Docker 2023

Diego Saavedra

Dec 10, 2023

Table of contents

1 Curso de Docker

¡Bienvenidos al Curso de Docker!

Este curso te guiará a través de un viaje desde los fundamentos hasta el dominio de Docker, la plataforma de contenedores líder en la industria.

1.1 ¿Qué es este Curso?

Este curso exhaustivo te llevará desde los conceptos básicos de Docker hasta la implementación práctica de aplicaciones y servicios en contenedores. A través de una combinación de teoría y ejercicios prácticos, explorarás cada aspecto diseñado para fortalecer tus habilidades en el uso de Docker. Desde la instalación inicial hasta la orquestación de contenedores con Docker Compose, este curso te proporcionará las herramientas y conocimientos necesarios para desarrollar, desplegar y escalar aplicaciones con eficiencia.

1.2 ¿A quién está dirigido?

Este curso está diseñado tanto para aquellos que están dando sus primeros pasos en Docker como para aquellos que desean profundizar en sus conocimientos. Ya seas un estudiante, un profesional en busca de nuevas habilidades o alguien apasionado por la tecnología de contenedores, este curso te brindará la base necesaria para trabajar de manera efectiva con Docker en cualquier entorno.

1.3 ¿Cómo contribuir?

Valoramos tu participación en este curso. Si encuentras errores, deseas sugerir mejoras o agregar contenido adicional, ¡nos encantaría recibir tus contribuciones! Puedes contribuir a través de nuestra plataforma en línea, donde puedes compartir tus comentarios y sugerencias. Juntos, podemos mejorar continuamente este recurso educativo para beneficiar a la comunidad de usuarios de Docker.

Este curso ha sido creado con el objetivo de proporcionar acceso gratuito y universal al conocimiento de Docker. Estará disponible en línea para que cualquiera, sin importar su ubicación o circunstancias, pueda acceder y aprender a su propio ritmo.

¡Esperamos que disfrutes este emocionante viaje de aprendizaje y descubrimiento en el mundo de Docker y los contenedores!

Part I

Curso Docker

2 1. Comandos Básicos y Atajos

2.1 Conceptos:

2.1.1 Docker

Es una plataforma para desarrollar, enviar y ejecutar aplicaciones en contenedores. Un contenedor es una instancia ejecutable de una imagen.

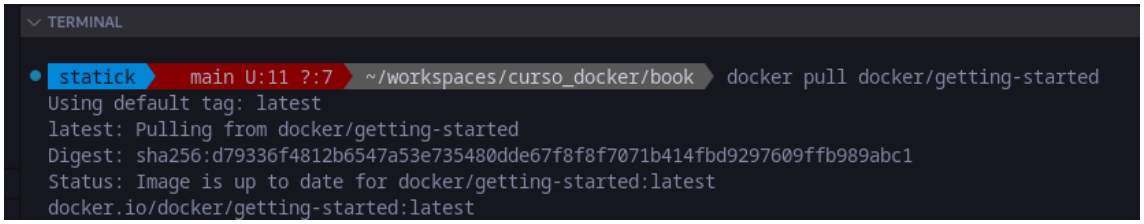
2.1.2 Contenedor

Es una instancia de una imagen que se ejecuta de manera aislada. Los contenedores son ligeros y portátiles, ya que incluyen todo lo necesario para ejecutar una aplicación, incluidas las bibliotecas y las dependencias.

2.2 Ejemplos:

Descargar una imagen:

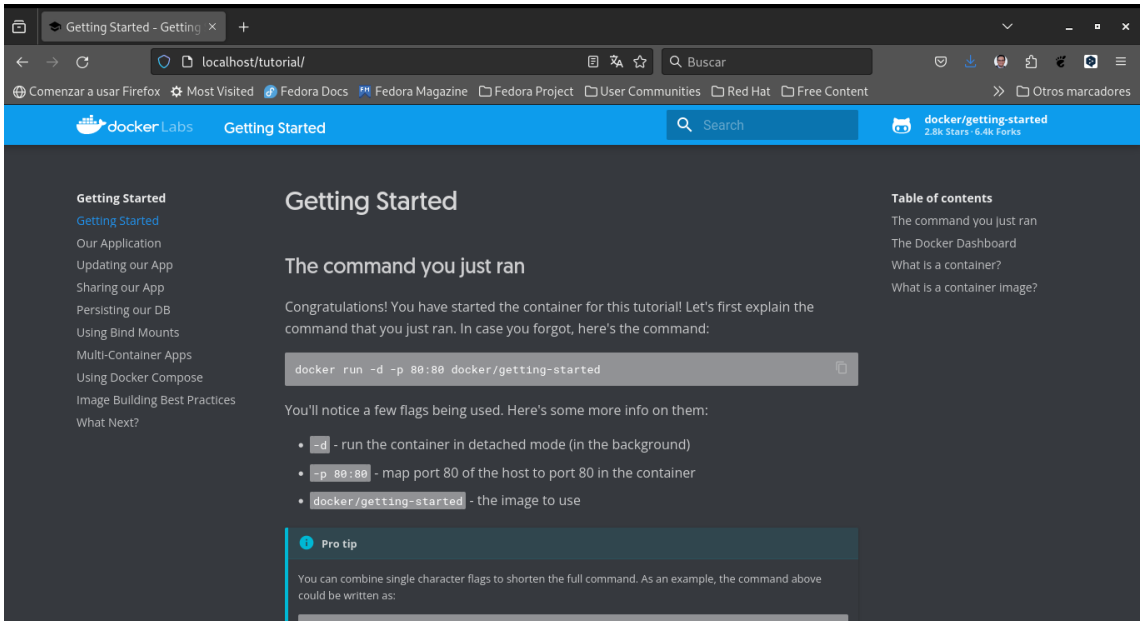
```
docker pull docker/getting-started
```



Este comando descarga la imagen **getting-started** desde el registro público de Docker.

Correr un contenedor en el puerto 80:

```
docker run -d -p 80:80 docker/getting-started
```



Este comando ejecuta un contenedor desenlazado en segundo plano (-d) y mapea el puerto 80 de la máquina host al puerto 80 del contenedor (-p 80:80).

Descargar una imagen desde un registro.

```
docker pull <IMAGE_NAME:TAG>
```

Listar las imágenes descargadas.

```
docker images
```

Listar contenedores en ejecución.

```
docker ps
```

Listar todos los contenedores, incluyendo los detenidos.

```
docker ps -a
```

Ejecutar un contenedor a partir de una imagen.

```
docker run -d -p <HOST_PORT>:<CONTAINER_PORT> <IMAGE_NAME:TAG>
```

Detener un contenedor en ejecución.

```
docker stop <CONTAINER_ID>
```

Iniciar un contenedor detenio.

```
docker start <CONTAINER_ID>
```

Eliminar un contenedor.

```
docker rm <CONTAINER_ID>
```

Eliminar una imagen.

```
docker rmi <IMAGE_NAME:TAG>
```

2.3 Atajos y Comandos Adicionales:

Ejecutar comandos dentro de un contenedor en ejecución.

```
docker exec -it <CONTAINER_ID> /bin/bash
```

Obtener detalles sobre un contenedor o imagen.

```
docker inspect <CONTAINER_ID or IMAGE_NAME:TAG>
```

Ver los logs de un contenedor.

```
docker logs <CONTAINER_ID>
```

Utilizar Docker Compose para gestionar aplicaciones multi-contenedor.

```
docker-compose up -d
```

2.4 Práctica:

- Descarga la imagen de Nginx desde el registro público.
- Crea y ejecuta un contenedor de Nginx en el puerto 8080.
- Detén y elimina el contenedor creado
- Utiliza los comandos para detener y eliminar un contenedor.

Resolución de la Actividad Práctica

1. Abre tu terminal o línea de comandos.
2. Descarga la imagen de Nginx desde el registro público de Docker:

```
docker pull nginx
```

3. Crea y ejecuta un contenedor de Nginx en el puerto 8080:

```
docker run -d -p 8080:80 nginx
```

Elige un puerto en tu máquina local (por ejemplo, 8080) para mapearlo al puerto 80 del contenedor.

4. Verifica que el contenedor esté en ejecución:

```
docker ps
```

5. Si el contenedor está en ejecución, detenlo utilizando el siguiente comando:

```
docker stop <CONTAINER_ID>
```

Reemplaza `<CONTAINER_ID>` con el ID real del contenedor que obtuviste en el paso anterior.

6. Elimina el contenedor detenido:

```
docker rm <CONTAINER_ID>
```

Reemplaza `<CONTAINER_ID>` con el ID real del contenedor.

> **Tip**
>
> Combina los comandos docker ps, docker stop, y docker rm para gestionar contenedores eficientemente.
> ¡Practica estos pasos para familiarizarte con el ciclo de vida de los contenedores Docker!

2.5 ¿Qué Aprendimos?

- Aprendimos a descargar imágenes, correr contenedores y gestionarlos básicamente.
- Entendimos la importancia de las banderas en los comandos Docker.

3 Actividad Práctica

3.1 Objetivo:

Familiarizarse con los comandos básicos de Docker y atajos para gestionar contenedores de manera eficiente. Instrucciones:

- Utilizando el comando docker run, inicia un contenedor de la imagen “nginx” en segundo plano, mapeando el puerto 8080 del host al puerto 80 del contenedor.
- Detén y elimina el contenedor recién creado utilizando comandos Docker.
- Crea un nuevo contenedor con la imagen “alpine” y ejecuta un terminal interactivo dentro de él.
- Desde el contenedor alpine, instala el paquete curl utilizando el gestor de paquetes apk.
- Crea una imagen llamada “alpine-curl” a partir de este contenedor modificado.

3.2 Entregables:

- Documento explicando los comandos utilizados.
- Imagen Docker “alpine-curl” disponible localmente.

Resolución de la Actividad Práctica

- Iniciar un contenedor Nginx:

```
docker run -d -p 8080:80 --name my-nginx nginx
```

Detener y eliminar el contenedor Nginx:

```
docker stop my-nginx
docker rm my-nginx
```

Crear un contenedor Alpine interactivo:

```
docker run -it --name my-alpine alpine /bin/sh
```

Instalar el paquete curl desde el contenedor Alpine:

```
apk add --no-cache curl
```

Crear una nueva imagen “alpine-curl”:

```
docker commit my-alpine alpine-curl
```

3.3 Rubrica de Evaluación:

- Correcta ejecución de comandos: 6 puntos
- Clara documentación: 4 puntos
- Imagen “alpine-curl” creada correctamente: 10 puntos

4 2. Dockerfile y Docker Compose

4.1 Conceptos:

4.1.1 Dockerfile

Un Dockerfile es un archivo de texto que contiene instrucciones para construir una imagen Docker. Es esencialmente un script que define cómo se construirá la imagen.

4.1.2 Docker Compose

K Docker Compose es una herramienta para definir y gestionar aplicaciones Docker con múltiples contenedores. Permite definir la configuración de servicios, redes y Kvolúmenes en un archivo YAML.

4.2 Ejemplos:

Crear un Dockerfile para una aplicación Node.js:

```
FROM node:14
WORKDIR /app
COPY . .
CMD ["npm", "start"]
```

Este Dockerfile configura una imagen de Node.js, establece el directorio de trabajo, copia los archivos locales al contenedor y define el comando para ejecutar la aplicación.

Configurar Docker Compose para una aplicación Node.js:

```
version: '3'
services:
  myapp:
    build:
      context: .
      dockerfile: Dockerfile.node
    image: my-node-app
```

Este archivo docker-compose.yml define un servicio llamado “myapp” que construirá una imagen usando el Dockerfile “Dockerfile.node” y le asignará el nombre de “my-node-app”.

4.3 Práctica:

- Crea un Dockerfile para una aplicación Python simple.
- Configura un archivo docker-compose.yml para ejecutar la aplicación.

Resolución de la Actividad Práctica

Ejemplo de Dockerfile (nombre: Dockerfile.python):

```
FROM python:3.9
WORKDIR /app
COPY . .
CMD ["python", "app.py"]
```

Ejemplo de docker-compose.yml:

```
version: '3'
services:
  myapp:
    build:
      context: .
      dockerfile: Dockerfile.python
    image: my-python-app
```

4.4 ¿Qué Aprendimos?

- Aprendimos a crear un Dockerfile para personalizar una imagen Docker.
- Entendimos cómo usar Docker Compose para orquestar servicios en un entorno multi-contenedor.
- Practicamos la configuración básica de un Dockerfile y un archivo docker-compose.yml.

5 Actividad Práctica

5.1 Objetivo:

Practicar la creación de imágenes personalizadas utilizando Dockerfile y la orquestación de servicios con Docker Compose.

5.2 Instrucciones:

- Crea un Dockerfile para una aplicación Python simple que imprima “Hola, Docker” al ejecutarse.
- Construye la imagen a partir del Dockerfile.
- Utilizando Docker Compose, define un servicio que utilice la imagen creada y exponga el puerto 5000.
- Inicia el servicio con Docker Compose.
- Accede a la aplicación en http://localhost:5000 y verifica que imprime “Hola, Docker”.

5.3 Entregables:

- Dockerfile para la aplicación Python.
- Archivo Docker Compose.
- Documento explicando los comandos utilizados.
- Capturas de pantalla que demuestren el acceso a la aplicación.

Resolución de la Actividad Práctica

Dockerfile para la aplicación Python:

```
FROM python:3.9
CMD ["python", "-c", "print('Hola, Docker')"]
```

Construir la imagen:

```
docker build -t my-python-app .
```

Archivo Docker Compose (docker-compose.yml):

```yaml
version: '3'
services:
  myapp:
    image: my-python-app
    ports:
      - "5000:5000"
```

Iniciar el servicio con Docker Compose:

```
docker-compose up -d
```

Verificar el acceso a la aplicación:

Acceder a http://localhost:5000 en el navegador.

5.4 Rubrica de Evaluación:

- Correcta creación del Dockerfile: 6 puntos
- Imagen construida correctamente: 4 puntos
- Configuración adecuada en Docker Compose: 6 puntos
- Acceso exitoso a la aplicación: 4 puntos

6 3. Creación de un servidor web con Docker y Nginx

6.1 Conceptos

Docker es una plataforma que permite a los desarrolladores empaquetar aplicaciones en contenedores. Un **contenedor** es una unidad estándar de software que empaqueta el código y todas sus dependencias para que la aplicación se ejecute de manera rápida y confiable de un entorno informático a otro.

Nginx es un servidor web que puede usarse para servir contenido estático, como archivos HTML, CSS y JavaScript.

6.2 Ejemplos

Un ejemplo de un Dockerfile para un servidor web Nginx es el siguiente:

```
# Use an official nginx image as a parent image
FROM nginx:latest

# Set the working directory in the container to /usr/share/nginx/html
WORKDIR /usr/share/nginx/html

# Copy the 'web' directory (at your Dockerfile's location) into the container
COPY web .
```

6.3 Actividad Práctica

1. Crea un directorio llamado web y añade algunos archivos HTML, CSS y JavaScript.
2. Crea un Dockerfile como el del ejemplo anterior.
3. Construye una imagen Docker a partir del Dockerfile
4. Crea y ejecuta un contenedor a partir de la imagen

Resolución de la Actividad Práctica

Proyecto Web con Docker y Nginx

Este proyecto utiliza Docker y Nginx para servir una aplicación web estática.

Creación del Dockerfile

Crea un archivo llamado `Dockerfile` en la raíz del proyecto con el siguiente contenido:

```
# Use an official nginx image as a parent image
FROM nginx:latest

# Set the working directory in the container to /usr/share/nginx/html
WORKDIR /usr/share/nginx/html

# Copy the 'web' directory (at your Dockerfile's location) into the container
COPY web .
```

Construcción de la imagen Docker

Para construir una imagen Docker a partir del Dockerfile, ejecuta el siguiente comando en la terminal:

```
docker build -t my-nginx-image .
```

Este comando crea una nueva imagen Docker llamada **my-nginx-imag**e a partir del Dockerfile.

Creación del contenedor Docker.

Para crear y ejecutar un contenedor a partir de la imagen que acabas de crear, ejecuta el siguiente comando en la terminal:

```
docker run -it --rm -dp 8080:80 -v ${pwd}/web:/usr/share/nginx/html --name web my-nginx
```

Este comando crea y ejecuta un nuevo contenedor Docker llamado **web** a partir de la imagen **my-nginx-image**.

El contenedor sirve la aplicación web en el puerto 8080 y monta el directorio **web** de tu máquina local en el directorio **/usr/share/nginx/html** del contenedor.

Esto significa que cualquier cambio que hagas en los archivos de tu directorio **web** local se reflejará en vivo en el contenedor.

6.4 ¿Qué aprendimos?

En este tutorial, aprendimos cómo usar Docker y Nginx para crear un servidor web que sirve contenido estático. Aprendimos cómo crear un Dockerfile, cómo construir una imagen Docker a partir de un Dockerfile, y cómo crear y ejecutar un contenedor a partir de una imagen Docker. También aprendimos cómo montar un directorio de nuestra máquina local en un contenedor Docker para poder ver los cambios en vivo en nuestro servidor web.

7 Actividad Práctica

7.1 Objetivo:

Practicar el uso de volúmenes en Docker para persistir datos entre contenedores.

7.2 Instrucciones:

- Crea un nuevo volumen llamado “mydata”.
- Inicia un contenedor de la imagen “nginx” y vincula el volumen “mydata” al directorio “/usr/share/nginx/html” dentro del contenedor.
- Crea un archivo HTML dentro del volumen “mydata” con el mensaje “Hola, este es un archivo HTML persistente”.
- Inicia otro contenedor de la imagen “nginx” y vincula el mismo volumen “mydata” al directorio “/usr/share/nginx/html” dentro de este segundo contenedor.
- Verifica que ambos contenedores comparten el mismo archivo HTML creado en el paso 3.

7.3 Entregables:

- Documento explicando los comandos utilizados.
- Capturas de pantalla que demuestren la persistencia de datos entre contenedores.

Resolución de la Actividad Práctica

Crear un nuevo volumen:

```
docker volume create mydata
```

Iniciar el primer contenedor Nginx con el volumen:

```
docker run -d -p 8080:80 --name nginx-1 -v mydata:/usr/share/nginx/html nginx
```

Crear un archivo HTML dentro del volumen:

```
docker exec -it nginx-1 sh -c "echo 'Hola, este es un archivo HTML persistente' > /usr/
```

Iniciar el segundo contenedor Nginx con el mismo volumen:

```
docker run -d -p 8081:80 --name nginx-2 -v mydata:/usr/share/nginx/html nginx
```

Verificar la persistencia del archivo HTML:

- Acceder a http://localhost:8080 en el navegador.
- Acceder a http://localhost:8081 en el navegador.

7.4 Rubrica de Evaluación:

- Correcta creación y vinculación de volúmenes: 6 puntos
- Creación y persistencia de archivos en el volumen: 8 puntos
- Verificación exitosa de la persistencia entre contenedores: 6 puntos

8 4. Dockerizando un Ambiente de Desarrollo con Python y Django

8.1 Introducción

Docker ha revolucionado la forma en que desarrollamos, entregamos y ejecutamos aplicaciones. En este tutorial, exploraremos cómo dockerizar un ambiente de desarrollo para una aplicación Python utilizando el framework Django. Docker simplifica la creación de entornos aislados y reproducibles, lo que facilita el desarrollo y la colaboración en equipos.

8.2 ¿Qué es Docker?

Docker es una plataforma que permite desarrollar, enviar y ejecutar aplicaciones en contenedores. Un contenedor es una instancia ejecutable de una imagen que incluye todo lo necesario para ejecutar una aplicación, como bibliotecas, dependencias y el propio código. Creando un Dockerfile para una Aplicación Django

Un Dockerfile es esencial para construir una imagen Docker personalizada. Aquí hay un ejemplo básico para una aplicación Django:

```
# Usa una imagen oficial de Python como base
FROM python:3.12

# Establece el directorio de trabajo en el contenedor
WORKDIR /app

# Copia los archivos de requerimientos y el código de la aplicación
COPY requirements.txt .

# Instala las dependencias
RUN pip install --no-cache-dir -r requirements.txt

# Copia el resto de la aplicación
COPY . .

# Expone el puerto en el que se ejecutará la aplicación
EXPOSE 8000

# Comando para ejecutar la aplicación Django
```

```
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

Este Dockerfile utiliza una imagen base de Python, instala las dependencias desde el archivo requirements.txt y configura la aplicación Django para ejecutarse en el puerto 8000. Docker Compose para Orquestar Contenedores

Docker Compose es una herramienta que simplifica la gestión de aplicaciones multi-contenedor. Un archivo docker-compose.yml define la configuración de los servicios, redes y volúmenes. Aquí hay un ejemplo para nuestra aplicación Django:

```
version: '3'
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    image: my-django-app
    ports:
      - "8000:8000"
    volumes:
      - .:/app
    depends_on:
      - db
  db:
    image: postgres:latest
    environment:
      POSTGRES_DB: mydatabase
      POSTGRES_USER: myuser
      POSTGRES_PASSWORD: mypassword
```

Este archivo docker-compose.yml define dos servicios: web (la aplicación Django) y db (una base de datos PostgreSQL). El servicio web depende del servicio db, asegurando que la base de datos esté disponible antes de iniciar la aplicación Django.

8.3 Actividad Práctica

- Crea un directorio para tu proyecto Django y coloca el Dockerfile y el docker-compose.yml en él.
- Inicia el entorno de desarrollo ejecutando docker-compose up -d.
- Accede a la aplicación Django en http://localhost:8000 en tu navegador.
- Realiza cambios en tu aplicación y observa cómo Docker facilita la actualización del entorno.

Resolución de la Actividad Práctica

- Crea un directorio llamado mi-proyecto-django y coloca el Dockerfile y el docker-compose.yml en él.

Abre una terminal, navega al directorio del proyecto y ejecuta el siguiente comando:

```
docker-compose up -d
```

Esto construirá la imagen y ejecutará los contenedores en segundo plano.

Accede a la aplicación Django en tu navegador ingresando a http://localhost:8000. Deberías ver la aplicación en ejecución.

Realiza cambios en tu aplicación Django local, como modificar archivos estáticos o de templates. Los cambios se reflejarán automáticamente en el contenedor gracias al volumen configurado en el docker-compose.yml.

8.4 ¿Qué aprendimo?

En este tutorial, hemos explorado cómo dockerizar un ambiente de desarrollo para una aplicación Python y Django. Al utilizar Docker y Docker Compose, hemos creado un entorno consistente y reproducible, facilitando el desarrollo y la colaboración en equipo. Además, hemos practicado la creación de un Dockerfile, la configuración de Docker Compose y la gestión de servicios multi-contenedor. ¡Ahora estás listo para llevar tu desarrollo con Python y Django a un nivel más eficiente y portable!

9 Actividad Práctica

9.1 Objetivo:

Practicar la creación de imágenes personalizadas utilizando Dockerfile y la orquestación de servicios con Docker Compose.

9.2 Instrucciones:

- Crea un Dockerfile para una aplicación Python simple que imprima “Hola, Docker” al ejecutarse.
- Construye la imagen a partir del Dockerfile.
- Utilizando Docker Compose, define un servicio que utilice la imagen creada y exponga el puerto 5000.
- Inicia el servicio con Docker Compose.
- Accede a la aplicación en http://localhost:5000 y verifica que imprime “Hola, Docker”.

9.3 Entregables:

- Dockerfile para la aplicación Python.
- Archivo Docker Compose.
- Documento explicando los comandos utilizados.
- Capturas de pantalla que demuestren el acceso a la aplicación.

Resolución de la Actividad Práctica

Dockerfile para la aplicación Python:

```
FROM python:3.9
CMD ["python", "-c", "print('Hola, Docker')"]
```

Construir la imagen:

```
docker build -t my-python-app .
```

Archivo Docker Compose (docker-compose.yml):

```yaml
version: '3'
services:
  myapp:
    image: my-python-app
    ports:
      - "5000:5000"
```

Iniciar el servicio con Docker Compose:

```
docker-compose up -d
```

Verificar el acceso a la aplicación:

Acceder a http://localhost:5000 en el navegador.

9.4 Rubrica de Evaluación:

- Correcta creación del Dockerfile: 6 puntos
- Imagen construida correctamente: 4 puntos
- Configuración adecuada en Docker Compose: 6 puntos
- Acceso exitoso a la aplicación: 4 puntos

10 4: Buenas Prácticas y Seguridad

10.1 Conceptos:

10.1.1 Escaneo de imagen:

Después de construir una imagen, es una buena práctica realizar un escaneo en busca de vulnerabilidades. Esto implica el uso de herramientas como Snyk, que examinan la imagen en busca de posibles problemas de seguridad.

10.1.2 Capas de la imagen:

Cada imagen de Docker se construye en capas. Las capas son un aspecto clave para entender la construcción y optimización de imágenes. Docker utiliza un sistema de archivos en capas, donde cada instrucción en el Dockerfile crea una nueva capa en la imagen. Este enfoque permite la reutilización eficiente de capas entre imágenes, lo que ahorra espacio de almacenamiento y tiempo de construcción.

10.1.3 Multi-Stage builds:

Las Multi-Stage builds son una práctica recomendada para construir imágenes Docker. Permiten separar las dependencias necesarias para construir la aplicación de las necesarias para ejecutarla en producción. Al hacerlo, se reduce significativamente el tamaño de la imagen final. Un ejemplo común es construir una aplicación Node.js y luego copiar solo los artefactos necesarios en una imagen más ligera de Nginx para la producción.

10.2 Ejemplos:

Escaneo de imagen con Snyk:

```
snyk container test <IMAGE_NAME:TAG>
```

Utilizando la herramienta Snyk, podemos escanear una imagen en busca de vulnerabilidades. Esto es esencial para identificar y abordar posibles riesgos de seguridad antes de implementar la aplicación.

Uso de Multi-Stage builds:

```
FROM node:14 AS builder
WORKDIR /app
COPY . .
RUN npm install
RUN npm run build

FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
```

Este Dockerfile utiliza Multi-Stage builds para primero construir una aplicación Node.js y luego copiar solo los artefactos necesarios en una imagen más ligera de Nginx. Este enfoque no solo facilita el proceso de construcción sino que también resulta en imágenes más eficientes y seguras.

10.3 Práctica:

- Realiza un escaneo de vulnerabilidades en una imagen de tu elección utilizando Snyk.
- Implementa un Multi-Stage build en un Dockerfile para una aplicación de tu preferencia.

Resolución de la Actividad Práctica

Escaneo de imagen con Snyk:

```
snyk container test my-image:my-tag
```

Ejemplo de Multi-Stage Dockerfile:

```
FROM node:14 AS builder
WORKDIR /app
COPY . .
RUN npm install
RUN npm run build

FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
```

> Tip
>
> Recuerda crear contenedores efímeros y desacoplar aplicaciones para mejorar el rendimiento y la eficiencia.

10.4 ¿Qué Aprendimos?

- Comprendemos la importancia de realizar escaneos de seguridad en las imágenes de Docker.

- Aprendimos cómo implementar Multi-Stage builds para optimizar el tamaño y la seguridad de las imágenes.

11 Actividad Práctica

11.1 Objetivo:

Aplicar buenas prácticas y medidas de seguridad al trabajar con Docker.

11.2 Instrucciones:

- Escanea la imagen “alpine:latest” en busca de vulnerabilidades utilizando la herramienta Snyk.
- Implementa una política de etiquetado para las imágenes Docker que siga las mejores prácticas.
- Utiliza la herramienta “docker-compose lint” para verificar la validez del archivo Docker Compose.
- Implementa una red de contenedores y asegúrate de que solo los contenedores necesarios tengan acceso a ella.
- Crea un archivo “.dockerignore” para excluir archivos y directorios innecesarios en la construcción de imágenes Docker.

11.3 Entregables:

- Capturas de pantalla del escaneo de vulnerabilidades.
- Política de etiquetado para imágenes Docker.
- Resultado de la verificación del archivo Docker Compose.
- Documento explicando la implementación de la red de contenedores.
- Archivo “.dockerignore”.

Resolución de la Actividad Práctica

Escaneo de vulnerabilidades:

```
snyk container test alpine:latest
```

Política de etiquetado (ejemplo):

Se etiquetarán las imágenes con el formato “versión-año-mes-día” (ejemplo: 1.0-20230115).

Verificación del archivo Docker Compose:

```
docker-compose config
```

Implementación de una red de contenedores:

```
docker network create my-network
```

Archivo ".dockerignore" (ejemplo):

```
node_modules
.git
.env
```

11.4 Rubrica de Evaluación:

- Escaneo de vulnerabilidades realizado con éxito: 4 puntos
- Correcta implementación de la política de etiquetado: 4 puntos
- Validación exitosa del archivo Docker Compose: 4 puntos
- Implementación adecuada de la red de contenedores: 4 puntos
- Correcta configuración del archivo ".dockerignore": 4 puntos

12 5. Dockerfile y Docker Compose

12.1 5.1. Dockerfile y su uso en la creación de imágenes Docker

12.1.1 Conceptos

Un Dockerfile es un archivo de texto que contiene las instrucciones para construir una imagen Docker. Se puede considerar como una especie de script que automatiza los comandos que normalmente se ejecutarían manualmente para construir una imagen.

12.1.2 Ejemplo

Aquí hay un ejemplo de cómo se ve un Dockerfile:

```
# Dockerfile
FROM python:3.8

ENV PYTHONUNBUFFERED 1

WORKDIR /app

COPY requirements.txt /app/

RUN pip install -r requirements.txt

COPY . /app/
```

Este Dockerfile realiza las siguientes acciones:

- Utiliza la imagen `python:3.8` como base.
- Establece la variable de entorno `PYTHONUNBUFFERED` en `1`.
- Establece `/app` como el directorio de trabajo dentro del contenedor.
- Copia el archivo `requirements.txt` al directorio de trabajo en el contenedor.
- Ejecuta `pip install -r requirements.txt` para instalar las dependencias especificadas en `requirements.txt`.
- Copia el directorio actual (es decir, todos los demás archivos y subdirectorios) al directorio de trabajo en el contenedor.

Para construir una imagen a partir de este Dockerfile, se utiliza el comando `docker build`.

12.1.3 Actividad Práctica

Crea un Dockerfile para una aplicación que utilice la imagen de Docker `node:14` como base, establezca `/usr/src/app` como el directorio de trabajo, copie el archivo `package.json` al directorio de trabajo, ejecute `npm install` para instalar las dependencias y copie el directorio actual al directorio de trabajo.

Resolución de la Actividad Práctica

Aquí está la solución a la actividad práctica:

```
# Dockerfile
FROM node:14

WORKDIR /usr/src/app

COPY package.json .

RUN npm install

COPY . .
```

12.1.4 ¿Qué aprendimos?

- Aprendimos qué es un Dockerfile y cómo se utiliza para automatizar la construcción de imágenes Docker.
- También aprendimos cómo leer y escribir un Dockerfile y cómo construir una imagen a partir de él.

12.2 7.2: Clase: Docker Compose y su uso en el desarrollo de aplicaciones

12.2.1 Conceptos

Docker Compose es una herramienta que permite definir y administrar aplicaciones multi-contenedor con Docker. Utiliza archivos YAML para configurar los servicios de la aplicación, lo que permite iniciar todos los servicios con un solo comando.

En el archivo `docker-compose.yml` proporcionado, se definen dos servicios: `db` y `web`.

- `db`: Este servicio utiliza la imagen de Docker `postgres` para crear un contenedor de base de datos PostgreSQL. La contraseña del usuario `postgres` se establece como `postgres`.

- `web`: Este servicio construye una imagen a partir del Dockerfile en el directorio actual, ejecuta el comando para iniciar el servidor Django, monta el directorio actual en `/app` dentro del contenedor, expone el puerto 8000 y depende del servicio `db`.

12.2.2 Ejemplos

Aquí hay un ejemplo de cómo se ve un archivo `docker-compose.yml`:

```
version: '3.8'

services:
  db:
    image: postgres
    environment:
      POSTGRES_PASSWORD: postgres
  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/app
    ports:
      - "8000:8000"
    depends_on:
      - db
    environment:
      DATABASE_URL: postgres://postgres:postgres@db:5432/postgres
```

Para iniciar los servicios definidos en este archivo, se utiliza el comando `docker-compose up`.

12.2.3 Actividad Práctica

Crea un archivo `docker-compose.yml` para una aplicación que incluya un servicio de base de datos PostgreSQL y un servicio web que utilice la imagen de Docker `nginx`.

Resolución de la Actividad Práctica

Aquí está la solución a la actividad práctica:

```
version: '3.8'

services:
  db:
    image: postgres
    environment:
      POSTGRES_PASSWORD: postgres
  web:
```

```
    image: nginx
    ports:
      - "8080:80"
    depends_on:
      - db
```

Aquí está la solución al problema:

```
version: '3.8'

services:
  db:
    image: postgres
    environment:
      POSTGRES_PASSWORD: postgres
  web:
    image: nginx
    ports:
      - "8080:80"
    depends_on:
      - db
```

En este archivo `docker-compose.yml`, hemos definido dos servicios: `db` y `web`.

- `db`: Este servicio utiliza la imagen de Docker `postgres` para crear un contenedor de base de datos PostgreSQL. La contraseña del usuario `postgres` se establece como `postgres`.
- `web`: Este servicio utiliza la imagen de Docker `nginx` para crear un contenedor web. El puerto 8080 del host se mapea al puerto 80 del contenedor. Este servicio depende del servicio `db`, lo que significa que el servicio `db` se iniciará antes que el servicio `web`.

12.2.4 ¿Qué aprendimos?

- Aprendimos qué es Docker Compose y cómo se utiliza para definir y administrar aplicaciones multi-contenedor.
- También aprendimos cómo leer y escribir un archivo `docker-compose.yml` y cómo iniciar los servicios definidos en él.

13 Actividad Práctica

13.1 Objetivo:

Crear un servicio distribuido con varias réplicas utilizando Docker Swarm.

13.2 Instrucciones:

Crea un servicio llamado “web” que ejecute tres replicas del contenedor Nginx con el siguiente comando:

```
docker service create --replicas 3 -p 8080:80 --name web nginx
```

Despliegue de Aplicaciones con Docker Compose

---

14 Desplegar una Aplicación Compleja con Docker Compose

14.1 Objetivo:

Desplegar una aplicación compuesta por servicios web y de base de datos utilizando Docker Compose.

14.2 Instrucciones:

- Crea un archivo llamado docker-compose.yml con el contenido proporcionado.
- Despliega la aplicación ejecutando el siguiente comando:

```
docker-compose up -d
```

- Utiliza el flag -d para realizar el despliegue en segundo plano.

Resolución de la Actividad Práctica

La resolución de la actividad práctica consiste en seguir las instrucciones proporcionadas para cada práctica y verificar que los servicios se desplieguen correctamente.

14.3 Rubrica de Evaluación:

- Configuración exitosa del clúster de Docker Swarm (Práctica 1): 5 puntos
- Creación correcta del servicio distribuido (Práctica 2): 5 puntos
- Creación del archivo docker-compose.yml: 3 puntos
- Despliegue exitoso de la aplicación con Docker Compose: 7 puntos

15 6. DevContainers

15.1 Conceptos:

15.1.1 DevContainers

DevContainers es una extensión de Visual Studio Code que permite definir entornos de desarrollo en contenedores. Facilita la configuración y el uso de entornos de desarrollo reproducibles.

15.2 Ejemplos:

Configurar un entorno DevContainers para un stack MEAN:

```
// En el archivo .devcontainer/devcontainer.json
{
  "name": "MEAN Stack",
  "dockerFile": "Dockerfile",
  "settings": {},
  "extensions": [
    "dbaeumer.vscode-eslint",
    "ms-azuretools.vscode-docker"
  ],
  "forwardPorts": [3000, 4200, 27017],
  "postCreateCommand": "yarn install && ng serve",
  "appPort": [3000]
}
```

Este archivo de configuración crea un entorno DevContainers utilizando un Dockerfile específico, instala extensiones de Visual Studio Code, reenvía puertos necesarios para el desarrollo MEAN stack, y ejecuta comandos post-creación.

15.3 Práctica:

- Configura un entorno DevContainers para un proyecto Python:
- Asegúrate de que el entorno tiene la extensión “ms-python.python” instalada.

Resolución de la Actividad Práctica

Ejemplo de archivo .devcontainer/devcontainer.json:

```
{
  "name": "Python Project",
  "dockerFile": "Dockerfile.python",
  "extensions": [
    "ms-python.python"
  ],
  "forwardPorts": [8000],
  "postCreateCommand": "pip install -r requirements.txt && python manage.py runserver 0
  "appPort": [8000]
}
```

> **Tip**
>
> Usa DevContainers para garantizar un entorno de desarrollo coherente y fácilmente replicable.

15.4 ¿Qué Aprendimos?

- Comprendemos cómo utilizar DevContainers para crear entornos de desarrollo específicos para proyectos.
- Ahora sabemos cómo configurar extensiones y reenviar puertos en entornos DevContainers.

16 Actividad Práctica

16.1 Objetivo:

Aplicar buenas prácticas y medidas de seguridad al trabajar con Docker.

16.2 Instrucciones:

- Escanea la imagen “alpine:latest” en busca de vulnerabilidades utilizando la herramienta Snyk.
- Implementa una política de etiquetado para las imágenes Docker que siga las mejores prácticas.
- Utiliza la herramienta “docker-compose lint” para verificar la validez del archivo Docker Compose.
- Implementa una red de contenedores y asegúrate de que solo los contenedores necesarios tengan acceso a ella.
- Crea un archivo “.dockerignore” para excluir archivos y directorios innecesarios en la construcción de imágenes Docker.

16.3 Entregables:

- Capturas de pantalla del escaneo de vulnerabilidades.
- Política de etiquetado para imágenes Docker.
- Resultado de la verificación del archivo Docker Compose.
- Documento explicando la implementación de la red de contenedores.
- Archivo “.dockerignore”.

Resolución de la Actividad Práctica

Escaneo de vulnerabilidades:

```
snyk container test alpine:latest
```

Política de etiquetado (ejemplo):

Se etiquetarán las imágenes con el formato “versión-año-mes-día” (ejemplo: 1.0-20230115).

Verificación del archivo Docker Compose:

```
docker-compose config
```

Implementación de una red de contenedores:

```
docker network create my-network
```

Archivo ".dockerignore" (ejemplo):

```
node_modules
.git
.env
```

16.4 Rubrica de Evaluación:

- Escaneo de vulnerabilidades realizado con éxito: 4 puntos
- Correcta implementación de la política de etiquetado: 4 puntos
- Validación exitosa del archivo Docker Compose: 4 puntos
- Implementación adecuada de la red de contenedores: 4 puntos
- Correcta configuración del archivo ".dockerignore": 4 puntos

17 7. Orquestación de Contenedores con Docker Swarm

17.1 Conceptos:

17.1.1 Docker Swarm

Es una herramienta de orquestación de contenedores incorporada en Docker. Permite gestionar y escalar aplicaciones en un clúster de Docker.

17.1.2 Nodos y Servicios

En el contexto de Docker Swarm, los nodos son las máquinas que participan en el clúster y los servicios son las aplicaciones que se ejecutan en esos nodos.

17.2 Ejemplos:

Inicializando un Swarm:

```
docker swarm init
```

Este comando inicia un clúster de Swarm en el nodo actual.

Creando un Servicio:

```
docker service create --replicas 3 -p 8080:80 --name web nginx
```

Aquí estamos creando un servicio llamado “web” que ejecuta tres replicas del contenedor Nginx.

17.3 Práctica:

- Configurar un Clúster de Docker Swarm:
- Inicia un clúster de Docker Swarm con dos nodos.
- Crea un servicio distribuido que ejecute una aplicación de ejemplo en el clúster.

Resolución de la Actividad Práctica

Inicializar el Swarm:

```
docker swarm init --advertise-addr <IP_DEL_NODO>
```

Unirse a un Nodo al Swarm:

```
docker swarm join --token <TOKEN> <IP_DEL_MANAGER>:<PUERTO>
```

Crear un Servicio:

```
docker service create --replicas 3 -p 8080:80 --name web nginx
```

> **Tip**
>
> Utiliza Docker Swarm para mejorar la escalabilidad y la administración de tus aplicaciones en contenedores.

18 Actividad Práctica

18.1 Objetivo:

Configurar un clúster de Docker Swarm con un nodo manager y varios nodos workers.

18.2 Instrucciones:

Inicializa el Swarm en el Nodo Manager ejecutando el siguiente comando:

```
docker swarm init --advertise-addr <IP_DEL_NODO_MANAGER>
```

Sustituye con la dirección IP del nodo que actuará como manager.

Únete a un Nodo al Swarm ejecutando el siguiente comando en cada Nodo Worker:

```
docker swarm join --token <TOKEN_GENERADO> <IP_DEL_NODO_MANAGER>:<PUERTO>
```

Reemplaza con el token proporcionado en el paso anterior y con la dirección IP del nodo manager.