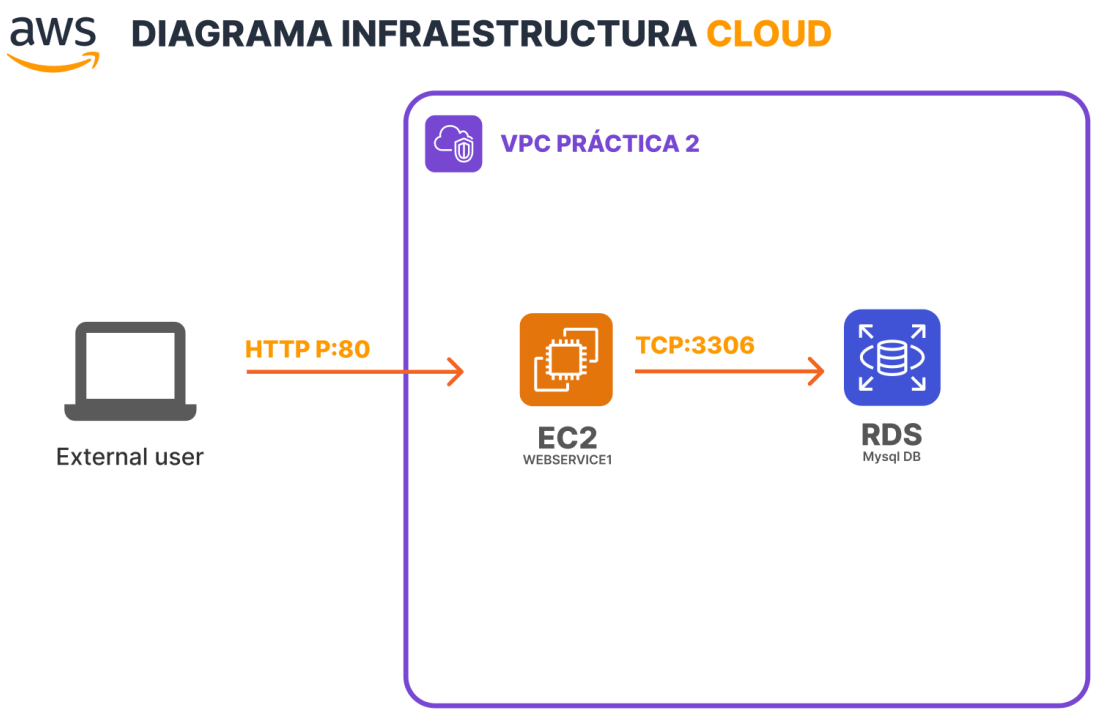
Actividad Extra: Despliegue de una BD RDS y conexión con una API RESTful

*Para la realización de esta actividad extra, hemos implementado una de las interfaces de los repositorios del proyecto entregado en la práctica 1 utilizando **Hibernate**, un ORM (Object-Relational Mapping) para Java que viene integrado con **Spring Boot**. El objetivo es conectar este ORM con la base de datos desplegada, permitiendo que nuestra aplicación disponga de una capa de persistencia en la nube.*

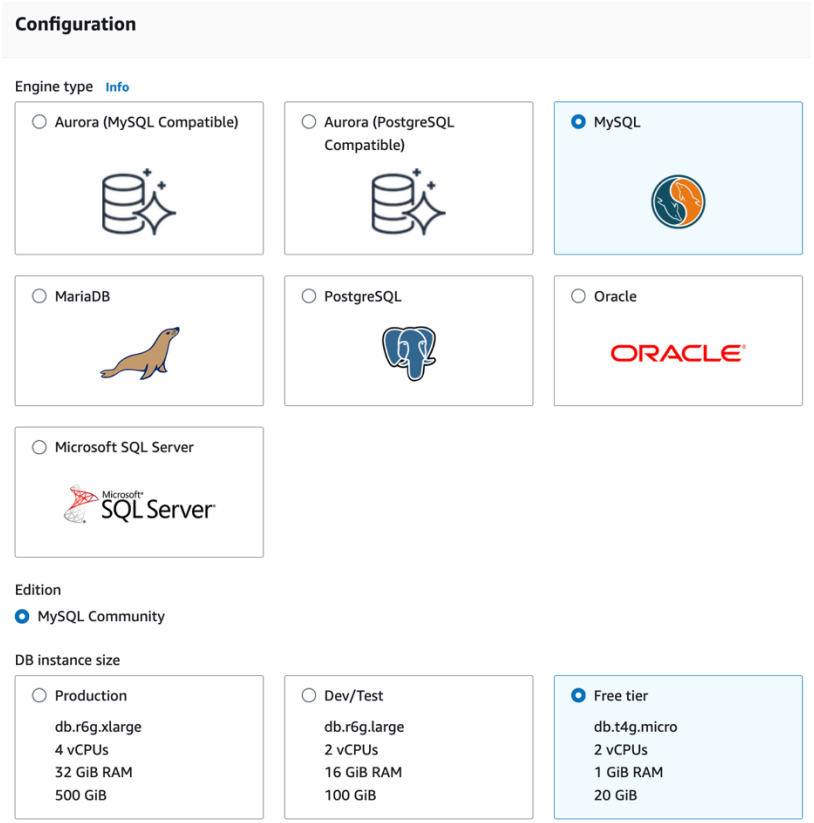
*Gracias a que seguimos una **arquitectura hexagonal**, no ha sido necesario modificar nuestra lógica de negocio. Dicha lógica ha sido modelada de manera que se mantenga desacoplada de dependencias externas, como puede ser la base de datos. Esto garantiza que la lógica permanezca independiente de los detalles de infraestructura.*

Para lograr la persistencia, simplemente hemos creado una implementación de uno de los repositorios. Luego, configuramos el inyector de dependencias para que esta nueva implementación sea inyectada en lugar del `MockRepository`, sin afectar el funcionamiento del núcleo de la aplicación.

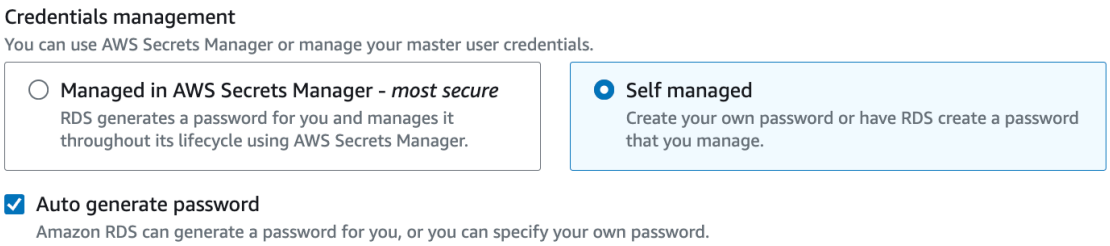


Despliegue de la base de datos Mysql

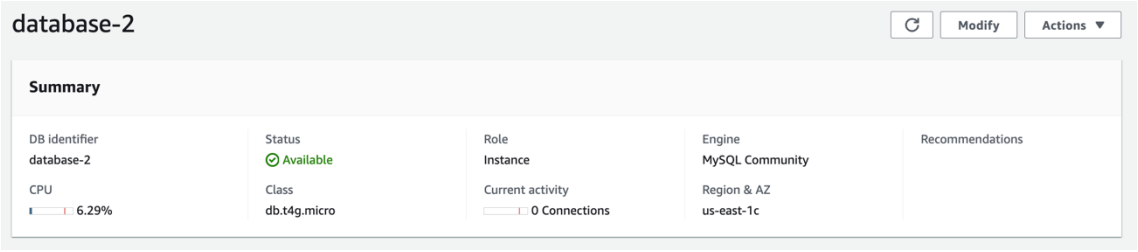
*Para la configuración de la base de datos, hemos optado por utilizar **MySQL** en una instancia **db.t4g.micro** de la capa gratuita de AWS. Esta instancia nos proporciona **1 GiB de RAM** y **20 GiB de almacenamiento**, lo cual es más que suficiente para realizar nuestras pruebas rápidas y asegurar que todo funcione correctamente. Esta configuración nos permite optimizar recursos mientras mantenemos un entorno adecuado para el desarrollo y testeo de la aplicación.*



En cuanto a la seguridad, optaremos por gestionar manualmente la contraseña de la base de datos. Para garantizar una mayor protección, utilizaremos un sistema de generación automática de contraseñas seguras. De esta manera, nos aseguramos de que la contraseña cumpla con altos estándares de seguridad y sea difícil de vulnerar, minimizando posibles riesgos asociados al acceso no autorizado.



Finalmente, hemos creado con éxito nuestra base de datos. Ahora está lista para ser utilizada en nuestra aplicación, permitiendo gestionar de manera eficiente la persistencia de datos y garantizando un entorno seguro y óptimo para nuestras pruebas y desarrollo.



*Modificaremos las reglas **inbound** del **Security Group** asignado a la base de datos para permitir conexiones **TCP** a través del puerto **3306**, el puerto predeterminado de **MySQL**. Esto permitirá que nuestra aplicación se conecte de forma segura a la base de datos desde las IPs o rangos de IPs especificados. En este caso permitiremos la conexión a todas las IPs.*

Creación de la base de datos

*En **MySQL**, al igual que en muchos otros sistemas de **gestión de bases de datos (SGBD),** es posible crear múltiples bases de datos en una misma instancia. Para que nuestra aplicación funcione correctamente, debe acceder a una base de datos específica. De forma predeterminada, MySQL incluye **4 tablas relacionadas con su funcionamiento interno**, por lo que será necesario crear una nueva base de datos.*

Para crearla, ejecutaremos el siguiente comando:

```
mysql -h database-1.ciymx7zbznko.us-east-1.rds.amazonaws.com -u admin -p
```

Una vez conectados a MySQL, podemos listar todas las bases de datos disponibles ejecutando el siguiente comando:

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
4 rows in set (0.12 sec)
```

*A continuación, crearemos una nueva con el nombre: **app***

```
mysql> CREATE DATABASE app;
Query OK, 1 row affected (0.13 sec)
```

Conexión con la base de datos

Una vez completados los pasos anteriores, procederemos a la conexión desde la aplicación. Para esto, debemos modificar el archivo `application.properties` *para incluir las credenciales de acceso a la base de datos. Sin embargo, como pushear credenciales sensibles a GitHub no es la mejor de las ideas que se le puede ocurrir a un estudiante de ingeniería informática, editaremos directamente el archivo en la instancia EC2 conectándonos por* ***SSH****.*

Accederemos a la instancia EC2 mediante SSH y modificaremos el archivo `application.properties` *para incluir las credenciales de la base de datos. Aunque esto funcionará para este caso práctico, es importante aclarar que no es el flujo más seguro ni eficiente para deployar una aplicación.*

Lo ideal sería utilizar un ***gestor de variables de entorno****, como* ***AWS Systems Manager Parameter Store****, o la herramienta de gestión de secretos de* ***AWS Secrets Manager*** *para inyectar las credenciales de manera automática y segura. Sin embargo, en esta práctica optamos por la solución más sencilla para facilitar la implementación.*

```
spring.application.name=Pepeducacion

# Server
server.port=8080

# MySQL Database Connection
spring.datasource.url=jdbc:mysql://database-1.ciymx7zbznko.us-east-1.rds.amazonaws.com:3306/app
spring.datasource.username=admin
spring.datasource.password=mndp]V3Q:Y86H+hwFv3u
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Hibernate Configuration
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true

# Use correct Hibernate dialect for MySQL 8.x
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
~
~
```

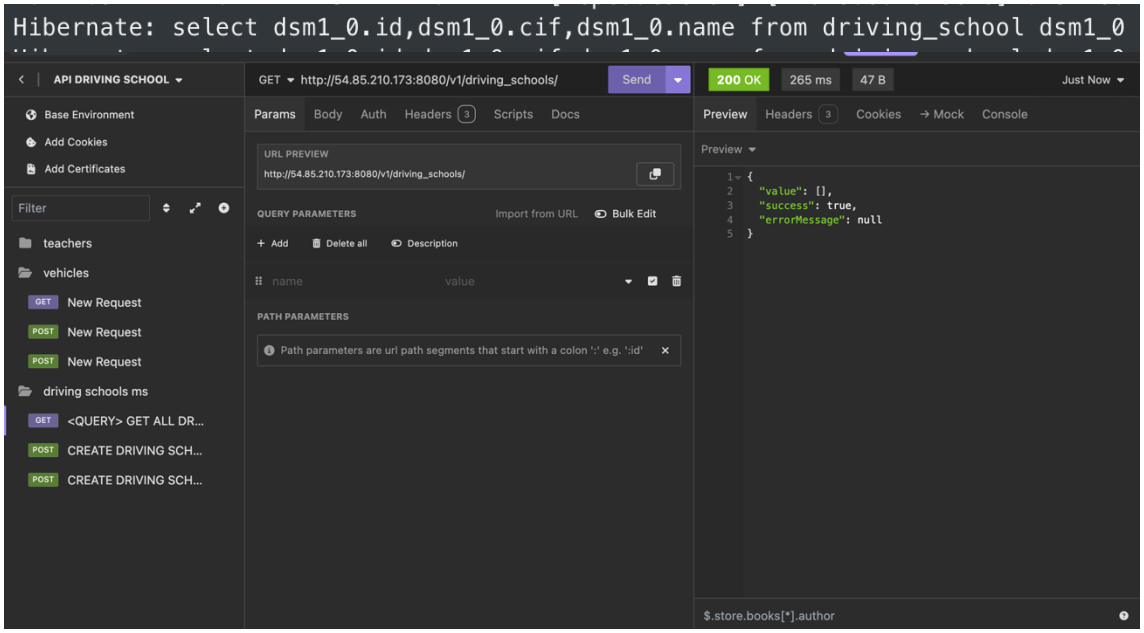
Ahora procederemos a hacer el ***build*** *de la imagen de Docker y posteriormente ejecutar un nuevo contenedor con dicha imagen.*

```
sudo docker build -it springboot
sudo docker run -p 8080:8080 springboot
```

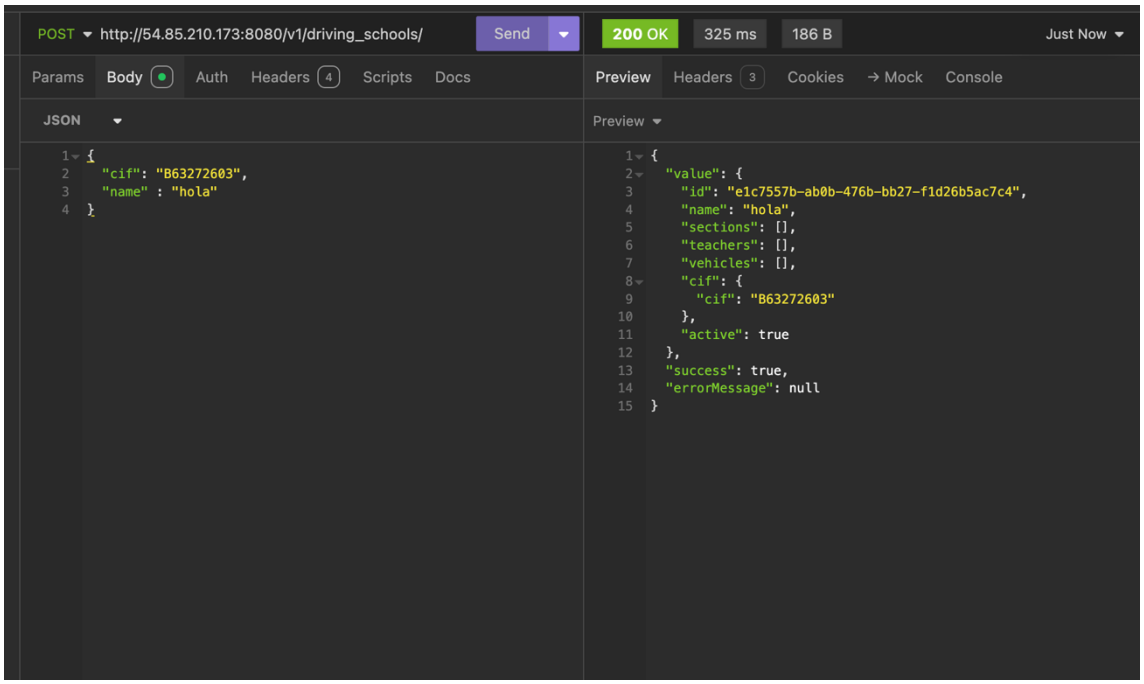
Podemos observar cómo ***Hibernate*** *comienza a ejecutar las instrucciones de* ***LDD*** *(Lenguaje de Definición de Datos) de manera correcta, lo que implica la creación y configuración de las tablas necesarias en la base de datos. Una vez que todas las estructuras de la base de datos han sido configuradas, el servidor se lanza exitosamente, lo que confirma que la aplicación está conectada correctamente a la base de datos y está lista para recibir solicitudes.*

```
Hibernate: drop table if exists driving_school
Hibernate: drop table if exists driving_school_sections
Hibernate: drop table if exists driving_school_teachers
Hibernate: drop table if exists driving_school_vehicles
Hibernate: drop table if exists section
Hibernate: drop table if exists teacher
Hibernate: drop table if exists vehicle
Hibernate: create table driving_school (id BINARY(16) not null, cif varchar(255) not null, name varchar(255) not null, primary key (id)) engine=InnoDB
Hibernate: create table driving_school_sections (driving_school_model_id BINARY(16) not null, sections_id BINARY(16) not null) engine=InnoDB
Hibernate: create table driving_school_teachers (driving_school_model_id BINARY(16) not null, teachers_id BINARY(16) not null) engine=InnoDB
Hibernate: create table driving_school_vehicles (driving_school_model_id BINARY(16) not null, vehicles_id BINARY(16) not null) engine=InnoDB
Hibernate: create table section (latitude float(53), longitude float(53), id BINARY(16) not null, primary key (id)) engine=InnoDB
Hibernate: create table teacher (id BINARY(16) not null, dni varchar(255), primary key (id)) engine=InnoDB
Hibernate: create table vehicle (id BINARY(16) not null, plate_number varchar(255), primary key (id)) engine=InnoDB
Hibernate: alter table driving_school_sections add constraint UKosxs8qxxubthk269b2ng5l9p8 unique (sections_id)
Hibernate: alter table driving_school_teachers add constraint UKpuwcw4hx538hdwqpahe1g4caf unique (teachers_id)
Hibernate: alter table driving_school_vehicles add constraint UKo4xsv0vj2e0lvv7ghtfrswwqj unique (vehicles_id)
Hibernate: alter table driving_school_sections add constraint FKa44lc8l3qtviyodibw3oj0ton foreign key (sections_id) references section (id)
Hibernate: alter table driving_school_sections add constraint FKt05hp7jsyvma1dm91x66t79b2 foreign key (driving_school_model_id) references driving_school (id)
Hibernate: alter table driving_school_teachers add constraint FKs75ci80s2d7gjb9hv319ipw2j foreign key (teachers_id) references teacher (id)
Hibernate: alter table driving_school_teachers add constraint FKnadqojk3y199ced8be3ec084w foreign key (driving_school_model_id) references driving_school (id)
Hibernate: alter table driving_school_vehicles add constraint FKfbu188g4b9c1625rmttapvetx foreign key (vehicles_id) references vehicle (id)
Hibernate: alter table driving_school_vehicles add constraint FKnewnuy0uhwov59np02dxyd3m3 foreign key (driving_school_model_id) references driving_school (id)
2024-09-17T21:59:32.721Z  INFO 1 --- [Pepeducacion] [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2024-09-17T21:59:33.307Z  WARN 1 --- [Pepeducacion] [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries
 may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2024-09-17T21:59:33.927Z  INFO 1 --- [Pepeducacion] [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path '/'
2024-09-17T21:59:33.973Z  INFO 1 --- [Pepeducacion] [           main] c.e.p.apps.springBootRestApi.Main        : Started Main in 8.848 seconds (process running for 9.995)
```

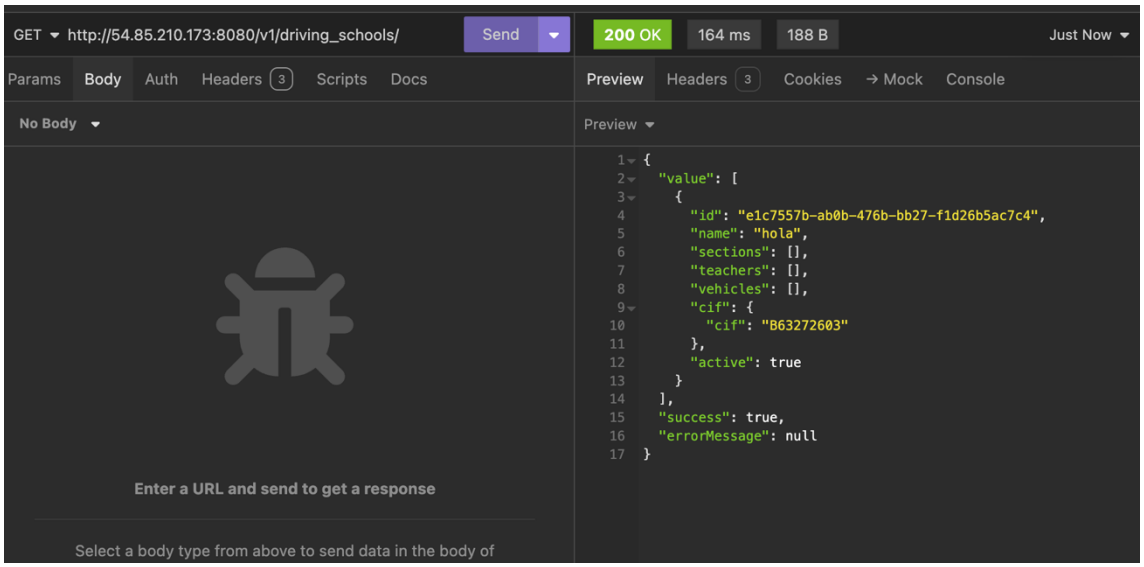
*Ahora probaremos que la API funciona correctamente. Utilizaremos el cliente REST **Insomnia** para realizar diferentes peticiones HTTP a nuestra API. Como podemos ver, al realizar una petición **GET** a la ruta de **autoescuelas**, obtenemos una respuesta exitosa y los datos esperados. Además, en la consola de **Spring Boot**, podemos observar que se ha realizado la consulta correspondiente a la base de datos mediante un **query**, lo que confirma que la conexión y la interacción con la base de datos están funcionando como se espera.*



*Procedemos a crear un nuevo registro en la base de datos. Utilizando **Insomnia**, enviamos una petición **POST** para crear una nueva **driving school**. Como podemos observar, la creación del registro se realiza correctamente y recibimos una respuesta exitosa con el nuevo recurso creado. Además, la consola de **Spring Boot** confirma la inserción al mostrar la ejecución del **query** correspondiente en la base de datos. Esto verifica que la operación de inserción en nuestra API está funcionando de manera adecuada.*



*Ahora, al realizar nuevamente una petición **GET** para listar todas las **driving schools** a través de **Insomnia**, podemos observar que el listado ha sido actualizado correctamente. El nuevo registro que acabamos de crear aparece en la respuesta, confirmando que la base de datos ha sido modificada y que los datos se reflejan en tiempo real en nuestra API. Esto garantiza que tanto la inserción como la consulta de datos están funcionando de manera adecuada.*



En resumen, todo está funcionando correctamente. Si observamos más de cerca, podemos ver que nuestra base de datos ha gestionado varias conexiones y ha registrado actividad de uso, lo que indica que las operaciones de inserción, consulta y demás interacciones con la base de datos se están ejecutando de manera eficiente. Esto confirma que nuestra API y la capa de persistencia están funcionando como se esperaba.