Práctica 4. Empaquetando nuestra aplicación Node.js con Dockerfile

1. Introducción

Hasta ahora hemos utilizado imágenes oficiales (como mongo:7.0) descargadas directamente de Docker Hub. Sin embargo, para desplegar nuestra propia aplicación de Node.js, necesitamos crear nuestra propia **imagen personalizada**.

Un **Dockerfile** es un archivo de texto con instrucciones que Docker lee para construir una imagen automáticamente. Esto nos permite empaquetar nuestro código, nuestras dependencias (node_modules) y la configuración necesaria en un solo "paquete" listo para funcionar en cualquier sitio.

En esta práctica vamos a desplegar en un **VPS** una **aplicación Node.js conectada a MongoDB**, usando:

- Un `Dockerfile` para construir la imagen de la aplicación.
- Dos contenedores levantados manualmente con `docker run` (MongoDB y Node.js)
- Una **red Docker** para que se vean por nombre.
- Un volumen para persistencia de MongoDB.
- Un script de inicialización (como en la Práctica 3) para poblar la base de datos automáticamente.

Proyecto a desplegar: libros (AULES).

2. Conceptos clave de Dockerfile

Para esta práctica utilizaremos las siguientes instrucciones:

- **FROM**: define la imagen base (ej. node:20).
- **WORKDIR**: establece el directorio de trabajo dentro del contenedor.
- **COPY**: copia archivos desde el VPS al contenedor.
- **RUN**: ejecuta comandos durante la construcción (ej. npm install).
- **EXPOSE**: su función es documentar el puerto interno en el que escucha la aplicación dentro del contenedor. Es información para el usuario, no una configuración de red. Para que dicho puerto sea accesible desde el exterior, es necesario utilizar la opción `-p` al arrancar el contenedor.
- **ENV**: define variables de entorno (como la URI de conexión a Mongo).
- **CMD**: el comando que arrancará nuestra aplicación cuando el contenedor se inicie.

3. Preparación de la aplicación en el VPS

Antes de construir la imagen Docker de la aplicación Node.js, es necesario que el código esté en el VPS. Existen varias formas válidas de hacerlo.

La aplicación ya está configurada para obtener desde variables de entorno tanto la URI de MongoDB como el resto de parámetros sensibles.

El archivo `.env` debe existir en el servidor, pero no debe subirse a Git ni copiarse dentro de la imagen Docker. Es una buena práctica crear el `.env` directamente en el VPS. Para ello se usa `.gitignore` y `.dockerignore`, veremos su contenido en apartados posteriores.

3.1 Usando GitHub y git clone

El proyecto se sube previamente a un repositorio público en GitHub y después se descarga en el VPS usando Git.

1. Crea el archivo `.gitignore` en la raíz del proyecto de libros con el siguiente contenido:

```
node_modules/
.env
```

2. Sube el proyecto a GitHub desde el equipo local.
3. Conéctate al VPS por SSH.
4. Instala Git y configura tu usuario de GitHub, si todavía no lo has hecho (ver práctica 1).
5. Clona el repositorio directamente en `/home/debian` con el comando: `git clone https://URL_REPOSITORIO`. Se creará la carpeta del proyecto automáticamente en ese directorio.

3.2 Usando VS Code con Remote-SSH

Recuerda que en la práctica 1 se configuró VSCode para trabajar directamente sobre el servidor. Gracias a eso, el VPS se comporta como si fuera una carpeta local.

Hay dos formas de subir la aplicación con VSCode.

3.2.1 Clonar desde VS Code

1. Conéctate al VPS con **Remote - SSH**
2. Sitúate en una carpeta del VPS, por ejemplo: `/home/debian/`
3. Asegúrate de haber creado `.gitignore`
4. Abre un terminal integrado
5. Clona el repositorio previamente subido: `git clone https://URL_REPOSITORIO`

3.2.2 Copiar archivos directamente

1. Conéctate al VPS con **Remote - SSH**
2. Sitúate en una carpeta del VPS, por ejemplo: `/home/debian/`
3. Arrastra desde tu ordenador la carpeta de la aplicación Node.js al explorador de VSCode

4. El resultado será: `home/debian/libros`

Antes de arrastrar, revisa que la carpeta solo contenga el código necesario. Si arrastras toda la carpeta del proyecto sin revisar, puedes copiar cosas que NO deben ir al VPS, por ejemplo:

- node_modules/
- .env

> **NOTA:** Cuando trabajamos con Remote-SSH, arrastrar una carpeta al explorador de VS Code copia los archivos al servidor, pero no crea control de versiones ni sincronización automática.

3.3 Usando un cliente FTP/SFTP (FileZilla)

Se copian los archivos desde el equipo local al VPS usando un cliente gráfico como FileZilla, normalmente mediante SFTP (sobre SSH).

1. Abre FileZilla.
2. Conéctate al VPS usando:
   - *Host*: IP del VPS
   - *Usuario*: debian
   - *Protocolo*: SFTP
3. Arrastrar la carpeta del proyecto al VPS, por ejemplo a: `/home/debian`

Esta es la opción **menos recomendada**. En entornos reales, no es habitual subir proyectos por FTP.

4. Gestión de la configuración con variables de entorno

Como hemos comentado anteriormente, en aplicaciones reales, **la configuración no debe estar escrita directamente en el código**. Datos como direcciones de bases de datos, usuarios o contraseñas pueden cambiar según el entorno (desarrollo, pruebas, producción).

Lo recomendable es utilizar **variables de entorno** y un **fichero** `.env` en la carpeta raíz de la aplicación para definir la configuración de la aplicación con la información sensible.

4.1 El archivo .env

Desde el VPS, sitúate en el directorio del proyecto `/home/debian/libros` y crea el siguiente archivo `.env`:

```
# Puerto interno en el que escucha la app Node.js
PORT=3000

# Clave secreta para las sesiones de Express
SESSION_SECRET=secretoNode

# Configuración de MongoDB
MONGO_ROOT_USER=adminlibros
MONGO_ROOT_PASS=adminlibros
MONGO_DB_NAME=libros

# Credenciales de la aplicacion (opcional)
APP_USER=librosuser
APP_PASS=librosuser

# URI de conexión interna
# (usando el nombre del contenedor MongoDB como host)
MONGO_URI=mongodb://librosuser:librosuser@libros-mongo-db:27017/libros?authSource=lib
```

Las variables `MONGO_ROOT_USER` y `MONGO_ROOT_PASS` solo son utilizadas por el contenedor de MongoDB durante su inicialización.

La variable `MONGO_URI` define la URI de conexión a MongoDB que usará la aplicación Node.js para conectarse a la base de datos desde dentro de un contenedor Docker.

En general, tiene esta estructura:

```
mongodb://usuario:password@host:puerto/baseDatos?authSource=baseAutenticacion
```

Para nuestro caso concreto, contiene este valor:

```
mongodb://librosuser:librosuser@libros-mongo-db:27017/libros?authSource=libros
```

Veamos el significado de cada elemento:

- `librosuser:librosuser`: son las credenciales del usuario de la aplicación.
  - NO es el root de MongoDB
  - Se crea en el script `init-libros.js`, que veremos más adelante.
  - Tiene permisos solo sobre la base de datos libros
- `@`: separador entre credenciales y servidor.
- `libros-mongo-db`: es el nombre del contenedor MongoDB.

    - No es localhost

- `:27017` : es el puerto del servicio MongoDB dentro del contenedor.
- `/libros` : nombre de la base de datos a la que se conecta la aplicación.

    - Si no existe, MongoDB la crea al primer uso
    - Coincide con la usada en el script de inicialización

- `?authSource=libros` : indica dónde MongoDB debe validar el usuario. Basicamente significa que "*el usuario librosuser se autentica contra la base de datos libros*". Es necesario porque:

    - El usuario no es root
    - No está definido en la base admin

5. Creación del Dockerfile y .dockerignore (imagen de Node.js)

Una vez ya hemos subido la carpeta de la aplicación Node.js al VPS, ha llegado el momento de crear la imagen en la que estará nuestra aplicación para posteriormente crear un contenedor Docker con ella.

1. Crea el siguiente archivo llamado `Dockerfile` dentro de la carpeta de la aplicación (`/home/debian/libros`):

```
# 1. Imagen base de Node.js
FROM node:20

# 2. Directorio de trabajo dentro del contenedor
WORKDIR /libros

# 3. Copiar archivos de dependencias
COPY package*.json ./

# 4. Instalar dependencias
RUN npm install

# 5. Copiar el resto del código
COPY . .

# 6. Exponer el puerto que usa Express (ej. 3000)
EXPOSE 3000

# 7. Comando de arranque
CMD ["node", "index.js"]
```

2. Crea también un archivo `.dockerignore` en la misma carpeta raíz de la aplicación para evitar copiar carpetas pesadas o innecesarias.

```
node_modules
.env
```

En este caso, incluimos `node_modules` porque se generará dentro de la imagen y `.env` porque no es recomendable que las contraseñas se queden dentro de la imagen de Docker, sino que se inyectarán en el contenedor al arrancar con `--env-file`, como veremos más adelante.

6. Construir la imagen de la aplicacion Node.js

Una vez tenemos creado nuestro `Dockerfile`, pasamos a crear la imagen de la aplicación Node.js.

Desde `/home/debian/libros`, ejecuta el siguiente comando:

```
docker build -t libros-node .
```

En este comando `build` localiza el fichero `Dockerfile` en el directorio actual, lee y ejecuta sus instrucciones de arriba a abajo, crea una imagen local llamada `libros-node` con la aplicación de libros y sus dependencias.

Puedes comprobar el resultado con el comando: `docker images`

El siguiente paso es crear una red Docker para que nuestros contenedores puedan comunicarse. Después de eso, ya podremos levantar los contenedores, primero siempre el de MongoDB y después el de nuestra aplicación.

7. Crear una red Docker

En Docker, cada contenedor vive "aislado" (como si fuese su propio mini ordenador). Para que dos contenedores se comuniquen **por nombre** y de forma sencilla, los conectamos a una **misma red Docker**.

```
docker network create libros-red
```

8. Preparar el script de inicialización de MongoDB

Antes de crear el contenedor con MongoDB, vamos a crear un script para poblar la base de datos.

En `/home/debian/mongo-init` (carpeta creada en la Práctica 3), crea el archivo:

```
nano /home/debian/mongo-init/init-libros.js
```

Un posible contenido de dicho script podría ser el siguiente:

```
// init-libros.js
// Inicialización automática para el proyecto Node.js "libros"

// 1) Seleccionar / crear la base de datos
const dbLibros = db.getSiblingDB("libros");

// 2) Crear usuario específico de la aplicación
dbLibros.createUser({
  user: "librosuser",
  pwd: "librosuser",
  roles: [
    { role: "readWrite", db: "libros" }
  ]
});

// 3) Crear colecciones
dbLibros.createCollection("libros");
dbLibros.createCollection("autores");

// 4) Insertar datos de ejemplo coherentes con los modelos

dbLibros.autores.insertMany([
  {
    nombre: "Miguel de Cervantes",
    anyoNacimiento: 1547
  },
  {
    nombre: "Gabriel García Márquez",
    anyoNacimiento: 1927
  }
]);

dbLibros.libros.insertMany([
  {
    titulo: "Don Quijote de la Mancha",
    editorial: "Francisco de Robles",
    precio: 19.95,
    portada: "quijote.jpg"
  },
  {
    titulo: "Cien años de soledad",
    editorial: "Sudamericana",
    precio: 24.50,
    portada: "cien-anyos.jpg"
  }
]);

// 5) Mensaje visible en logs para verificación
print(">>> MongoDB init OK: BD 'libros', colecciones y datos creados");
```

Recuerda que este script se ejecuta automáticamente cuando Mongo arranca por primera vez con un volumen vacío.

9. Crear volumen Docker para persistencia de MongoDB

MongoDB guarda sus datos en `/data/db`. Si no usamos un volumen, al borrar el contenedor se perdería la base de datos.

Para crear el volumen, ejecutaremos el siguiente comando:

```
docker volume create libros-data
```

Si queremos evitar efectos de ejecuciones anteriores, y asegurar que se ejecuta el script de inicializacion, podemos borrar el volumen con el siguiente comando:

```
docker volume rm libros-data 2>/dev/null
```

> **NOTA:** La redirección `2>/dev/null` se utiliza para descartar mensajes de error no relevantes, como el intento de borrar un volumen inexistente.

10. Lanzar MongoDB (primer contenedor)

Ahora vamos a lanzar un contenedor con MongoDB que use:

- El volumen `libros-data` para los datos, y
- La carpeta `/home/debian/mongo-init` como script de arranque

Es importante **arrancar MongoDB antes que la aplicación**.

El siguiente comando levanta el contenedor MongoDB definiendo explícitamente el usuario administrador (root) de la base de datos.

```
docker run -d \
  --name libros-mongo-db \
  --network libros-red \
  --restart unless-stopped \
  -v libros-data:/data/db \
  -v /home/debian/mongo-init:/docker-entrypoint-initdb.d:ro \
  -e MONGO_INITDB_ROOT_USERNAME=adminlibros \
  -e MONGO_INITDB_ROOT_PASSWORD=adminlibros \
  mongo:7.0
```

Explicación de cada opción:

- `-d` : ejecuta en segundo plano.
- `--name libros-mongo-db` : nombre del contenedor; será el host al que se conecte la aplicacion.
- `--network libros-red` : lo mete en la red compartida.
- `--restart unless-stopped` : reinicio automático tras reiniciar el VPS.
- `-v libros-data:/data/db` : persistencia de datos.
- `-v /home/debian/mongo-init:/docker-entrypoint-initdb.d:ro` : scripts init (solo lectura).
- `-e MONGO_INITDB_ROOT_USERNAME/PASSWORD` : crea el usuario administrador (root) de Mongo (solo para administración y para poder inicializar).
- `mongo:7.0` : imagen usada.

Observa que no usamos `-p` al arrancar MongoDB. Cuando creamos una red Docker y conectamos varios contenedores a ella Docker proporciona:

- Resolución de nombres (por nombre de contenedor)
- Comunicación directa entre contenedores
- Acceso a los puertos internos de cada servicio

Esto significa que:

- MongoDB escucha en su puerto interno (27017)
- Node.js puede conectarse a ese puerto sin exponerlo al exterior
- Basta con usar el nombre del contenedor como host

Solo publicamos puertos cuando el servicio debe ser accesible desde fuera de Docker, como es el caso del contenedor con la aplicacion de Node.js, que debe estar accesible desde el navegador.

Por otro lado, si queremos levantar el contenedor de forma limpia, utilizando las variables de entorno definidas anteriormente, en lugar de usar múltiples flags `-e` en el comando `docker run`, utilizaremos el flag `--env-file` para lanzar los contenedores de forma limpia con este otro comando:

```
docker run -d --name libros-mongo-db \
  --network libros-red \
  -v libros-data:/data/db \
  -v /home/debian/mongo-init:/docker-entrypoint-initdb.d:ro \
  --env-file .env \
  mongo:7.0
```

- `--env-file .env` carga las variables de entorno dentro del contenedor.

Comprobación de la inicialización

```
docker logs libros-mongo-db
```

Aquí debe aparecer el mensaje del `print(...)` del script, indicando que se ejecutó correctamente.

11. Lanzar la aplicacion Node.js (segundo contenedor)

```
docker run -d \
  --name libros-web \
  --network libros-red \
  --restart unless-stopped \
  -p 80:3000 \
  -e MONGO_URI="mongodb://librosuser:librosuser@libros-mongo-db:27017/libros?authSour
  libros-node
```

Explicación:

- `-p 80:3000` : el servidor escuchará en el puerto 80 del VPS y redirige al 3000 interno del contenedor.
- `-e MONGO_URI=...` : inyectamos la URI de conexión a Mongo.
- Host `libros-mongo-db` : es el nombre del contenedor de Mongo en la red **libros-red**. La aplicación NO debe usar localhost, porque dentro del contenedor localhost sería el propio contenedor de Node, no el de MongoDB.

Observa que en este caso sí usamos la opción `-p` . A diferencia del contenedor de MongoDB, la aplicación Node.js debe ser accesible desde fuera de Docker, concretamente desde el navegador web del usuario.

Con la opción `-p 80:3000` estamos indicando que el puerto 80 del VPS (acceso desde Internet) se redirige al puerto 3000 del contenedor (donde escucha Express). De esta forma, las peticiones que llegan al VPS por el puerto 80 se envían al contenedor de Node.js, permitiendo acceder a la aplicación desde el navegador.

En resumen, solo publicamos puertos cuando un servicio debe ser accesible desde el exterior. Para la comunicación interna entre contenedores dentro de la misma red Docker, no es necesario publicar ningún

puerto.

Igual que en el caso del contenedor MongoDB, si queremos levantar el contenedor de una forma más limpia, usaremos este otro comando:

```
docker run -d --name libros-web \
  --network libros-red \
  -p 80:3000 \
  --env-file .env \
  libros-node
```

12. Comprobación del despliegue

1. Ver contenedores levantados:

```
docker ps
```

2. Ver logs de la aplicación:

```
docker logs libros-web
```

3. Abrir en el navegador http://IP_PUBLICA_DEL_VPS

13. Entrega

Documenta el proceso incluyendo:

- Captura del archivo `Dockerfile` y explicación de cada línea.
- Captura del comando `docker build` finalizando correctamente.
- Captura de `docker ps` donde se vean ambos contenedores (libros-mongo-db y libros-web) en ejecución.
- Captura de la aplicación funcionando en el navegador (IP del VPS) mostrando datos que vienen de la base de datos.