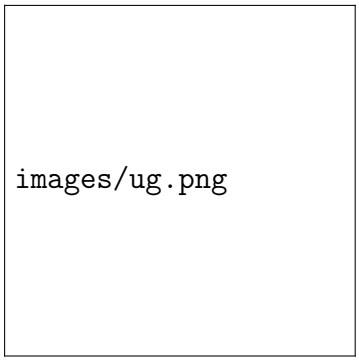

Universidad de Guanajuato

Materia: Bases de datos no relacionales

Carrera: Licenciatura en Ingeniería de Datos e Inteligencia Artificial

Práctica 1:Montar API rest en DOCKER

Alumno: José Adrián Rodríguez González

NUA: 148661

Profesor: Dr. Yahir

17 de septiembre de 2025

Índice general

Capítulo 1

Introducción

En esta práctica se desarrolló un sistema de CRUD para una base de datos en una universidad, en la cual registrará estudiantes. Se usó Docker para orquestar los contenedores donde vivirá el servidor y la base de datos. Se utilizó una base de datos del tipo no relacional basada en documentos, mongodb. Y las tecnologías utilizadas para el desarrollo del servidor fueron javascript con node y el framework de express para hacer los endpoints.

Capítulo 2

Desarrollo

Para esto se desarrollaron dos elementos, la parte de la base de datos y la parte del servidor. La base de datos como se mencionó en la introducción, fue mongodb. Esta base de datos va a estar dentro de un contenedor de docker. Sin embargo, tanto este contenedor como el de la API vivirán en contenedores temporales, por lo que se debe de hacer un volumen que permita la persistencia de los datos, sobre todo en la base de datos. Así que inicialmente podemos crear nuestro volumen por medio de los siguientes comandos

```
docker volume create mongo_db
docker network create practica_1
docker run --rm -d --name base_universidad --network practia_1 -p \
27017:27017 -v mongo_db:/data/db \
-e MONGO_INITDB_ROOT_USERNAME=root -e MONGO_INITDB_ROOT_PASSWORD=root \
-e MONGO_INITDB_DATABASE=universidad mongo:6.0
```

El primer comando sirve para crear un volumen con ese nombre, mongo_db, después se crea una red, por medio del segundo comando y con el nombre *practia_1*. Los volúmenes permiten que si el contenedor es temporal, la información almacenada se queda en un sector de memoria, y si se remueve el contenedor, ninguna información se pierde. Esto es práctico debido que evitamos tener almacenado contenedores inactivos. Por la otra parte, el segundo comando crea un switch virtual, esto implica que le asigna direcciones ip a cada uno de los elementos contenidos dentro de ese switch o que tengan esa conexión, a su vez de que tiene su propio enrutador. Solamente que ese switch se hace en base el localhost de nuestra máquina, por lo que si se desea por otro dispositivo externo, se debe primeramente,se deben de exponer los puertos de la comunicación principal, de esta manera si nos queremos conectar a esa dirección, podemos realizar mediante curl y colocar la ip que posee el dispositivo que monta los contenedores. Esta IP está asignada a un conexión de red, por lo que el dispositivo que desee conectarse al otro dispositivo, debe estar en la misma red,(si se desa realizar por ese método)

Tal que ahora tenemos el volumen y nuestro switch virtual, ahora se puede crear una base de datos. El tercer comando monta un contenedor temporal mediante rm, y el proceso de montado es d, por lo tanto, los logs se ven en segundo plano, con el nombre de base_universidad, se le indica el nombre del switch virtual que previamente creamos, se desvelan los puertos para la comunicación 27017, y se coloca el nombre del volumen creado al cual, vivirá en el contenedor en la ruta descrita después de los ":".

Se crean las variables de entorno que es el nombre de usuario, contraseña y el nombre de una base de datos inicial, finalmente la imagen que requerimos que es mongo en versión 6.0.

Para realizar el segundo se va a crear una imagén de dockerfile.

```
FROM node:24-alpine
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm ci --only=production
COPY . .
EXPOSE 3000
CMD [ "node","index.js" ]
```

Esta imagen primermamenmte necesitara la versión 24 de node, después configura como la raíz de trabajo */usr/src/app*. Se copian y pegan los archivos package*.json,(el asterisco es un comodín, por lo que no importa que es lo que siga o no siga en el texto, solo con ese formato de inicio es suficiente para identificar cuales son los archivos de nuestro interés). y los pega en la raíz de nuestro directorio de trabajo. Se ejectua una instalación de aquellas dependecias que son de producción, esto quita aquellas que son de desarrollo. Después copia todo el proyecto a la raíz del directorio. Se expone el puerto 3000 y el comando por defecto que se ejecuta se determinar por CMD. Este comando es más que la inicialización del proyecto por medio de node y el archivo de index.Finalmente se monta la imagen con el siguiente comando

```
docker build -t mi-api .
```

Se construye y se nombra por medio *-t mi-api*, y busca la dirección en donde está tanto el docker file como los archivos que copiará al indicarle que es en esa raíz. Con esta imagen se puede realizar y levantar el contenedor con el siguiente comando

```
docker run --rm -d --name api_universidad --network practia_1 \
-p 3000:3000 -v $(pwd):/usr/src/app -v /usr/src/app/node_modules \
--env-file .env mi-api:latest
```

Con esté comando podemos crear y levatnar el contenedor donde vendrá la API. Mediante rm se hace temporal, y de igual manera el log se hace en segundo plano por d. Se le nombra *api_ universidad* y se sube al switch que también se conectó nuestra base de datos. Después se levantan los puertos y (esta parte es opcional, ya que se usa cuando todavía está en desarrollo, si se pone en producción, solamente se debe de eliminar está sección). Al colocarle -v creamos un volumen que estará en nuestro directorio del proyecto, además de las dependencias. Esto con el fin de que si se realiza un cambio, se pueda ver reflejado sin tener que para el contenedor. En producción por lo general se omite. Se envían las variables de entorno, puede ser manualmente o enviarlo por un archivo .env. Finalmente se pide la imagen a usar, ya previamente construida. Una vez que ya están los dos, y el código fue desarrollado de manera correcta, se puede proceder a realizar pruebas. Pero antes de llegar al código se debe de explicar que hace.

2.1. Código

Como se mencionó en la introducción, se realizó un código en javascript con la arquitectura de modelo controlador. Durante el desarrollo de los contenedores, se mencionó que es necesario unas dependencias, y es que en un principo, para realizar el proyecto, se debió de haber instalado node y npm para poder realizar pruebas iniciales. Para ello con el comando *npm init*, se inicia un proyecto y con *npm init -y*, se pueden omitir los pasos de inicialización. Las dependencias necesarias son *express* y *mongoose*. El primero es un framework que permite el desarrollo de servidores y apis de manera más sencilla que realizarlo manualmente con node, y mongoose es un ORM que permite interactuar a javascript con mongo de una forma más simple. Los ORM por lo general ya tienen su abstracción de objeto para evitar que se deba de escribir código del lenguaje de la base de datos. Nuestro archivo principal será *index.js*, que contiene todas las llamadas para la base y la definición de endpoints. La raíz del proyecto está construido tal que

```
config/
controller/
Dockerfile
index.js
.env
model/
node_modules/
package-lock.json
package.json
reamde.md
student.service.js
```

Aquellos nombre que terminan con /, son rutas. El primero a observar es el Dockerfile, que es la imagen que montaremos al contenedor. El segundo es la raíz del proyecto, el tercero el archivo que contienen las variables de entorno.Los que terminan en json en esta caso, indican cuales dependencias se están utilizando, un readme con un pequeño resumen del proyecto y student.servce.js es un archivo que contiene los elementos o funciones que se usarán para la interacción de la base de datos, más adelante se explica en detalle este archivo El primer archivo a observar para la interacción de la base de datos es la conexión. Aquí se importa el paquete de mongoose y se crean dos funciones, una que permite la conexión y la segunda la desconexión de manera asíncrona. En la conexión es necesario pasarle las variables de entorno y crear nuestra url que permitirá la conexión a mongo. Por medio de un try catch, capturamos si hubieron errores en la conexión, ya que se intenta conectar y si fue correcto, imprime a que usuario se conectó de la base de datos. Si no, solo que no se pudo conectar. Finalmente una función que permite la des-conexión. El uso de funciones asíncronas es debido a que muchos de los procesos que se realizarán si no son exitosos, no queremos que nuestro programa concluya, quizás que brinde un mensaje de error, más no la detención del programa. Para ello, se utilizan las funciones asíncronas, o también por si una de las funciones tarda más, que la aplicación completa no quede congelada. Además de que las implementaciones desarrollados por este ORM son asíncronas, así que forzosamente las funciones que sean asíncronas deben de ser ejecutadas y contenidas en funciones que sean asíncronas. El ejemplo del código es el siguiente

Listing 2.1: Conexión a MongoDB con Mongoose

```
import mongoose from "mongoose";

export const connect = async () => {
    const {MONGO_USER, MONGO_PASS, MONGO_HOST, MONGO_PORT,
       MONGO_DB} = process.env;
    const url = `mongodb://${MONGO_USER}:${encodeURIComponent(
       MONGO_PASS)}@${MONGO_HOST}:${MONGO_PORT}/${MONGO_DB}?
       authSource=admin`;
    console.log(url)
    try {
        await mongoose.connect(url);
        console.log(`Conectado: ${MONGO_USER}`);
    } catch (error) {
        console.log("Error, no se pudo conectar");
    }
}

export const disconnect = async () => {
    await mongoose.disconnect();
    console.log("Se desconecto")
}
```

ESte archivo se encuentra en la carpeta config con el nombre de db.js Después, pasamos a la sección de modelo, que define un schema, que es lo más parecido a las tablas en bases de datos relacionales. Contendrá un nombre que es un string, un variable requerida y que se recorten espacios en blanco. Una fecah de nacimiento del tipo Date siendo una variable requerida, y un correco electrónico único, en minúsculas y requerido. Finalmente una marca de tiempo de cuando se creo ese dato. Este schema se le denominará student. Este objeto tendrá métodos asociados, sin embargo solo utilizaremos aquellos que sean para