra de nuevo para verificar que funciona en el nuevo puerto y, a continuación, enviará los cambios a su repositorio de GitHub.

Paso 1: Abrir los archivos de aplicación de ejemplo.

Asegúrese de que todavía está en el directorio **~/labs/devnet-src/jenkins/sample-app**, ya que estos son los archivos que están asociados con su repositorio de GitHub. Abra **sample_app.py** y **sample-app.sh** para su edición.

Paso 2: Editar los archivos de aplicación de muestra.

a. En sample_app.py, cambie la única instancia del puerto 8080 a 5050 como se muestra a continuación.

```
from flask import Flask
from flask import request
from flask import render_template

sample = Flask (__name__)

@sample .route ("/")
def main():
    return render_template («index.html»)

if __name__ == "__main__":
    sample.run (host="0.0.0.0", port=5050)
```

b. En sample-app.sh, cambiar las tres instancias del puerto 8080 a 5050 como se muestra a continuación.

```
#!/bin/bash

mkdir tempdir
mkdir tempdir/templates
mkdir tempdir/static

cp sample_app.py tempdir/.
cp -r templates /* tempdir/templates/.
cp -r static/* tempdir/static/.

echo "FROM python" >> tempdir/DockerFile
echo "RUN pip install flask" >> tempdir/DockerFile
echo "COPY ./static /home/myapp/static/ ">> tempDir/DockerFile
echo "COPY ./templates /home/myapp/templates/ ">> tempdir/DockerFile
echo "COPY sample_app.py /home/myapp/" >> tempdir/DockerFile
echo "EXPOSE 5050" >> tempdir/DockerFile
```

```
echo "CMD python3 /home/myapp/sample_app.py" >> tempdir/DockerFile

cd tempdir

docker build -t sampleapp.

docker run -t -d -p 5050:5050 —name samplerunning sampleapp
docker ps -a
```

Paso 3: Crear y verificar la aplicación de muestra.

a. Introduzca el comando **bash** para compilar su aplicación utilizando el nuevo puerto 5050.

```
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$ bash. /sample-app.sh
Sending build context to docker daemon 6.144kB
Step 1/7: FROM python
 ---> 4f7cd4269fa9
Step 2/7: RUN pip install flask
 —> Usando caché
 ---> 57a74c0dff93
Step 3/7: COPY ./static /home/myapp/static/
 —> Usando caché
 ---> e70310436097
Step 4/7: COPY ./templates /home/myapp/templates/
 —> Usando caché
 ---> e41ed6d0f933
Step 5/7: COPY sample_app.py /home/myapp/
 ---> 0a8d152f78fd
Paso 6/7: EXPOSE 5050
 —> Ejecutando en d68f6bfbcffb
Extrayendo contenedor intermedio d68f6bfbcffb
 ---> 04fa04a1c3d7
Step 7/7: CMD python3 /home/myapp/sample_app.py
 —> Ejecutando en ed48fdbc031b
Extrayendo contenedor intermedio ed48fdbc031b
 ---> ec9f34fa98fe
Ec9f34fa98fe compilado con éxito
Etiquetado con éxito sampleapp:latest
d957a4094c1781ccd7d86977908f5419a32c05a2a1591943bb44eeb8271c02dc
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
d957a4094c17 sampleapp «/bin/sh -c 'python...» Hace 1 segundo Hasta menos de un
segundo 0.0.0. 0:5050 ->5050/tcp samplerunning
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$
```

b. Abra una pestaña del navegador y navegue a localhost:5050. Debería verse el mensaje **Me estás llamando desde 172.17.0.1**

c. Apagar el servidor cuando haya verificado que funciona en el puerto 5050. Vuelva a la ventana de terminal donde se está ejecutando el servidor y presionar CTRL+C para detener el servidor.

Paso 4: Enviar sus cambios a GitHub.

a. Ahora todo está listo para enviar los cambios al repositorio de GitHub. Introduzca los siguientes comandos.

```
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$ git add *
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$ git status
En rama maestra
Cambios que deben confirmarse:
  (usar "git restore --staged <file>..." para quitar de almacenamiento)
        modificado: sample-app.sh
        modificado: sample_app.py
        nuevo archivo: TempDir/DockerFile
        nuevo archivo: tempdir/sample_app.py
        nuevo archivo: tempdir/static/style.css
        nuevo archivo: tempdir/templates/index.html

devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$ git commit -m "Se cambió
el puerto de 8080 a 5050."
[master 98d9b2f] Se ha cambiado el puerto de 8080 a 5050.
 6 archivos cambiados, 33 inserciones (+), 3 eliminaciones (-)
 modo de creacion 100644 TempDir/DockerFile
 modo de creacion 100644 tempdir/sample_app.py
 modo de creacion 100644 tempdir/static/style.css
 modo de creacion 100644 tempdir/templates/index.html
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$ git push origin master
Nombre de usuario para 'https://github.com': username
Contraseña para 'https://AllJohns@github.com': password
Enumerando objetos: 9, hecho.
Contando objetos: 100% (9/9), hecho.
Compresión Delta utilizando hasta 2 hilos
Comprimiendo objetos: 100% (6/6), hecho.
Escribiendo objetos: 100% (6/6), 748 bytes | 748,00 KIB/s, hecho.
Total 6 (delta 2), reutilizado 0 (delta 0)
remote: Resolviendo deltas: 100% (2/2), completado con 2 objetos locales.
A https://github.com/AllJohns/sample-app.git
   a6b6b83.. 98d9b2f maestro -> maestro
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$
```

b. Puede verificar que su repositorio de GitHub esté actualizado visitando **https://github.com/*github-user*/sample-app**. Debería ver un nuevo mensaje (puerto cambiado de 8080 a 5050.) y que se ha actualizado la última marca de tiempo de confirmación.

Parte 4: Descargar y ejecutar la imagen de Jenkins Docker

En esta parte, se descargará la imagen de Jenkins Docker. A continuación, inicie una instancia de la imagen y verifique que el servidor Jenkins se esté ejecutando.

Paso 1: Descargar la imagen de Jenkins Docker.

Puede encontrar la imagen de Jenkins Docker aquí: https://hub.docker.com/r/jenkins/jenkins. En el momento de escribir este laboratorio, ese sitio especifica usar el comando **docker pull jenkins/jenkins** para descargar el último contenedor Jenkins. Se debería obtener una salida similar a la siguiente:

```
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$ docker pull
jenkins/jenkins:lts
Está extrayendo de jenkins/jenkins
```

```
3192219afd04: Extrayendo de la capa fs
17c160265e75: Extrayendo de la capa fs
cc4fe40d0e61: Extrayendo de la capa fs
9d647f502a07: Extrayendo de la capa fs
d108b8c498aa: Extrayendo de la capa fs
1bfe918b8aa5: Extracción completa
dafa1a7c0751:Extracción completa
650a236d0150: Extracción completa
cba44e30780e: Extración completa
52e2f7d12a4d: Extracción completa
d642af5920ea: Extracción completa
e65796f9919e: Extracción completa
9138dabbc5cc: Extracción completa
f6289c08656c: Extracción completa
73d6b450f95c: Extracción completa
a8f96fbec6a5: Extracción completa
9b49ca1b4e3f: Extracción completa
d9c8f6503715: Extracción completa
20fe25b7b8af: Extracción completa
Digest: sha 256:717 dcbe5920753187a20ba43058ffd3d87647fa903d98cde64dda4f4c82c5c48
Estado: Imagen más nueva descargada para jenkins/jenkins:lts
docker.io/jenkins/jenkins:lts
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$
```

Paso 2: Arrancar el contenedor de Jenkins Docker.

Introduzca el siguiente comando en **una línea**. Es posible que tenga que copiarlo en un editor de texto si está viendo una versión PDF de este laboratorio para evitar saltos de línea. Este comando iniciará el contenedor de Docker de Jenkins y luego permitirá que los comandos de Docker se ejecuten dentro de su servidor Jenkins.

```
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$ docker run —rm -u root -
p 8080:8080 -v jenkins-data: /var/jenkins_home -v $ (que docker)
:/usr/bin/docker -v /var/run/docker.sock:/sovar/run/docker.ck -v «$HOME»
:/home —name jenkins_server jenkins/jenkins:lts
```

Las opciones utilizadas en este comando de **docker run** son las siguientes:

- **—rm** : esta opción elimina automáticamente el contenedor Docker cuando se detiene.
- **-u** - Esta opción especifica el usuario. Desea que este contenedor Docker se ejecute como raíz (root) para que se permitan todos los comandos Docker introducidos dentro del servidor Jenkins.
- **-p** - Esta opción especifica el puerto en el que se ejecutará el servidor Jenkins localmente.
- **-v** - Estas opciones unen los volúmenes de montaje necesarios para Jenkins y Docker. El primer **-v** especifica dónde se almacenarán los datos de Jenkins. El segundo **-v** especifica dónde ubicar Docker para que pueda ejecutarse dentro del contenedor Docker que está ejecutando el servidor Jenkins. El tercer **-v** especifica la variable PATH para el directorio principal.

Paso 3: Verificar que el servidor Jenkins se esté ejecutando.

El servidor Jenkins debería estar ahora en ejecución. Copiar la contraseña de administrador que aparece en la salida, como se muestra en la siguiente ventana.

No introduzca ningún comando en esta ventana del servidor. Si detiene accidentalmente el servidor Jenkins, deberá volver a ingresar el comando **docker run** del paso 2 anterior. Después de la instalación inicial, la contraseña de administrador se muestra como a continuación.

```
<output omitted>
*************************************************************
*************************************************************
*************************************************************

Se requiere la configuración inicial de Jenkins. Se ha creado un usuario administrador
y se ha generado una contraseña.
Utilice la siguiente contraseña para proceder a la instalación:

77dc402e31324c1b917f230af7bfebf2 <—Su contraseña será diferente

Esto también se puede encontrar en: /var/jenkins_home/secrets/initialAdminPassword

*************************************************************
*************************************************************
*************************************************************
<output omitted>
2020-05-12 16:34:29 .608+0000 [id=19] INFO Hudson.webappMain$3 #run: Jenkins está
completamente en funcionamiento
```

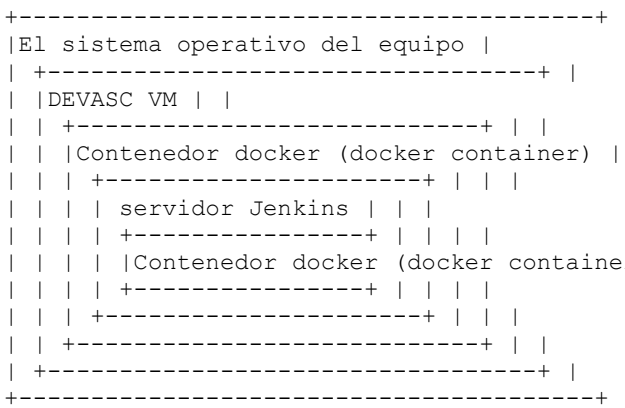
Nota: Si pierde la contraseña, no se muestra como se muestra arriba, o necesita reiniciar el servidor Jenkins, siempre puede recuperar la contraseña accediendo a la línea de comandos del contenedor Jenkins Docker. Cree una segunda ventana de terminal en VS Code e ingrese los siguientes comandos para que no detenga el servidor Jenkins.:

```
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$ docker exec -it
jenkins_server /bin/bash
root@19d2a847a54e: /# cat /var/jenkins_home/secrets/initialAdminPassword
77dc402e31324c1b917f230af7bfebf2
root @19d2a847a54e: /# exit
exit
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$
```

Nota: El ID del contenedor (19d2a847a54e resaltado anteriormente) y la contraseña serán diferentes.

Paso 4: Investigar los niveles de abstracción que se están ejecutando actualmente en el equipo.

El siguiente diagrama ASCII muestra los niveles de abstracción en esta implementación Docker-inside-Docker (dind). Este nivel de complejidad no es inusual en las redes e infraestructuras de nube actuales.



Parte 5: Configurar Jenkins

En esta parte, se completará la configuración inicial del servidor Jenkins.

Paso 1: Abrir una pestaña del navegador.

Vaya a http://localhost:8080/ e inicie sesión con su contraseña de administrador copiada.

Paso 2: Instalar los complementos de Jenkins recomendados.

Haga clic en **Instalar complementos sugeridos** y espere a que Jenkins descargue e instale los complementos. En la ventana del terminal, verá mensajes de registro a medida que la instalación continúe. Asegúrese de no cerrar esta ventana de terminal. Puede abrir otra ventana de terminal para acceder a la línea de comandos.

Paso 3: Omitir la creación de un nuevo usuario administrador.

Una vez finalizada la instalación, se le mostrará la ventana **Crear primer usuario administrador**. Por ahora, haga click en **Omitir y continúe como administrador** en la parte inferior.

Paso 4: Omitir la creación de una instancia.

En la ventana **Configuración de instancias**, no cambie nada. Haga clic en **Guardar y finalizar** en la parte inferior.

Paso 5: Empezar a usar Jenkins.

En la siguiente ventana, haga clic en **Empezar a usar Jenkins**. Ahora debería estar en el panel principal con un mensaje similar a **bienvenido a Jenkins!** .

Parte 6: Usar Jenkins para ejecutar una compilación de la aplicación

La unidad fundamental de Jenkins es el trabajo (también conocido como un proyecto). Puede crear trabajos que realicen una variedad de tareas, entre las que se incluyen las siguientes:

- Recuperar código de un repositorio de administración de código fuente como GitHub.
- Compilar una aplicación utilizando una secuencia de comandos (script) o una herramienta de compilación.
- Empaquetar una aplicación y ejecutarla en un servidor

En esta parte, creará un trabajo de Jenkins simple que recupera la última versión de su aplicación de muestra de GitHub y ejecuta el script de compilación. En Jenkins, puede probar su aplicación (Parte 7) y agregarla a una canalización de desarrollo (Parte 8).

Paso 1: Crear un nuevo trabajo.

a. Haga click en el enlace Crear un trabajo (Create a Job) directamente debajo del mensaje: ¡Bienvenido a Jenkins! También puede hacer clic en **Nuevo elemento (New Item)** en el menú de la izquierda.

b. En el campo **Escriba un nombre de elemento (Enter an Item Name)** ponga como nombre **BuildAppJob**.

c. Haga click en **Proyecto de estilo libre (Freestyle Project)** como tipo de trabajo. En la descripción, la abreviatura de ACS significa administración de configuración de software (Software Configuration Management, SCM), que es una clasificación de software responsable del seguimiento y control de los cambios en el software.

d. Arrástrelo hasta la parte inferior de la página y haga click en **OK**.

Paso 2: Configurar el Jenkins BuildAppJob.

Ahora se encuentra en la ventana de configuración donde se pueden introducir detalles sobre su trabajo. Las pestañas en la parte superior son accesos directos a las secciones siguientes. Hacer clic en las pestañas para explorar las opciones que se pueden configurar. Para este trabajo, sólo se necesita agregar algunos detalles de configuración.

a. Hacer clic en la pestaña **General** y agregar una descripción para su trabajo. Por ejemplo, "**Mi primer trabajo en Jenkins.**"

b. Hacer clic en la pestaña **Administración de código fuente (Source Code Management)** y elegir el botón de opción **Git**. En el campo URL del repositorio, agregue el enlace del repositorio de GitHub para la aplicación de muestra, teniendo en cuenta que su nombre de usuario distingue mayúsculas de minúsculas. Asegúrese de agregar la extensión .git al final de su URL.

   Por ejemplo:

   `https://github.com/github-username/sample-app.git`

c. Para **Credenciales (Credentials)**, haga clic en el botón **Agregar (Add)** y elija **Jenkins**.

d. En el cuadro de diálogo **Agregar credenciales (Add Credentials)**, rellene con su nombre de usuario y contraseña de GitHub y, a continuación, haga clic en **Agregar (Add)**.

   Nota: Recibirá un mensaje de error que indica que la conexión ha fallado. Esto se debe a que aún no ha seleccionado las credenciales.

e. En el menú desplegable de **Credenciales (Credentials)** donde actualmente dice **Ninguno (None)**, elija las credenciales que acaba de configurar.

f. Después de agregar la URL y las credenciales correctas, Jenkins prueba el acceso al repositorio. No debería haber mensajes de error. Si los hay, verificar la URL y las credenciales. Se tendrá que **añadirlos** de nuevo, ya que en este momento no hay forma de eliminar los que se han introducido anteriormente.

g. En la parte superior de la ventana de configuración de **BuildAppJob**, haga clic en la pestaña **Construir/Crear/Compilar (Build)**.

h. Para el menú desplegable **Agregar paso de compilación (Add build step)**, elija **Ejecutar shell (Execute shell)**.

i. En el campo **Comando (Command)**, escriba el comando que utiliza para ejecutar la compilación para el script sample-app.sh.

   ```
   Bash./sample-app.sh
   ```

j. Haga clic en el botón **Save** (Guardar). Será regresado al panel de Jenkins con **BuildAppJob** seleccionado.

Paso 3: Que Jenkins compile la aplicación.

En la parte izquierda, haga click en **Crear/compilar ahora (Build Now)** para iniciar el trabajo. Jenkins descargará su repositorio de Git y ejecutará el comando de compilación **bash./**sample-app.sh. Su compilación debería tener éxito porque no ha cambiado nada en el código desde la Parte 3 cuando modificó el código.

Paso 4: Acceder a los detalles de compilación.

A la izquierda, en la sección **Historial de compilación (Build History)**, haga click en su número de compilación, que debería ser el **#1**, a menos que haya compilado la aplicación varias veces.

Paso 5: Ver la salida de la consola.

A la izquierda, haga click en **Salida de consola (Console Output)**. Debe ver un resultado similar a lo siguiente. Observe los mensajes de éxito en la parte inferior, así como la salida del comando **docker ps -a**. Se están ejecutando dos contenedores docker: uno para su aplicación de muestra que se ejecuta en el puerto local 5050 y otro para Jenkins en el puerto local 8080.

```
Iniciado por el usuario admin
Ejecutar como SYSTEM
Construir en el espacio de trabajo /var/jenkins_home/workspace/buildAppJob
utilizar la credencial 0cf684ea-48a1-4e8b-ba24-b2fa1c5aa3df
Clonar el repositorio Git remoto
Clonar del repositorio https://github.com/github-user/sample-app
 > git init /var/jenkins_home/workspace/buildAppJob # timeout=10
Recuperar los cambios ascendentes de https://github.com/github-user/sample-app
 > git —version # timeout=10
usando GIT_ASKPASS para establecer credenciales
 > git fetch —tags —progress — https://github.com/github-user/sample-app
+refs/heads/*:refs/remotes/origin/* # timeout=10
 > git config remote.origin.url https://github.com/github-user/sample-app # timeout=10
 > git config —add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* #
timeout=10
 > git config remote.origin.url https://github.com/github-user/sample-app # timeout=10
Recuper los cambios ascendentes de https://github.com/github-user/sample-app
usar GIT_ASKPASS para establecer credenciales
 > git fetch —tags —progress — https://github.com/github-user/sample-app
+refs/heads/*:refs/remotes/origin/* # timeout=10
 > git rev-parse refs/remotes/origin/master^ {commit} # timeout=10
 > git rev-parse refs/remotes/origin/origin/master^ {commit} # timeout=10
Verificando revisión 230ca953ce83b5d6bdb8f99f11829e3a963028bf
(refs/remotos/origin/master)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f 230ca953ce83b5d6bdb8f99f11829e3a963028bf # timeout=10
Mensaje de confirmación: "Se han cambiado los números de puerto de 8080 a 5050"
 > git rev-list —no-walk 230ca953ce83b5d6bdb8f99f11829e3a963028bf # timeout=10
[BuildAppJob] $ /bin/sh -xe /tmp/jenkins1084219378602319752.sh
+ bash ./sample-app.sh
Enviando contexto de compilación a docker daemon 6.144kB

Paso 1/7: FROM python
 ---> 4f7cd4269fa9
Paso 2/7: RUN pip install flask
 —> Usando caché
 ---> 57a74c0dff93
Paso 3/7: COPY ./static /home/myapp/static/
 —> Usando caché
 ---> aee4eb712490
Paso 4/7: COPY ./templates /home/myapp/templates/
 —> Usando caché
 ---> 594cdc822490
Paso 5/7: COPY sample_app.py /home/myapp/
 —> Usando caché
 ---> a001df90cf0c
Paso 6/7: EXPOSE 5050
 —> Usando caché
 ---> eae896e0a98c
Paso 7/7: CMD python3 /home/myapp/sample_app.py
```

```
 —> Usando caché
 ---> 272c61fddb45
Compilado con éxito 272c61fddb45
Etiquetado con éxito sampleapp:latest
9c8594e62079c069baf9a88a75c13c8c55a3aeaddde6fd6ef54010953c2d3fbb
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
9c8594e62079 sampleapp «/bin/sh -c 'python...» Hace menos de un segundo Hasta menos de un segundo 0.0.0. 0:5050 ->5050/tcp samplerunning
e25f233f9363 jenkins/jenkins:lts «/sbin/tini — /usr/...» Hace 29 minutos Hasta 29 minutos 0.0.0. 0:8080 ->8080/tcp, 50000/tcp jenkins_server
Terminado: EXITOSO
```

Paso 6: Abra otra pestaña del navegador web y verifique que se esté ejecutando la aplicación de muestra.

Escriba la dirección local, **localhost:5050**. Debería ver el contenido de su index.html mostrado en color de fondo azul acero claro con **Usted me está llamando desde 172.17.0.1** mostrado como H1.

Parte 7: Usar Jenkins para probar una compilación

En esta parte, crearemos un segundo trabajo que pruebe la compilación para asegurarse de que funciona correctamente.

Nota: Debe detener y quitar el contenedor acoplador (docker container) de samplerunning.

```
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$ docker stop samplerunning
samplerunning
devasc @labvm: ~/labs/devnet-src/jenkins/sample-app$ docker rm samplerunning
samplerunning
```

Paso 1: Iniciar un nuevo trabajo para probar su aplicación de muestra.

a. Vuelva a la pestaña del navegador web Jenkins y haga clic en el enlace **Jenkins** en la esquina superior izquierda para volver al panel principal.

b. Haga click en el enlace **Nuevo Elemento (New Item)** para crear un nuevo trabajo.

c. En el campo **Escriba un nombre de elemento**, rellene con el nombre **TestAppJob**.

d. Haga clic en **Proyecto de estilo libre (Freestyle project)** como tipo de trabajo.

e. Arrastre hasta la parte inferior de la página y haga click en **OK**.

Paso 2: Configurar Jenkins TestAppJob.

a. Añadiremos una descripción para el trabajo. Por ejemplo, "Mi primera prueba de Jenkins."

b. Deje **Administración de código fuente (Source Code Management)** establecida en **Ninguno (None)**.

c. Haga clic en la pestaña **Generar/crear/compilar desencadenadores (Build triggers)** y marque la casilla **Crear/compilar/generar después de que se hayan creado otros proyectos**. Para **Proyectos para ver (Projects to watch)**, rellene con el nombre **BuildAppJob**.

Paso 3: Escribir el script de prueba que debe ejecutarse después de una compilación estable del BuildAppJob.

a. Haga clic en la pestaña **Compilar (Build)**.

b. Haga clic en **Agregar paso de compilación (Add build step)** y elija **Ejecutar shell (Execute shell)**.

c. Introducir el siguiente script. El comando **if** debe estar todo en una línea incluyendo **;then**. El comando **grep** verifica la salida devuelta desde el comando cURL para ver si es devuelto **Usted me llama desde 172.17.0.1**. Si es verdadera (if true), el script sale con un código de 0, lo que significa que no hay errores en la compilación **BuildAppJob**. Si es falso (if false), el script sale con un código de 1, lo que significa que el **BuildAppJob** falló.

```
if curl http://172.17.0.1:5050/ | grep «Me estás llamando desde 172.17.0.1"; then
   exit 0
else
   exit 1
fi
```

d. Haga clic en **Guardar** y, a continuación, en el enlace **Volver al panel** en el lado izquierdo.

Paso 4: Hacer que Jenkins que ejecute el trabajo BuildAppJob de nuevo.

a. Actualice la página web con el botón de actualizar de su navegador.

b. Ahora debería ver los dos trabajos enumerados en una tabla. Para el **trabajo BuildAppJob**, haga clic en el botón de compilación en el extremo derecho (un reloj con una flecha).

Paso 5: Verificar que ambos trabajos hayan finalizado.

Si todo va bien, deberíamos ver la marca de tiempo para la actualización de la columna **Último éxito** para **BuildAppJob** y **TestAppJob**. Esto significa que el código para ambos trabajos se ejecutó sin errores. Pero también podemos verificar esto por nosotros mismos.

Nota: Si las marcas de hora no se actualizan, asegúrese de activar la actualización automática haciendo clic en el vínculo situado en la esquina superior derecha.

a. Haga clic en el enlace para **TestAppJob**. Bajo **Enlaces permanentes (Permalinks)**, haga clic en el vínculo de la última compilación y, por consiguiente, haga clic en **Salida de consola (Console Uotput)**. Debe ver un resultado similar al siguiente:

```
Iniciado por el proyecto ascendente "BuildAppJob" número de compilación 13
originariamente causada por:
 Iniciado por el usuario admin
Ejecutando como SYSTEM
Construyendo en el espacio de trabajo /var/jenkins_home/workspace/testAppJob
[TestAppJob] $ /bin/sh -xe /tmp/jenkins1658055689664198619.sh
+ grep Me estás llamando desde 172.17.0.1
+ curl http://172.17.0.1:5050/
  % Total% Recibido% Xferd Velocidad Promedio Tiempo Tiempo Tiempo Actual
                                 Carga de descarga Velocidad total gastada izquierda

  0 0 0 0 0 0 0 0 --:--:-- --:--:-- --:--:-- 0
100 177 100 177 0 0 29772 0 --:--:-- --:--:-- --:--:-- 35400
    <h1>Me estás llamando desde 172.17.0.1</h1>
+ exit 0
Terminado: EXITOSO
```

b. No es necesario verificar que su aplicación de ejemplo se está ejecutando porque el **TestAppJob** ya hizo esto por usted. Sin embargo, puede abrir una pestaña del navegador para **172.17.0.1:5050** para ver que realmente se está ejecutando.

Parte 8: Crear una tubería/canalización en Jenkins

Aunque actualmente puede ejecutar sus dos trabajos simplemente haciendo clic en el botón **Compilar ahora** para **BuildAppJob**, los proyectos de desarrollo de software suelen ser mucho más complejos. Estos proyectos pueden beneficiarse enormemente de la automatización de compilaciones para la integración continua de los cambios de código y la creación continua de compilaciones de desarrollo listas para implementarse. Esta es la esencia del CI/CD. Una canalización se puede automatizar para ejecutarse en función de una variedad de desencadenadores, incluyendo periódicamente, basado en una encuesta de GitHub para cambios, o desde un script ejecutado de forma remota. Sin embargo, en esta parte, programaremos una canalización en Jenkins para ejecutar sus dos aplicaciones cada vez que hagamos click en el botón de canalización **Compilar ahora (Build Now)**.

Paso 1: Crear un trabajo de tubería/canalización.

a. Haremos clic en el enlace **Jenkins** en la parte superior izquierda y, a continuación, **Nuevo elemento (New Item)**.

b. En el campo **Escriba un nombre de elemento (Enter an item name)**, escribiremos **SamplePipeline**.

c. Seleccionaremos **Tubería (pipeline)** como tipo de trabajo.

d. Arrastre hasta la parte inferior de la página y haga click en **OK**.

Paso 2: Configurar el trabajo SamplePipeline.

a. En la parte superior, haga clic en las pestañas e investigue cada sección de la página de configuración. Observe que hay varias formas diferentes de activar una compilación. Para el trabajo **SamplePipeline**, lo activará manualmente.

b. En la sección **Pipeline**, agregue la siguiente secuencia de comandos (script).

```
node {
   stage ('Preparation') {
       CatchError (BuildResult: 'ÉXITO') {
          sh 'docker stop samplerunning'
          sh 'docker rm samplerunning'
       }
   }
   stage('Build') {
       build 'BuildAppJob'
   }
   stage ('Results') {
       build 'TestAppJob'
   }
}
```

Este script realiza lo siguiente:

- Crea una compilación de nodo único en lugar de un nodo distribuido o multinodo. Las configuraciones distribuidas o de varios nodos son para tuberías más grandes que las que está construyendo en este laboratorio y están fuera del alcance de este curso.
- En la etapa de **preparación**, **SamplePipeline** se asegurará primero de que las instancias anteriores del contenedor Docker **BuildAppJob** se detengan y se eliminen. Pero si aún no hay un contenedor en ejecución, obtendrá un error. Por lo tanto, utilizaremos la función **CatchError** para detectar cualquier error y devolver un valor "EXITOSO (SUCCESS)". Esto asegurará que la canalización continúe hasta la siguiente etapa.
- En la etapa de **compilación**, **SamplePipeline** compilará el **BuildAppJob**.

- En la etapa **Resultados**, **SamplePipeline** construirá su **TestAppJob**.

c. Haga clic en **Guardar** y volverá al panel de Jenkins para el trabajo de **SamplePipeline**.

Paso 3: Ejecutar SamplePipeline.

A la izquierda, haga clic en **Compilar ahora (Build Now)** para ejecutar el trabajo de **SamplePipeline**. Si se escribió correctamente el código del script de canalización (pipeline), entonces la **vista de escenario (stage view)** debería mostrar tres cuadros verdes con el número de segundos que cada etapa tardó en compilarse. Si no es así, haga clic en Configurar a la izquierda para volver a la configuración de **SamplePipeline** y comprobar el script de Pipeline.

Paso 4: Comprobar la salida de SamplePipeline.

Haga clic en el enlace de compilación más reciente en **Enlaces permanentes (permalikns)**y, a continuación, haga clic en **Salida de consola (Console Output)**. Debe ver un resultado similar al siguiente:

```
Iniciado por el usuario admin
Funcionando en nivel de durabilidad: MAX_SUPERVABILIDAD
[Pipeline] Inicio de la tubería
Nodo [Pipeline]
Ejecutando en Jenkins en /var/jenkins_home/workspace/samplePipeline
[Pipeline] {
[Pipeline] stage
[Pipeline] {(Preparation)
[Pipeline] CatchError
[Pipeline] {
[Pipeline] sh
+ docker stop samplerunning
samplerunning
[Pipeline] sh
+ docker rm samplerunning
samplerunning
[Pipeline] }
[Pipeline]//CatchError
[Pipeline] }
[Pipeline]// stage
[Pipeline] stage
[Pipeline] {(build)
[Pipeline] build (BuildAppJob)
Programando el proyecto: BuildAppJob
Comenzando a compilar: BuildAppJob #15
[Pipeline] }
[Pipeline]// stage
[Pipeline] stage
[Pipeline] {(Results)
[Pipeline] build (Building TestAppJob)
Programación del proyecto: TestAppJob
Iniciando la compilación: TestAppJob #18
[Pipeline] }
[Pipeline]// stage
[Pipeline] }
[Pipeline]// node
```

```
[Pipeline] Fin de la tubería
Terminado: EXITOSO
```