

Aplicación de Spring Boot con MySQL y Docker

Introducción

En este artículo, vamos a desarrollar una API CRUD RESTful con Spring Boot con Spring 2 + JPA y con la base de datos MySQL. crearemos una imagen docker y ejecutaremos la imagen de la aplicación como un contenedor docker.

Por lo tanto diseñaremos una solución de gestión de usuarios donde realizaremos un CRUD sobre la base de datos.

Antes de comenzar, aquí hay una lista de lo que necesitamos para completar este tutorial:

Requisitos

Lenguaje de programación	Java SE 11	https://www.oracle.com/java/technologies/javase-jdk11-downloads.html https://adoptopenjdk.net/installation.html
Entorno Integrado de Desarrollo (IDE)	IntelliJ	https://www.jetbrains.com/idea/download/
Maven	Versión 3+	https://maven.apache.org/download.cgi
Base de Datos	MySQL	
Docker	Docker	
Lombok	1.18.12	https://projectlombok.org/setup/intellij
Postman	Postman	

Crearemos un Recurso de Contacto exponiendo tres servicios usando URIs RESTful y métodos HTTP:

- Recuperar todos los contactos - @GetMapping("/contacts")
- Obtener detalles de contacto específico - @GetMapping("/contacts/{id}")

- Borrar un contacto - `@DeleteMapping("/contacts/{id}")`
- Crear un nuevo contacto - `@PostMapping("/contacts")`
- Actualizar los datos de contacto existentes - `@PutMapping("/contacts/{id}")`

Creación de la aplicación en Spring Boot

Para crear la aplicación SpringBoot, utilizaremos <https://start.spring.io/>, que proporciona algún código inicial (una Clase principal y pom.xml)

Por último seleccionamos las Dependencias que se utilizarán en este desarrollo, para este ejemplo seleccionamos las siguientes dependencias **Spring Data JPA**, **Lombok**, **Spring Web Starter** y **MySQL Driver**. Una de las ventajas de Spring Boot es que proporciona paquetes iniciales que simplifican su configuración de Maven. Los iniciadores Spring Boot son en realidad un conjunto de dependencias que puede incluir en su proyecto. Puede escribir las dependencias en el campo de búsqueda o cambiar a la versión completa y ver todos los paquetes de inicio y las dependencias disponibles como se puede observar en la figura 1.

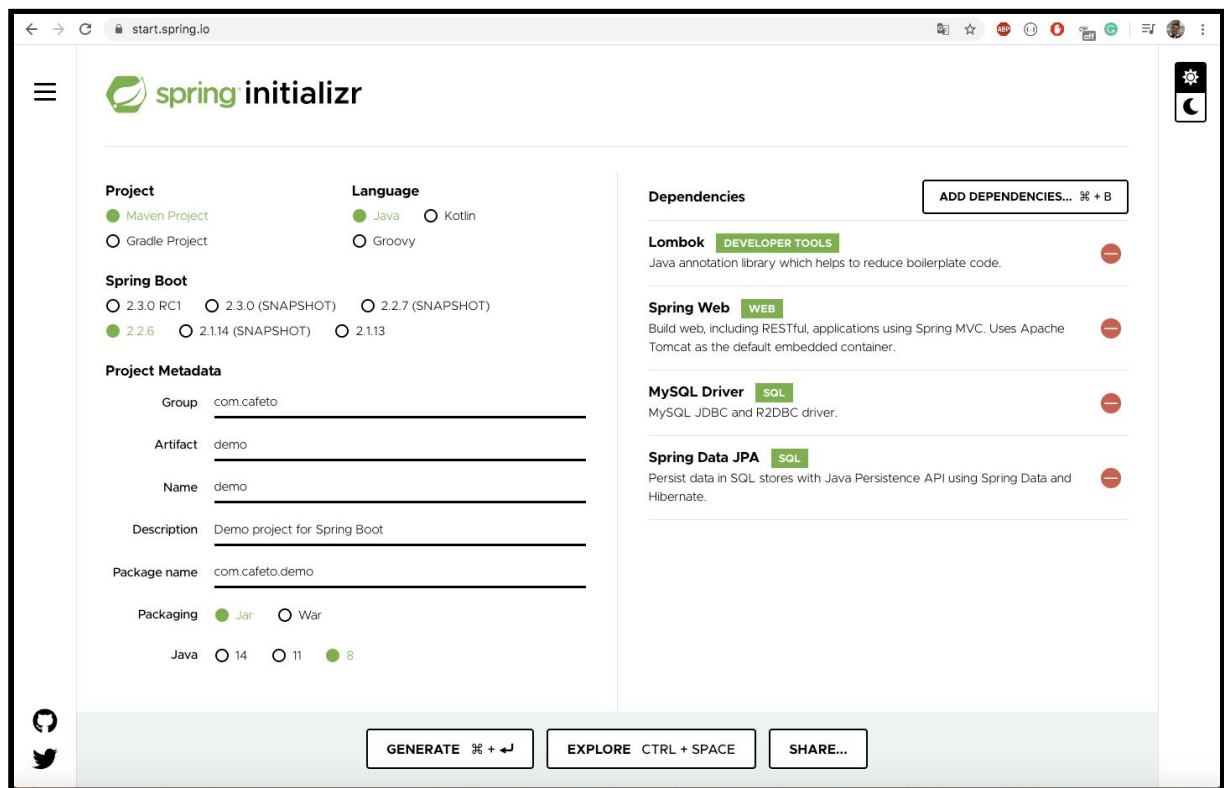


Figura 1

Generate: **Maven Project**

Java Version: **11**

Spring Boot: **2.2.6**

Group: **com.cafeto**

Artifact: **spring-docker-mysql**

Dependencies: **Web, JPA, Lombok, MySQL**

Estructura de la Aplicación

Como se puede observar la **figura 2**, la estructura de la aplicación estará dividida por paquete.

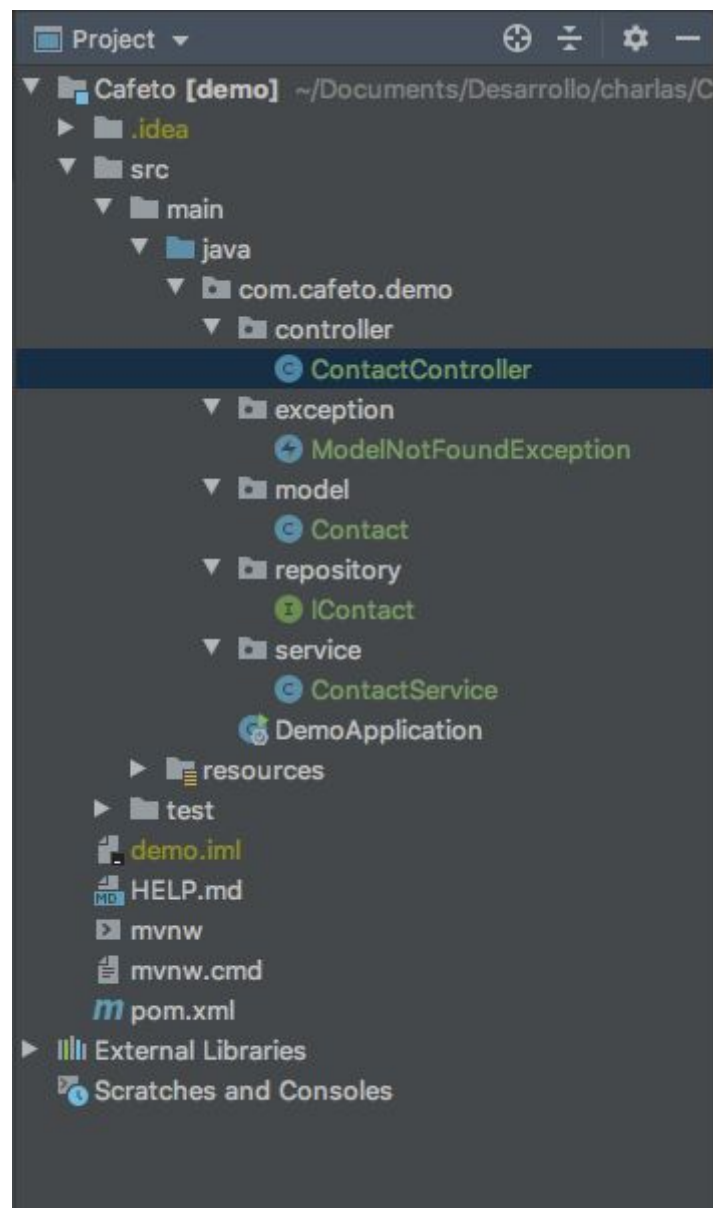


Figura 2

Esta parte se encuentra en el archivo pom.xml que es un plugin de Spring boot(que sirve para empaquetar la aplicación)

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <finalName>demo</finalName>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.0</version>
      <configuration>
        <source>${java.version}</source>
        <target>${java.version}</target>
        <annotationProcessorPaths>
          <path>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.8</version>
          </path>
        </annotationProcessorPaths>
      </configuration>
    </plugin>
  </plugins>
</build>
<repositories>
  <repository>
    <id>projectlombok.org</id>
    <url>https://projectlombok.org/edge-releases</url>
  </repository>
</repositories>
```

Una breve explicación de las clases o interfaces que se utilizaran en el ejemplo.

Contact.java

Esta clase de modelo representa un contacto. Esta clase contiene un id de contacto que se auto genera, con sus respectivo nombre y apellido.

@AllArgsConstructor

@NoArgsConstructor

@Data

@Entity

public class Contact {

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long idContact;
private String firstName;
private String lastName;

}

```

Las siguientes anotaciones son del proyecto Lombok y nos ayudan a mantener nuestras clases (especialmente las de modelo/POJO) más limpias sin los getters y setters:

AllArgsConstructor: crea automáticamente un constructor de clases con todos los argumentos (propiedades).

NoArgsConstructor: crea automáticamente un constructor de clase vacío con todos los argumentos (propiedades).

Data: crea toString, equals, hashCode, getters y setters.

IContactRepository.java

El Spring Boot Data JPA proporciona una interfaz CrudRepository para dar soporte a las operaciones básicas de las operaciones CRUD, dándole funcionalidades a la clase entidad, como se puede observar la

```

@Repository
public interface IContactRepository
    extends JpaRepository<Contact, Long> { }

```

La interfaz del JpaRepository proporciona una forma simple y fácil de acceder a todas las operaciones del CRUD.

contactoService.java

La clase de ContactService facilita la comunicación del repositorio con las operaciones de guardar, actualizar, eliminar, listar por Idcontacto y listar todas las contactos

```

@Service
public class ContactService {

    @Autowired
    private IContact dao;

    public Contact save(Contact t) { return dao.save(t); }

    public void deleteById(long id) { dao.deleteById(id); }
}

```

```

public Iterable<Contact> list() { return dao.findAll(); }

public Optional<Contact> listId(long id) {
    return dao.findById(id);
}
}

```

ContactController.java

Los servicios web son aplicaciones que se comunican a través de Internet utilizando el protocolo HTTP. Hay muchos tipos diferentes de arquitecturas de servicios web, pero la idea principal en todos los diseños es la misma. En este artículo, estamos creando un servicio web RESTful a partir de lo que es un diseño, con el objetivo de ver la información de una contacto. crear endpoint permite crear un nuevo registro de la contacto en el sistema, también existe la opción de un endpoint, donde devuelve una respuesta **JSON** donde se visualiza la información de todas las contactos disponibles en la aplicación, ver/{id} endpoint permite buscar información de la contacto con el id de contacto apropiada.

```

@Slf4j
@RestController
public class ContactController {

    @Autowired
    ContactService contactService;

    @PostMapping("/save")
    public ResponseEntity create(@Valid @RequestBody Contact contact) {
        return ResponseEntity.ok(contactService.save(contact));
    }

    @GetMapping("/listAll")
    public Iterable<Contact> listAllPersons() {
        return contactService.list();
    }

    @GetMapping("/list/{id}")
    public Contact listPersonById(@PathVariable("id") long id) {
        Optional<Contact> person = contactService.listId(id);
        if(person.isPresent()) {
            return person.get();
        }
        throw new ModelNotFoundException("Invalid find person provided");
    }

    @PutMapping("/update/{id}")

```

```

    public ResponseEntity<Contact> update(@PathVariable Long id, @Valid @RequestBody
    Contact contact) {
        if (!contactService.listId(id).isPresent()) {
            log.error("Id " + id + " is not existed");
            ResponseEntity.badRequest().build();
        }
        return ResponseEntity.ok(contactService.save(contact));
    }

    @DeleteMapping("/delete/{id}")
    public ResponseEntity delete(@PathVariable Long id) {
        if (!contactService.listId(id).isPresent()) {
            log.error("Id " + id + " is not existed");
            ResponseEntity.badRequest().build();
        }
        contactService.deleteById(id);
        return ResponseEntity.ok().build();
    }
}

```

ModelNotFoundException.java

Lanzará una excepción de tiempo de ejecución personalizada si el identificador de la contacto no existe en la aplicación.

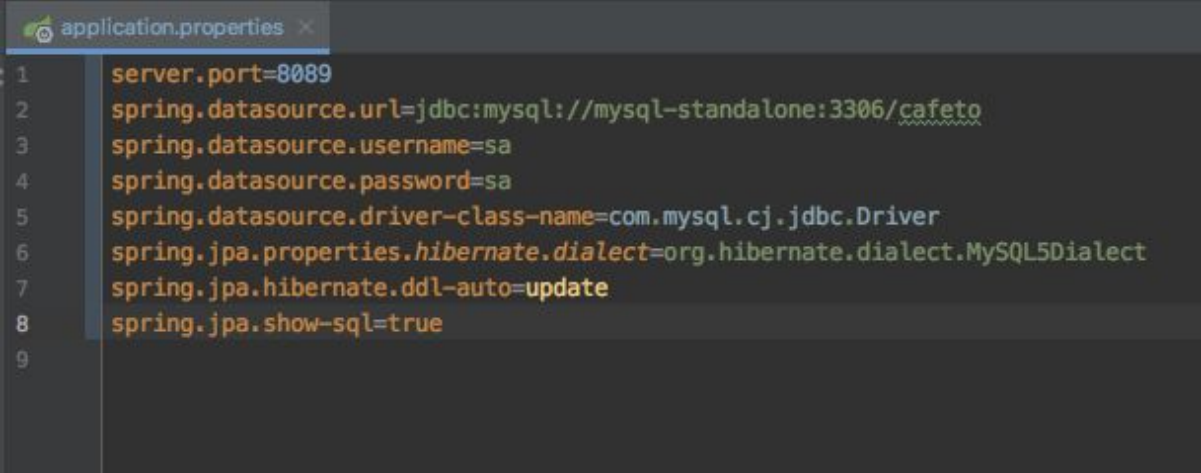
```

@ResponseStatus(HttpStatus.NOT_FOUND)
public class ModelNotFoundException extends RuntimeException {
    public ModelNotFoundException(String mensaje) {
        super(mensaje);
    }
}

```

Application.properties

Spring Boot application.properties nos permite configurar la aplicación Spring Boot como se puede observar en la **figura 8**. Nuestra aplicación se ejecutara en el puerto 8089 conectado a una base de datos MySQL con nombre de usuario **sa** y contraseña cómo **sa**.

A screenshot of an IDE window titled 'application.properties'. The file contains the following configuration: server.port=8089, spring.datasource.url=jdbc:mysql://mysql-standalone:3306/cafeto, spring.datasource.username=sa, spring.datasource.password=sa, spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver, spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect, spring.jpa.hibernate.ddl-auto=update, and spring.jpa.show-sql=true.

```
1 server.port=8089
2 spring.datasource.url=jdbc:mysql://mysql-standalone:3306/cafeto
3 spring.datasource.username=sa
4 spring.datasource.password=sa
5 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
6 spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
7 spring.jpa.hibernate.ddl-auto=update
8 spring.jpa.show-sql=true
9
```

Figura 8

Crear la imagen Docker

Creamos un nuevo archivo con el nombre **Dockerfile** dentro de la carpeta **cafeto** al mismo nivel que se encuentran la carpeta **src**, con las siguientes instrucciones como se puede observar en la **figura 9**, esto es con el objetivo de construir una imagen con la versión de Java y el ejecutable de la aplicación.

A screenshot of an IDE window titled 'Dockerfile'. The file contains the following instructions: FROM openjdk:11, ADD target/demo.jar demo.jar, EXPOSE 8089, and ENTRYPOINT ["java", "-jar", "demo.jar"].

```
1 FROM openjdk:11
2 ADD target/demo.jar demo.jar
3 EXPOSE 8089
4 ENTRYPOINT ["java", "-jar", "demo.jar"]
```

Figura 9

- 1) openJDK 11.
- 2) Adicionamos en el directorio target el jar de la demo en este caso se llama demo.
- 3) En esta línea se configura el puerto de la aplicación lo colocamos el 8089.
- 4) Ejecutamos el comando Java -Jar.

Desplegar la Aplicación

Creamos la Imagen Docker, usando el comando **sudo docker build -t demo .** desde el directorio donde se encuentra el archivo Docker. Este comando instruye a Docker para crear la imagen de nuestra aplicación.

mvn clean package

mvn compile

mvn install

sudo docker build -t demo .

Como se puede observar la **figura 10**, fueron creadas exitosamente las imágenes.

```
Terminal: Local x +
MacBook-Pro-de-Geovanny:Cafeto gmendoza$ sudo docker build -t demo .
Sending build context to Docker daemon  40.9MB
Step 1/4 : FROM openjdk:11
--> f5de33dc9079
Step 2/4 : ADD target/demo.jar demo.jar
--> 5c4a31190cbe
Step 3/4 : EXPOSE 8089
--> Running in d59a9382ad4c
Removing intermediate container d59a9382ad4c
--> 9bfc64323a92
Step 4/4 : ENTRYPOINT ["java", "-jar", "demo.jar"]
--> Running in a31bc3969d41
Removing intermediate container a31bc3969d41
--> d765bb664a79
Successfully built d765bb664a79
Successfully tagged demo:latest
MacBook-Pro-de-Geovanny:Cafeto gmendoza$
```

Figura 10

Si quieren verificar las imagenes creadas utilizamos el siguiente comando **docker images** como se puede observar en la **figura 11**

```
Terminal: Local x +
MacBook-Pro-de-Geovanny:Cafeto gmendoza$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
demo                 latest              d765bb664a79       About a minute ago 668MB
openjdk              11                  f5de33dc9079       2 weeks ago        627MB
MacBook-Pro-de-Geovanny:Cafeto gmendoza$
```

Figura 11

En el siguiente paso ejecutamos el comando **sudo docker pull mysql:5.7** para para desplegar en la máquina un contenedor docker de MySQL.

Para este caso, seleccionamos la versión de MySQL y podemos usar el comando de arriba. De esta manera, podemos pull de la imagen mysql-server:5.7, como se puede observar en la **figura 12**.

```
Terminal: Local × Local (2) × +
MacBook-Pro-de-Geovanny:jugnucaragua gmendoza$ sudo docker pull mysql:5.7
Password:
5.7: Pulling from library/mysql
68ced04f60ab: Pulling fs layer
f9748e016a5c: Pulling fs layer
da54b038fed1: Pulling fs layer
6895ec5eb2c0: Pull complete
111ba0647b87: Pull complete
c1dce60f2f1a: Pull complete
702ec598d0af: Pull complete
63cca87a5d4d: Pull complete
ec05b7b1c5c7: Pull complete
834b1d9f49b0: Pull complete
8ded6a30c87c: Pull complete
Digest: sha256:f4a5f5be3d94b4f4d3aef00fbc276ce7c08e62f2e1f28867d930deb73a314c58
Status: Downloaded newer image for mysql:5.7
docker.io/library/mysql:5.7
MacBook-Pro-de-Geovanny:jugnucaragua gmendoza$
```

Figura 12

Desplegar el contenedor

Para desplegar el servidor de MySQL podemos usar el siguiente comando, **sudo docker run --name mysql-standalone -e MYSQL_ROOT_PASSWORD=sa -e MYSQL_DATABASE=cafeto -e MYSQL_USER=sa -e MYSQL_PASSWORD=sa -d mysql:5.7**, para que funcione como un contenedor Docker.

Verificamos los registros de inicio de MySQL usando el siguiente comando:

docker container logs mysql-standalone

Para conectarnos al contenedor mysql, ejecutamos el siguiente comandos:

docker exec -it mysql-standalone bash -l (Donde mysql-standalone es el nombre que se colocó al contenedor)

iniciamos sesion en MySQL con el siguiente comando **mysql -usa -psa**, como se puede observar en la **figura 14**.

```
Terminal: Local x Local (2) x +
MacBook-Pro-de-Geovanny:jugnicaragua gmendoza$ docker exec -it mysql-standalone bash -l
root@81f30223e0af:/# mysql -usa -psa
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.29 MySQL Community Server (GPL)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> 
```

Figura 14

Para verificar si está creada la base de datos cafeto, utilizamos el siguiente comando **show databases;** como se puede observar en la **figura 15**.

```
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| cafeto |
+-----+
2 rows in set (0.00 sec)
```

Figura 15

Desplegar el contenedor de la Aplicación

Ejecutamos nuestra aplicación Sprint Boot, usando los siguientes comandos:

- **mvn clean package**
- **mvn install**

docker run -d -p 8089:8089 --name demo --link mysql-standalone:mysql demo

```
MacBook-Pro-de-Geovanny:jugnicaragua gmendoza$ docker run -d -p 8089:8089 --name demo-nica --link mysql-standalone:mysql demo-nica  
c4d3cb4bb3f9a28c8a0d2ee01da091f7f2ee28d27d8a1f87d02addfc456b7913  
MacBook-Pro-de-Geovanny:jugnicaragua gmendoza$
```

Figura 16

Verificamos los registros de inicio del contenedor de la aplicación, usando el siguiente comando:

docker container logs demo

Nota: En el caso que requiera eliminar los contenedores o imágenes de docker.

- Eliminar todos los contenedores **`docker rm -vf $(docker ps -a -q)`**
- Eliminar todas las imágenes **`docker rmi -f $(docker images -a -q)`**

Ejecutar en POSTMAN

Crear un nuevo registro

Crear una nueva solicitud POST para crear un nuevo registro de contacto. Una vez que el registro se crea con éxito, obtenemos el ID de la contacto como respuesta. Como se puede observar en la **figura 17**

<http://localhost:8089/save>

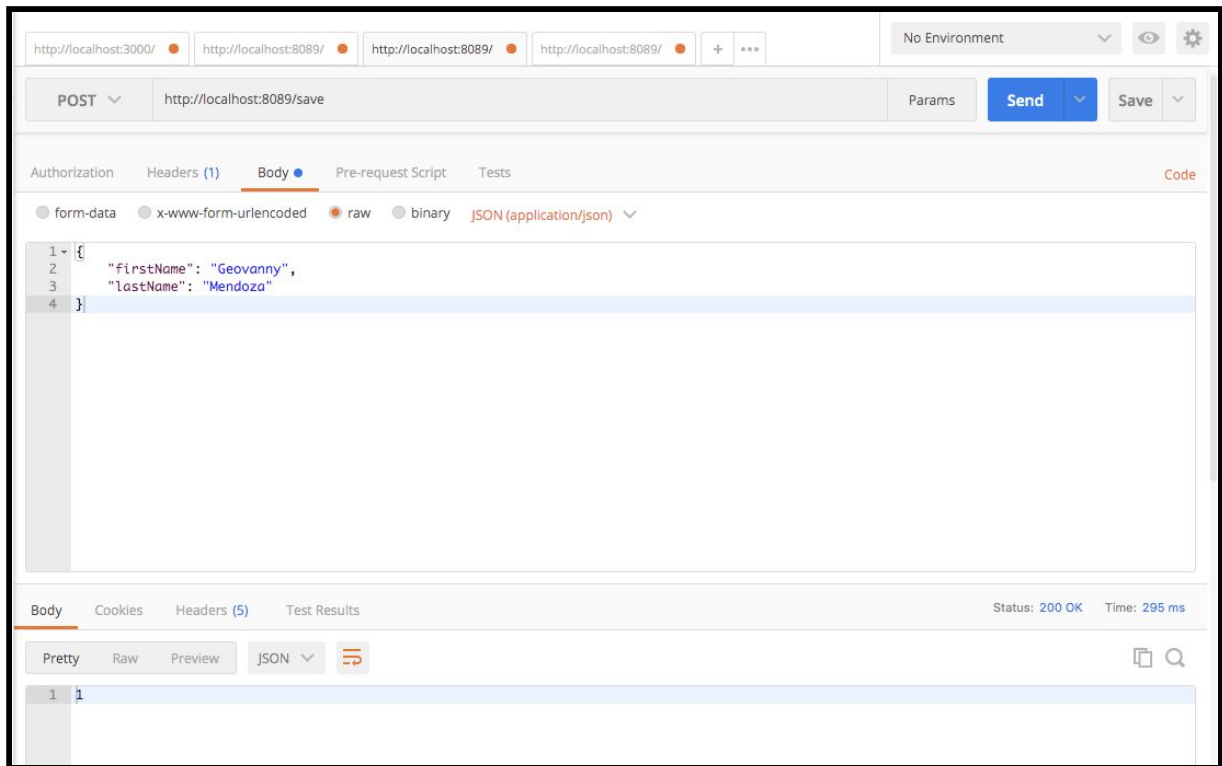


Figura 17

Ver todas las contactos registradas

Es una operación GET devuelve todas las contactos registrada en la aplicación, Como se puede observar en la **figura 18**

<http://localhost:8089/listAll>

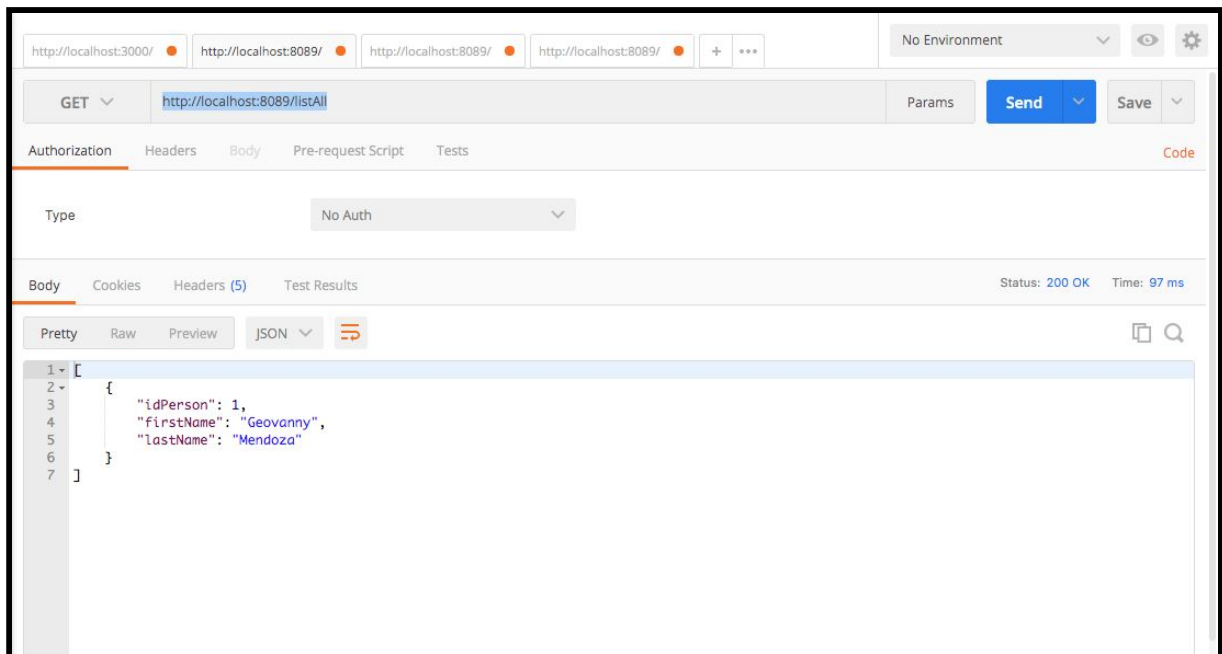


Figura 18

Repositorio

- <https://github.com/Geovanny0401/charlas/tree/master/Cafeto>

Conclusión

Se desarrolló una aplicación en spring boot para desplegarla por medio de un contenedor docker y conectada con una base de datos MySQL.

Referencias

- <https://spring.io/projects/spring-framework>
- <https://stackoverflow.com/questions/44785585/how-to-delete-all-docker-local-docker-images>
- <https://medium.com/codefountain/develop-a-spring-boot-and-mysql-application-and-run-in-docker-end-to-end-15b7cdf3a2ba>
- <https://mkyong.com/docker/docker-spring-boot-examples/>
- <http://geovanny0401.blogspot.com/2018/08/construir-una-aplicacion-web-con-vaadin.html>