

Elaborado por: Alejandro Viveros Sánchez
Fecha de Entrega: 30 septiembre 2020
Título: Programación de una API

Tabla de contenido

Objetivo:	3
Marco teórico	3
API.....	3
Mongo DB	3
Node JS	3
Docker.....	3
Justificación de tecnologías:	3
Realización del programa:	4
Pruebas y Resultados:.....	12
Problemas y errores:	14
Conclusiones:.....	16
Referencias:	16

Objetivo:

La siguiente practica está enfocada a identificar las habilidades de interactuar con APIs, contenedores, microservicios y distintos lenguajes de programación.

Marco teórico:

API

Api es la abreviatura de Application Programming Interfaces. Se trata de un conjunto de definiciones y protocolos que se utilizan para desarrollar e integrar el software de las aplicaciones, permitiendo la comunicación entre dos aplicaciones de software a través de un conjunto de reglas.

O más formalmente se podría especificar como un módulo de un software que se comunica o interactúa con otro para cumplir con una o varias funciones.

Lo que nos permite las API son facilitarles el trabajo a los desarrolladores.

Mongo DB

Es una base de datos orientada a documentos, lo que significa que no guardara en registros sino en documentos en BSON, que es una representación binaria de JSON.

Un gran uso de mongo DB son para las aplicaciones CRUD o de desarrollos de web actuales.

Un impedimento de Mongo es que no existen las transacciones ni tampoco los joins.

Node JS

Es un entorno en tiempo de ejecución multiplataforma, para la capa del servidor basado en Javascript.

Uno de sus principales usos es en los servidores web. Algunas incorporaciones que cuenta, son los módulos.

Docker

Su propósito es la creación de contenedores, los contenedores no buscan mas que la capacidad de ejecutar varios procesos y aplicaciones por separado para para hacer un mejor uso de la infraestructura

Docker provee los que será una imagen que nos permite compartir la aplicación con todas las dependencias en varios entornos.

Justificación de tecnologías:

El uso de NodeJS como lenguaje para el desarrollo de la API, es que esta basado en gran parte en Javascript, siendo este uno de los lenguajes más conocidos, además de contar con algunas otras características como la incorporación de Node Package Manager(NPM), que permite expandir el código añadiendo el código por medio de módulos, también se cuenta con un alto rendimiento en proyectos donde necesitamos ejecución en tiempo real.

MongoDB tenemos la ventaja de que se adecua la perfección con JavaScript, además de que es ideal para entornos con pocos recursos de computación.

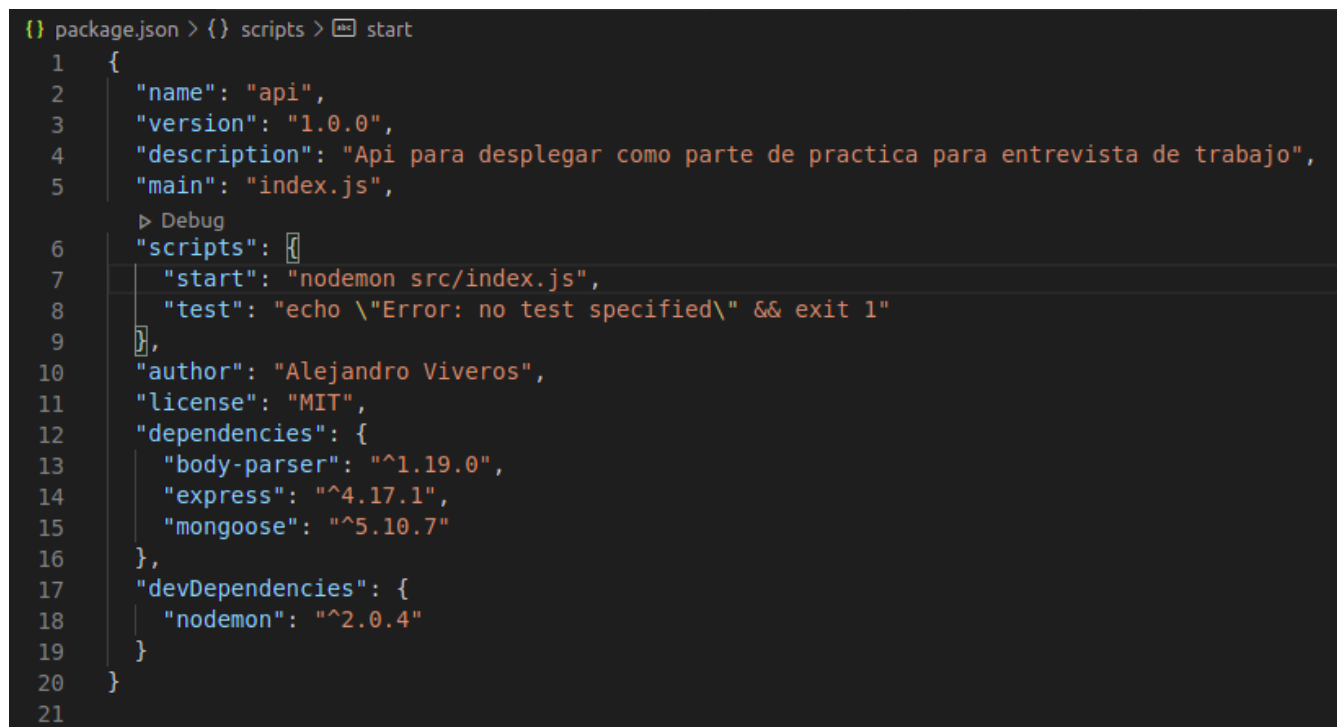
Docker es una gran herramienta, ya que ofrece pariedad, lo que significa que las imágenes se ejecutan igual donde sea, esto implica quitar la necesidad de configurar entornos, depurar problemas. Con ello obtenemos una API capaz de ejecutarse en cualquier computador.

Realización del programa:

Como parte inicial en el desarrollo de la Api, se optó por desarrollarla en Node js, esto con el fin de poder conectarla con una base de datos de MongoDB.

En un principio se desarrolló la una carpeta llamada “API”, donde ahí va a contener todos nuestros archivos relacionados con la API, desde los archivos de Node, así como sus dependencias, los archivos Docker, entre otras.

Así que primeramente nos colocamos en la carpeta “API”, y por medio del comando : **npm init**. Se generará un archivo json, el cual nos va a servir para ver las dependencias que vamos a instalar en Docker. El archivo antes mencionado luce de la siguiente manera:



```
{ } package.json > { } scripts > start
1  {
2    "name": "api",
3    "version": "1.0.0",
4    "description": "Api para desplegar como parte de practica para entrevista de trabajo",
5    "main": "index.js",
6    "scripts": {
7      "start": "nodemon src/index.js",
8      "test": "echo \"Error: no test specified\" && exit 1"
9    },
10   "author": "Alejandro Viveros",
11   "license": "MIT",
12   "dependencies": {
13     "body-parser": "^1.19.0",
14     "express": "^4.17.1",
15     "mongoose": "^5.10.7"
16   },
17   "devDependencies": {
18     "nodemon": "^2.0.4"
19   }
20 }
21
```

Figura 1: package.json.

Se describir algunos requerimientos del json, así como se descargó el archivo, además de su razón en el proyecto y su importancia:

- **body-parser**: Es un requerimiento necesario, para obtener los datos de los JSON para realizar la comunicación entre el cliente y el servidor. De una manera coloquial se diría que estamos “parsenado ” el mensaje a enviar. Este requerimiento lo podemos descargar por medio del comando: **“npm install -save body-parser”**
- **express**: Este Framework nos permite la comunicación cliente-servidor. Esto nos ayudara a crear nuestro servidor, para poder instalar este requerimiento solo debemos de introducir el siguiente comando: **“npm install express -S”**
- **mongoose**: Es un requerimiento necesario para conectarnos con la base de datos en MongoDB. Como observaremos más adelante, vamos a definir lo que es el “Schema” para definir una lista de propiedades de nuestro mensaje. Para instalar este requerimiento es necesario de introducir el siguiente comando: **“npm i -S mongoose”**

- nodemon: Es un requerimiento que nos ayudara en las ejecuciones de nuestra API. Anteriormente para ejecutar nuestra API, debemos de introducir el comando: `node index.js`, con ello se ejecutaba la API, ahora con mongoose, cada vez que guardemos un cambio en algún fichero, se ejecutará nuestra API. Para lograr esto también debemos de añadir en “package.json” en la parte de “scripts” la línea `start: nodemon src/index.js`, que nos dice que cada vez que se ejecute la API, con nodemon ejecutará el fichero “src/index.js”. La manera de instalar nodemon en nuestra API es por medio del comando: `npm i -D nodemon`. Ahora para ejecutar la API solo es necesario el comando: `npm start`.

Otro importante archivo que nos ayudara cuando tengamos que “Dockerizar” nuestra API, es el fichero “package-lock.json”, la importancia de este proyecto es que le diría a Docker que dependencias instalar así como sus versiones. Este fichero, tiene un aspecto similar a la siguiente captura:

```
{ package-lock.json > {} dependencies > {} util-deprecate > resolved
1 {
2   "name": "api",
3   "version": "1.0.0",
4   "lockfileVersion": 1,
5   "requires": true,
6   "dependencies": {
7     "@sindresorhus/is": {
8       "version": "0.14.0",
9       "resolved": "https://registry.npmjs.org/@sindresorhus/is/-/is-0.14.0.tgz",
10      "integrity": "sha512-9NET910DNaIPngYnLLPeg+0gzqsi9uM4mSboU5y6p8S5DzMTVEsJZrawi+BoDNUVBa2DhJqQYUFvMDfgU062LQ==",
11      "dev": true
12    },
13    "@szmarczak/http-timer": {
14      "version": "1.1.2",
15      "resolved": "https://registry.npmjs.org/@szmarczak/http-timer/-/http-timer-1.1.2.tgz",
16      "integrity": "sha512-XIB2XbzHTN6ieIjfIMV9hlVcfPU26s2vafYWQcZHWXHX0xiaRZYEDKEwdl129Zyg50+foYV2jCgtrqSA6qNuNSA==",
17      "dev": true,
18      "requires": {
19        "defer-to-connect": "^1.0.1"
20      }
21    },
22  },
23 }
```

Figura 2: Fragmento del fichero “package-lock.json”.

Ya definida las dependencias y requerimientos. Creamos una carpeta llamada “src”, que es donde ira todo lo relacionado con el Código de la API.

Los pasos que se siguieron para la elaboración de la API, fue inicialmente crear un solo fichero para nuestra API en general, este fichero se llamaba “index.js”. Este fichero se definían desde los puertos de las bases de datos, hasta las ruta de la API, lo cual lo hacía muy ineficiente y poco escalable. Por ende, se decide “refactorizar” que significa hacer más escalable la API, poniendo en archivos diferentes las funcionalidades de la API. Refactorizando la API, quedo establecido los ficheros como se muestran en la siguiente captura:

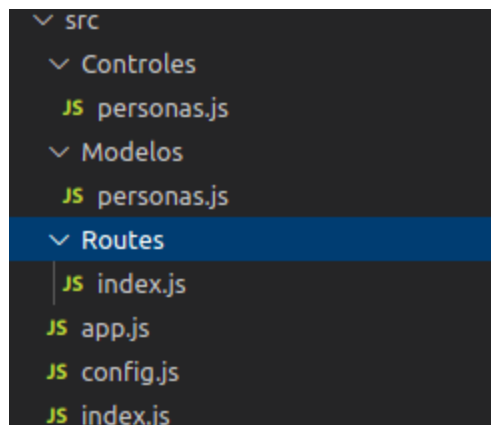


Figura 3: Ruta del código de la API

Algo importante en cuanto a la “refactorización” es que solo se hizo hasta que la API ya estaba funcionando correctamente.

Describiendo con atención los archivos descritos en la Figura3, se tiene:

- Controles/personas.js: En este fichero creamos las funciones que representaran el funcionamiento de nuestra API. Dentro de estas funciones definimos mensajes de error, en caso de que haya problema al conectarse al servidor, así como los mensajes de respuesta en caso de que la función haya funcionado correctamente. Por mostrar como ejemplo el funcionamiento de una de estas funciones, se muestra la siguiente figura:

```
//Creamos funcion get que nos permite obtener la informacion de una persona por el ID
function getPersona(req, res) {
  //variable del id, se obtiene de la URL
  let personaId = req.params.personaId

  //Buscamos que exista el id correspondiente
  Persona.findById(personaId, (err, persona) => {
    //Tramos los posibles errores
    if (err) return res.status(500).send({ message: 'Error al realizar la peticion' })
    if (!persona) return res.status(400).send({ message: 'La persona no existe' })

    //regresamos el producto
    res.status(200).send({ persona })
  })
}
```

Figura 4: Captura de la funcion getPersona() del fichero “Controles/personas.js”.

Las funciones que se crearon en este fichero fueron las siguientes:

1. getPersona .- Regresaba los datos de una persona, por medio del Id que se ingresaba en la URL.
 2. getPersonas .- Regresaba los datos de todas las personas que estaban en la base de datos, no hay necesidad de ingresar Id.
 3. savePersona .- Permite ingresar una nueva persona en la base de datos, solo era necesario ingresar todos los datos, especificados en el esquema.
 4. updatePersona .- Permite modificar los datos de una persona, solo se necesita ingresar el Id en la Url, y manda los datos que se quieren modificar
 5. deletePersona .- Permite eliminar un usuario de la base de datos, solo es necesario ingresar el Id de la persona que se desea eliminar.
- Modelos/personas.js: se define el modelo del documento para la base de datos de MongoDB, con ello generamos un esquema, para así ser usado en la Api. De la siguiente manera quedo definido el esquema:

```
//Creamos nuestro esquema
const PersonaSchema = Schema({
  //Campos en el esquema
  //Variable string para nombre de la persona
  nombre: String,
  //Calificación de la persona
  calificación: { type: Number, default: 0 },
  //Materia, solo esta restringido para seleccionar cualquiera de las materias que se muestran
  materia: { type: String, enum: ['matematicas', 'ciencias', 'historia'] },
})
```

Figura 5: Esquema del fichero “Modelos/personas.js”.

Como se observa en el esquema queda definido tres variables, describiendo tales variables serian:

1. nombre. - que es de tipo string, nos permite guardar su nombre.
 2. calificación. - que es de tipo number, nos permite guardar su calificación.
 3. materia. - que es de tipo String, nos permite poner el nombre de la materia, el nombre de la materia está restringida a solo: “matematicas”, “ciencias” e “historia”.
 4. id. - este MongoDB, lo crea por defecto.
- Routes/index.js. - En este fichero se definen las rutas de la API, con estas rutas se pueden acceder a los distintos tipos de peticiones, como lo serian: get, post, put y delete. Ahí mandamos a llamar a nuestras funciones previamente definidas en la carpeta “Controles”. La siguiente figura muestra como quedaron las rutas de la API:

```
//Instanciamos nuestros controladores
const personatCtrl = require('../Controles/personas')

//Ruta de tipo get, con controlador
api.get('/sps/helloworld/v1', personatCtrl.getPersonas)

//Ruta de tipo get, con controlador
api.get('/sps/helloworld/v1/:personaId', personatCtrl.getPersona)

//Ruta de tipo post, con controlador
api.post('/sps/helloworld/v1', personatCtrl.savePersona)

//Ruta de tipo put, con controlador
api.put('/sps/helloworld/v1/:personaId', personatCtrl.updatePersona)

//Ruta de tipo delete, con controlador
api.delete('/sps/helloworld/v1/:personaId', personatCtrl.deletePersona)
```

Figura 6: Rutas de la API, definidas en el fichero “Routes/index.js”.

- App.js. - Aquí se describe la funcionalidad de express y body-parse, esto con el fin de obtener los archivos JSON que especificamos para el servidor. Además de que usamos nuestros controladores. Esto se puede ver reflejado con más claridad en la siguiente figura:

```

//Aquí ira la funcionalidad del express

//Importamos express
const express = require('express')

//Importamos body-parser
const bodyParser = require('body-parser')

//Creamos nuestro servidor
const app = express()

//Llamamos las rutas
const api = require('./Routes')

//Añadimos middleware, capas a nuestra app
app.use(bodyParser.urlencoded({ extended: false }))

//Recibiremos peticiones en formato JSON
app.use(bodyParser.json())

//Que usemos api de nuestros controladores
app.use('/api', api)

//exportamos
module.exports = app

```

Figura 7: fichero “app.js”.

- config.js. - Aquí van configurado lo que son los puertos para la API, así como los de MongoDB. Esto se puede ver con más detalle en la siguiente figura:

```

module.exports = {
  //Instanciamos el puerto como una constante
  //Escogemos el puerto 8090
  port: process.env.PORT || 8090,
  //conexion a la base de datos. Defecto: 27017. Enlazamos al servicio de mongo Docker
  db: process.env.MONGODB || 'mongodb://mongo/registro'
}

```

Figura 8: fichero “config.js”.

- Index.js. - En este fichero es donde corren todos los ficheros que describimos previamente. Es donde se conecta con la base de datos, así como esta la API funcionando en el puerto 8090. Esto se observa claramente en la siguiente captura:


```

//Importamos mongoose
const mongoose = require('mongoose')

//Referencia a nuestro archivo app.js
const app = require('./app')

//Referencia a nuestra configuraciones de conexion
const config = require('./config')

//Usamos las configuraciones de conexion para los puertos a la base de datos
mongoose.connect(config.db, (err, res) => {
  //lanzar un error en caso de que lo haya
  if (err) throw err
  //En caso de que haya conexion
  console.log('Conexion a la base de datos exitosa !!!');
})

//Usamos las configuraciones de conexion para la API
app.listen(config.port, () => {
  //Mensaje que se muestra en nuestra terminal
  console.log(`API REST corriendo en http://localhost:\${config.port}/api/sps/helloworld/v1`);
})

```

Figura 9: fichero “index.js”.

Con los ficheros previamente descritos se puede correr nuestra aplicación. Esto se puede ver claramente en la parte de resultados.

En cuanto en a la parte de “Dockerizar” la API, se crean los siguientes ficheros:

- .dockerignore.- En este archivo solo definimos que no tome en cuenta la carpeta “node_modules”.
- Docker-compose.yml.- En este archivo Docker componemos distintas imágenes, este caso serian dos, la de la API y la de mongo, así como las configuraciones de ambas imágenes, esto lo podemos ver más claramente en la siguiente imagen:

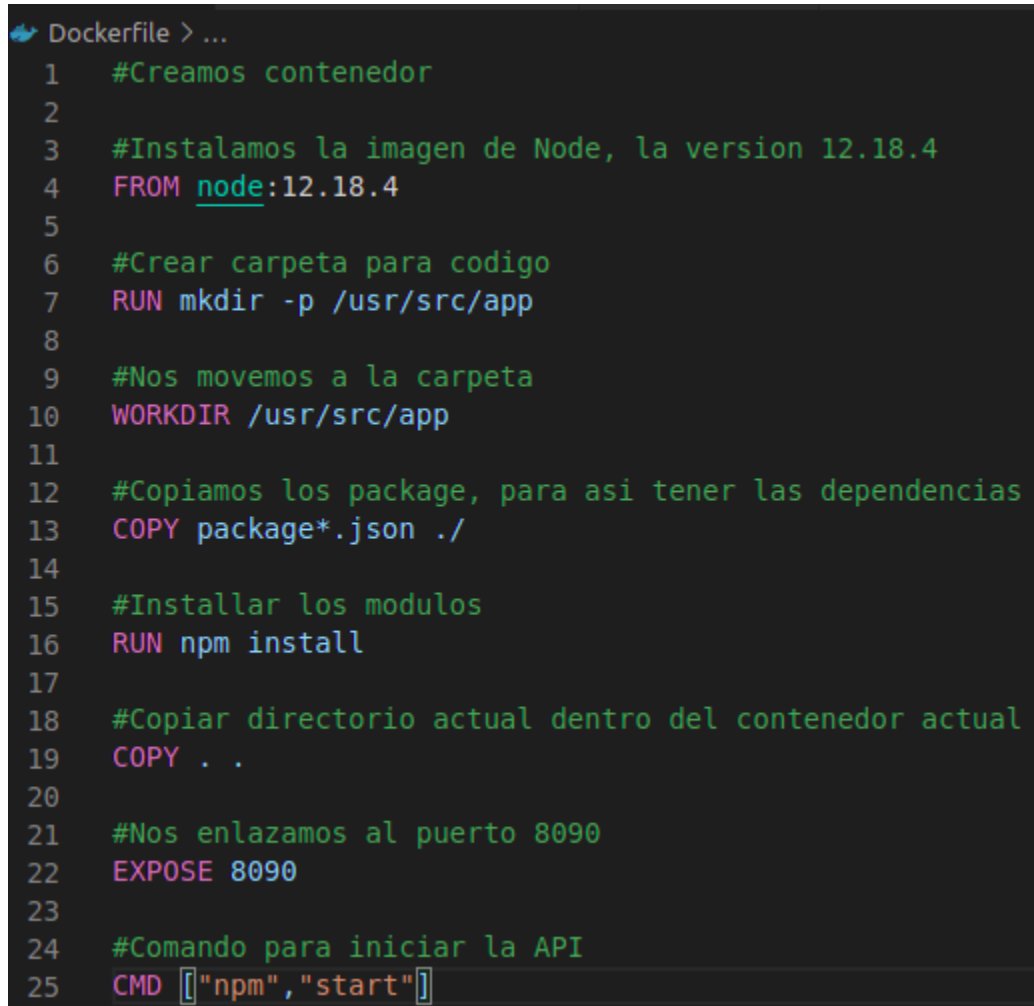
```

🔥 docker-compose.yml
1  #Componer distintas imagenes o contenedores en docker
2
3  #Especificamos version docker compose
4  version: "3"
5
6  #Especificar los servicios(imagenes a crear)
7  services:
8      api:
9          #Nombre de nuestro contenedor
10         container_name: api_sps
11         #En caso de que haya un error reiniciar
12         restart : always
13         #Correr los directorios, directorio actual
14         build : .
15         #Especificacion de puertos, que corra del 8090 al 8090
16         ports:
17             - "8090:8090"
18         #Enlazamos con mongo
19         links:
20             - mongo
21         #Agregamos un volumen
22         volumes:
23             #todo lo que esta ahi, se copie en la siguiente directorio
24             - ../usr/src/app
25
26         #Contenedor enlazado
27         mongo:
28             #Nombre de nuestro contenedor
29             container_name: base_mongo
30             #Imagen en la que esta basada
31             image: mongo
32             #Especificamos los puertos la conexion
33             ports:
34                 - "27018:27017"

```

Fichero 10: Fichero “Docker-compose.yml”.

- Dockerfile.- Definimos lo que sería el contenedor de la API esto con el fin de usarla en el docker-compose y así componerla con la base de datos de MongoDB. Es lo podemos ver más claramente en la siguiente figura:



```
Dockerfile > ...
1  #Creamos contenedor
2
3  #Instalamos la imagen de Node, la version 12.18.4
4  FROM node:12.18.4
5
6  #Crear carpeta para codigo
7  RUN mkdir -p /usr/src/app
8
9  #Nos movemos a la carpeta
10 WORKDIR /usr/src/app
11
12 #Copiamos los package, para asi tener las dependencias
13 COPY package*.json ./
14
15 #Instalar los modulos
16 RUN npm install
17
18 #Copiar directorio actual dentro del contenedor actual
19 COPY . .
20
21 #Nos enlazamos al puerto 8090
22 EXPOSE 8090
23
24 #Comando para iniciar la API
25 CMD ["npm", "start"]
```

Figura 11: Fichero “Dockerfile”.

Finalmente, solo se subió aun repositorio en Git-hub, desde una terminal de Ubuntu. Esto se puede observar en la siguiente captura:

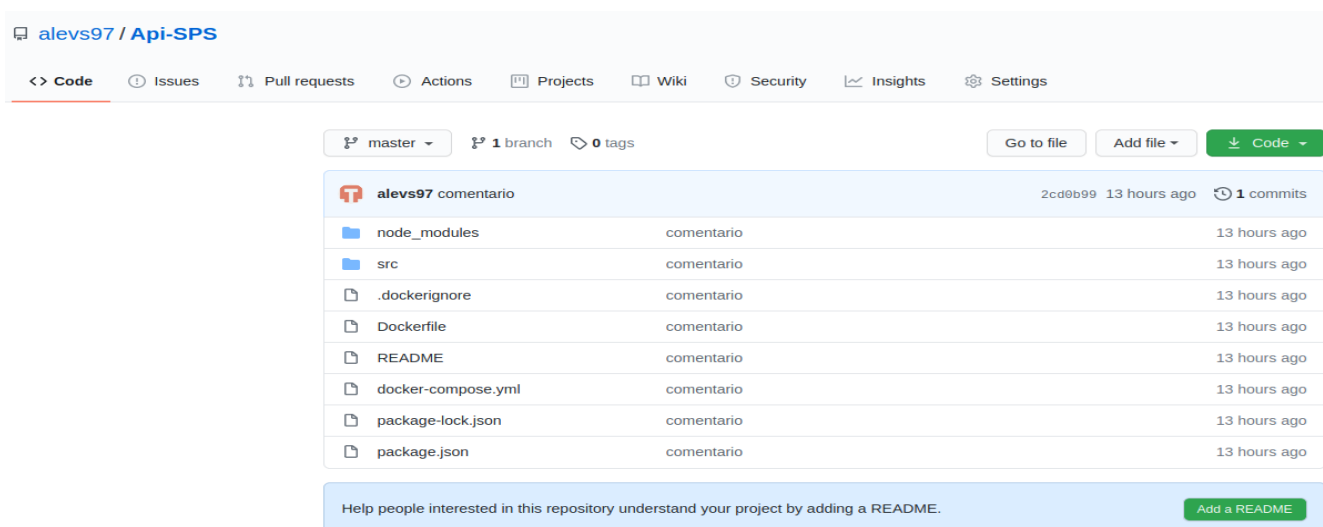


Figura 12: Repositorio en la web.

Pruebas y Resultados:

Probando dentro del contenedor:

Realizando GET(global).-

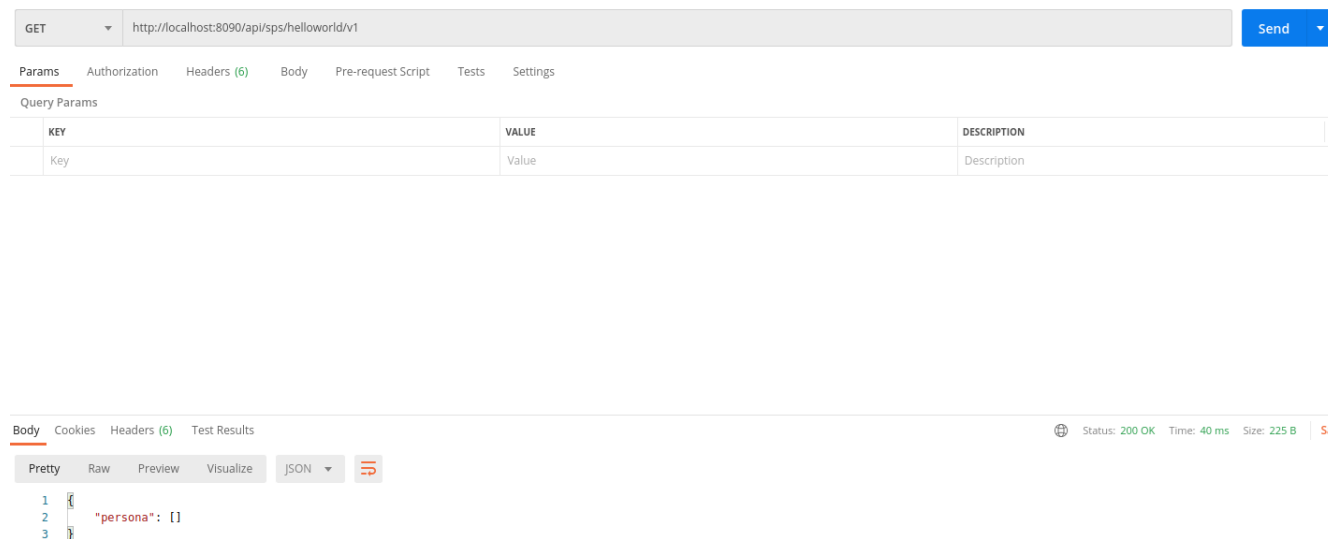


Figura 13: Éxitos de la función GET(Global).

Realizando GET(por ID):

GET http://localhost:8090/api/sps/helloworld/v1/5f740de399d263001f89692f Send

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> nombre	Alejandro	
<input checked="" type="checkbox"/> calificacion	10	
<input checked="" type="checkbox"/> materia	historia	
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 50 ms Size: 327 B Save

Pretty Raw Preview Visualize JSON

```
1 {
2   "persona": {
3     "calificacion": 10,
4     "id": "5f740de399d263001f89692f",
5     "nombre": "Alejandro",
6     "materia": "historia",
7     "_v": 0
8   }
9 }
```

Figura 14: Éxitos de la función GET(ID).

Realizando POST.-

POST http://localhost:8090/api/sps/helloworld/v1 Send

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> nombre	Alejandro	
<input checked="" type="checkbox"/> calificacion	10	
<input checked="" type="checkbox"/> materia	historia	
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 416 ms Size: 327 B Save

Pretty Raw Preview Visualize JSON

```
1 {
2   "persona": {
3     "calificacion": 10,
4     "id": "5f740de399d263001f89692f",
5     "nombre": "Alejandro",
6     "materia": "historia",
7     "_v": 0
8   }
9 }
```

Figura 15: Éxitos de la función POST

Realizando PUT:

PUT http://localhost:8090/api/sps/helloworld/v1/5f740de399d263001f89692f Send

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL

KEY	VALUE	DESCRIPTION
<input type="checkbox"/>		
<input checked="" type="checkbox"/> calificacion	7	
<input checked="" type="checkbox"/> materia	matematicas	
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 13 ms Size: 329 B Save

Pretty Raw Preview Visualize JSON

```
1 {
2   "persona": {
3     "calificacion": 7,
4     "id": "5f740de399d263001f89692f",
5     "nombre": "Alejandro",
6     "materia": "matematicas",
7     "_v": 0
8   }
9 }
```

Figura 16: Éxitos de la función PUT.

Realizando DELETE:

DELETE http://localhost:8090/api/sps/helloworld/v1/5f740de399d263001f89692f Send

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL

KEY	VALUE	DESCRIPTION
<input type="checkbox"/>		
<input checked="" type="checkbox"/> calificacion	7	
<input checked="" type="checkbox"/> materia	matematicas	
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 40 ms Size: 268 B Save

Pretty Raw Preview Visualize JSON

```
1 {
2   "message": "Se ha eliminado a la persona correctamente"
3 }
```

Figura 17: Éxitos de la función DELETE.

Problemas y errores:

1.- Ingreso de datos por Postman:

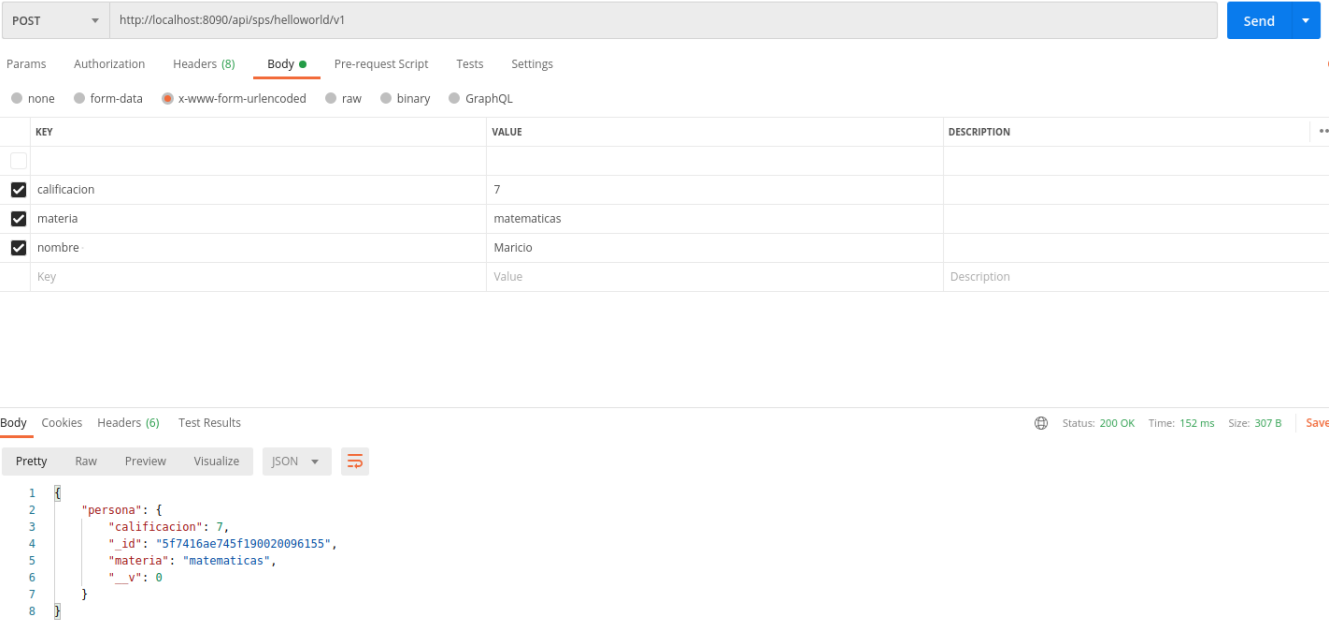
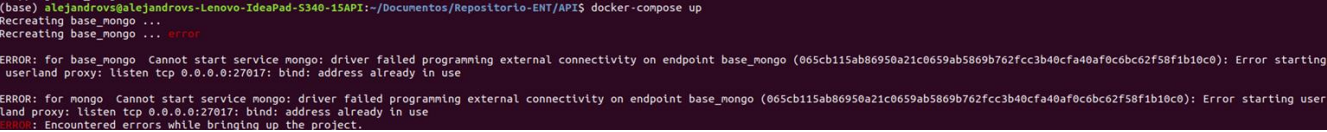


Figura 18: Error a ingresar datos en Postman

Este error se generó al usar el método POST, de la API, debido a que al ingresar en los campos se generaba un espacio. Por ende, su solución fue poner atención al llenar los campos.

2.-Error al desplegar la API en el contenedor:



Este error se debió a que en la maquina donde se desplego la API por primera vez, se le instalo MongoDB, por ende, MongoDB toma el puerto 27017, así que cuando se conecta a la imagen, estamos usando un puerto que ya en uso, la solución fue usar otro puerto para la base de datos.

3.- Problema con las versiones de las dependencias:

Este problema se debió que, al crear la API, había dependencias que no usaban la misma sintaxis. Por ende, no ejecutaba varias veces. La solución fue probar con varias versiones, revisando la documentación de esta.

4.- Error en la ejecución de los métodos DELETE y PUT.

Esto se debió a que la ruta que se establecieron para estos métodos estaba mal hecha. Por lo tanto, no recibía ningún parámetro. Se soluciono modificando la URL

5.-Obtencion del archivo JSON en el servidor.

Este error se debe a que en la API se definió el mensaje JSON en el body por ende, cuando se enviaba y mensajes POST, o cualquier otro. Se definían en otra parte en el “Postman”, haciendo que no recibiera el JSON. Esto se solucionó, configurando “Postman” para mandarlos en el body.

Conclusiones:

Como conclusión de la realización de esta API, cumple con todas las características necesarias para poder ser considerada, una API-REST. Incluso cuenta con características interesantes, como la capacidad que tiene de tener una base de datos e incluso tener las facilidades de que no se tiene que construir una nueva imagen cada vez que haya un cambio, esto debido a que cuenta con características como los volúmenes. Una API, debe contener todas aquellas características que le permitan ser escalable e incorporar funcionalidades cada vez mas avanzadas, pero su verdadera función es facilitar el desarrollo de aplicaciones web. Si esto lo complementamos con el poder de los contenedores, imágenes y volúmenes, tenemos una aplicación capaz de se ejecutada por cualquier usuario, brindando nuevas ventajas.

Referencias:

<https://www.xataka.com/basics/api-que-sirve/>

<https://apiumhub.com/es/tech-blog-barcelona/beneficios-de-utilizar-docker/>

<https://platzi.com/blog/beneficios-de-node/>

<https://www.javiergarzas.com/2015/07/que-es-docker-sencillo.html>