**Hands-On Lab**

Docker

**DevSecOps Theory to Practice**

**Contenido**

[1. Objetivo 3](#_Toc75616709)

[2. Instrucciones para levantar el entorno 3](#_Toc75616710)

[3. Codigo fuente de los laboratorios 5](#_Toc75616711)

[4. Ejercicio Docker 5](#_Toc75616712)

# Objetivo

El objetivo de esta sección es indicar los pasos para llevar a cabo el Hands-on Docker, el objetivo es entender como funcionan los contenedores y las ventajas que estos proporcionan.

# Instrucciones para levantar el entorno

Para iniciar el entorno deberás ingresar al portal de Microsoft Azure (<https://portal.azure.com>) con las credenciales que se te proporcionaron, tene en cuenta que la primera vez que inicies sesión se te solicitara un cambio de contraseña por razones de Seguridad.

Una vez dentro del portal veras una imagen como la siguiente:

A screenshot of a computer

Description automatically generated with medium confidence

Deberás ir al icono que se encuentra al lado de la barra de búsqueda:

A picture containing graphical user interface

Description automatically generated

Este icono te permite abrir una sesión de Cloud Shell, esto es una línea de comandos integrada 100 % en el navegador, lo que da como beneficio que no tengas que instalar absolutamente nada en tu computadora. Por ser la primera vez te va a aparecer una pantalla similar a la siguiente:

Graphical user interface, text, application

Description automatically generated

Le indican “Create Storage” y en cuestión de 2 a 3 minutos tendrán la consola lista para ejecutar comandos:

Text

Description automatically generated

Esta terminal la podrán iniciar las veces que sea necesario indicando el botón que se encuentra al lado del cuadro de búsqueda dentro del portal de Microsoft Azure.

# Codigo fuente de los laboratorios

Para facilitar la distribución de los laboratorios autoguiados los mismos han sido subidos a GitLab, con lo cual podrás clonarlos de ahí y hacerlos a medida que se vayan revisando a lo largo del curso. La URL del repositorio es:

<https://github.com/Cloud-Legion/DevSecOps-Theory-To-Practice.git>

La estructura es similar a la siguiente:

Graphical user interface, text, application, email

Description automatically generated

No te preocupes que los instructores te irán guiando con relación a que laboratorios hacer y estarán atentos a cualquier duda que tengas durante el desarrollo de los mismos😊

# Ejercicio Docker

Llego la hora del ejercicio, habiendo hecho lo anterior estas listo para empezar a meter mano, para esto abrí una sesión de Cloud Shell y espera que se abra la ventana:

Graphical user interface, text, application

Description automatically generated

Dentro de la terminal ejecuta el siguiente comando para clonar el repositorio desde GitHub a tu espacio privado dentro de la terminal anteriormente configurada:

**git clone https://github.com/Cloud-Legion/DevSecOps-Theory-To-Practice.git**

Esto te lo descargara a tu espacio privado y con el comando **ls** podrás visualizar la carpeta de manera local:

Text

Description automatically generated

Hecho esto debemos ingresar a la carpeta indicando la orden **cd DevSecOps-Theory-To-Practice**, podes tabular también en la primera palabra para que la terminal te lo autocomplete 😊, hecho eso veras algo como lo siguiente (que es lo mismo que verías en GitHub)



Deberás ir a la carpeta **\DevSecOps-Theory-To-Practice\Semana 1\Dia 4\Laboratorio Docker\Terraform** con la orden **cd**, una vez ahí indica el comando ls para ver el contenido de la carpeta, deberías ver 2 archivos como los siguientes:

**Text

Description automatically generated with medium confidence**

Estos son manifestos de Terraform y te permiten levantar cualquier tipo de infraestructura sin tener que configurar nada manualmente, para este caso puntual Terraform va a levantar una instancia 100 % configurada de Docker para que puedas desarrollar el laboratorio sin tener que hacer configuraciones de instalación, dentro de esta carpeta ejecuta el comando **terraform init**, al terminar la ejecucion deberías ver una pantalla como la siguiente indicando que se inicializo correctamente:

Text

Description automatically generated

Concluido esto ejecuta **terraform apply --auto-approve**, esto hará que se despliegue una máquina virtual dentro de tu misma suscripción de Azure con GitLab instalado, a esta maquina virtual vas a poder acceder a través de Internet con una IP publica provista automáticamente.

La provision de esta maquina virtual tardara de 5 a 7 minutos debido a configuraciones que deben realizarse, deberás ver una pantalla como la siguiente a lo largo del proceso:

Text

Description automatically generated

Terminada la ejecucion es probable que se sigan ejecutando tareas en segundo plano, pero en este punto deberías tener un output con la IP publica asignada:

Diagram

Description automatically generated with medium confidence

Concluido el proceso podés ingresar con estos datos:

**ssh azureuser@xxx.xxx.xxx.xxx**  (\*) Donde xxx.xxx.xxx.xxx es la IP Publica

Ingresar con los siguientes datos:

Usuario: **azureuser**

Password: **Password1234!**

Una vez dentro del servidor veras una pantalla como la siguiente:

**Text

Description automatically generated**

Si ejecutas el comando **sudo docker ps** veras que el daemon te responde que no tiene contenedores ejecutándose, pero te permite validar que efectivamente Docker esta instalado y listo para usarse.

Text

Description automatically generated with medium confidence

**Instrucciones básicas de Docker**

**Docker pull**

**pull** es un argumento para descargar la imagen docker desde el registro docker, es similar al repositorio git pero diseñado para almacenar las imágenes docker que el código.

Por defecto, el comando docker extraerá la imagen de DockerHub. Cuando se hace un pull, si la etiqueta/versión no está declarada para una imagen, Docker asume el ultimo TAG por defecto.

**docker pull nginx**

**Text

Description automatically generated**

**Docker run**

**run** es un argumento para iniciar un contenedor desde la imagen docker. Hay un montón de opciones que podemos utilizar, como un nombre para el contenedor, el puerto de enlace, y así sucesivamente.

Por ejemplo, podemos ejecutar un contenedor en segundo plano usando la opción -d y darle un nombre usando la opción -name.

**docker run -d --name demo nginx**

**Graphical user interface, text, website

Description automatically generated**

El contenedor se iniciará/ejecutará en segundo plano debido a que indicamos la opcion -d y hay un proceso corriendo de fondo como es el caso de NGINX.

**Docker exec**

**exec** es un argumento para ejecutar un comando dentro de un contenedor o para obtener el shell en el contenedor.

**docker exec demo whoami**

**Text

Description automatically generated**

O simplemente ejecutamos el comando **docker exec -it demo /bin/sh** para entrar a través de un shell al contenedor, nos daremos cuenta de que estamos dentro por el id del contenedor.

**Text

Description automatically generated**

**Listar Contenedores**

**ps** es un argumento para listar contenedores (forma de ejecución de imágenes). Proporciona varias opciones para filtrar la salida según el estado del contenedor como creado, en ejecución, salido o detenido, etc. Vamos a ejecutar un contenedor que sale y ver si la opción -a nos muestra el estado del contenedor.



**Construir imágenes de Docker**

A continuación veremos como crear una imagen de Docker con una aplicación real, para esto vamos a realizar los siguientes pasos

Tene en cuenta que en la maquina virtual que estas usando ya fueron instalados todo los prerrequisitos. Vamos a empezar clonando el repositorio de la aplicación de ejemplo que listamos a continuación con el siguiente comando:

**sudo su**

**git clone https://github.com/docker/getting-started.git**

**Text

Description automatically generated**

Accedemos al repositorio clonado mediante la orden **cd** y listamos los archivos con **ls -l**

Una vez dentro del repositorio accedemos a la carpeta denominada “app” mediante la orden **cd**

Text

Description automatically generated

Ahora debemos crear un archivo llamado Dockerfile, podemos hacerlo con el comando:

**nano Dockerfile**

Pegamos el siguiente contenido:

**FROM node:12-alpine**

**RUN apk add --no-cache python g++ make**

**WORKDIR /app**

**COPY . .**

**RUN yarn install --production**

**CMD ["node", "src/index.js"]**

Text

Description automatically generated

(\*) Fijate de que no te queden espacios antes de las instrucciones de Docker (FROM, RUN y demás)

Guardamos el archivo con CTRL +X y ejecutamos el siguiente comando, el cual será encargado de hacer build de la imagen

**docker build -t getting-started .**

**Text

Description automatically generated**

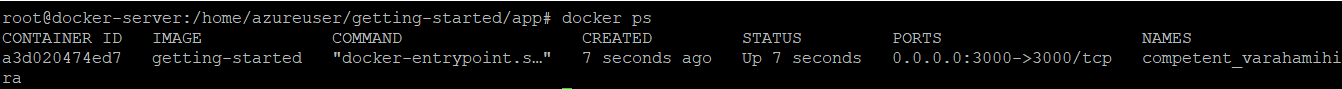
Una vez buildeada esta imagen podemos verla en nuestro repositorio de imágenes ejecutando el comando **docker Images**

Text

Description automatically generated

Con esto ya podemos crear un nuevo contenedor usando esta imagen que creamos anteriormente, para esto ejecutamos

**docker run -d -p 3000:3000 getting-started**

****

Con esto el contenedor se esta ejecutando, dado que es una aplicación basada en NodeJS por defecto expone en el puerto 3000, y dado que este puerto lo expusimos usando el flag **-p 3000:3000** deberíamos poder acceder a la aplicación si ponemos la IP Publica de la máquina virtual + :3000

Graphical user interface, text, application

Description automatically generated

Ahora que tenemos nuestro primer imagen creada con una aplicación real podemos empezar a hablar de aspectos de Seguridad, como vimos en el curso tenemos 3 aspectos, uno es seguridad en la imagen de Docker, el otro es la Seguridad y Hardening sobre la maquina virtual donde se ejecuta Docker y el tercero es hacer un lint (comprobación del Dockerfile). Empecemos con el punto 1.

Para hacer análisis de imágenes de Docker hay multitud de herramientas, para este caso practico utilizaremos **Trivy**, estos analizadores se encargan de revisar los componentes de Software con los que cuenta la imagen tales como dependencias y librerías, para instalarlo hacemos lo siguiente:

**wget https://github.com/aquasecurity/trivy/releases/download/v0.18.3/trivy\_0.18.3\_Linux-64bit.deb**

**sudo dpkg -i trivy\_0.18.3\_Linux-64bit.deb**

Text

Description automatically generated

Instalada la herramienta podremos ejecutar análisis contra las imágenes que tenemos en nuestra maquina virtual usando el comando

**trivy + nombre de la imagen**

por ej.

**trivy getting-started**

**Text

Description automatically generated**

Como vemos se han encontrado vulnerabilidades en la imagen que estuvimos construyendo anteriormente, esto cuando lo integremos a un pipeline nos va a permitir detectar vulnerabilidades en un estadio temprano del ciclo de desarrollo de manera automática.

En relación con el segundo aspecto en donde se puede hacer la aplicación de Seguridad en todo lo relacionado al hardening del host donde se ejecuta Docker (Maquina virtual o Servidor físico).

De vuelta, hay muchas pero muchas herramientas que nos pueden ayudar con esto, para este ejercicio estaremos usando **Docker Bench**, para ejecutar la herramienta descargaremos el siguiente script y lo ejecutaremos en la maquina virtual que estamos usando para el ejercicio y que ya cuenta con Docker instalado.

**git clone https://github.com/docker/docker-bench-security.git**

**cd docker-bench-security**

**sudo sh docker-bench-security.sh**

**Text

Description automatically generated**

**Text

Description automatically generated**

Como podemos ver esta herramienta agrupa las recomendaciones por diferentes categorías, imágenes, host y procesos. En el caso puntual de este laboratorio es un Docker instalado de manera predeterminada, con lo cual podemos ver que aun después de instalarlo hay cosas por ajustar. Esto puede ser usado también como parte de un pipeline automatizado.

Por último, podemos hacer lint del Dockerfile utilizando una herramienta como **hadolint**, esta permite analizar la sintaxis del Dockerfile y detectar configuraciones erróneas o peligrosas en lo que respecta a Seguridad (Ej. ejecutar el contenedor como Root)

Ejecutaremos el siguiente comando para hacerlo referenciando al Dockerfile que tenemos, el cual usamos para buildear la imagen anterior en este mismo ejercicio.

Text

Description automatically generated

**docker run –rm -i hadolint/hadolint < Dockerfile**

Text

Description automatically generated

Como podemos ver nos da recomendaciones de como deberíamos estructurar nuestras sentencias, prueben modificar el puerto de su Dockerfile del 3000 al 70000 a ver que error les arroja 😊. Esto integrado a un pipeline automatizado puede ayudarnos a corregir ciertos errores durante la marcha.

Concluido el laboratorio no te olvides de ejecutar **terraform destroy** para destruir la maquina virtual utilizada para las pruebas.