

GNP SEGUROS

EJERCICIO PRÁCTICO

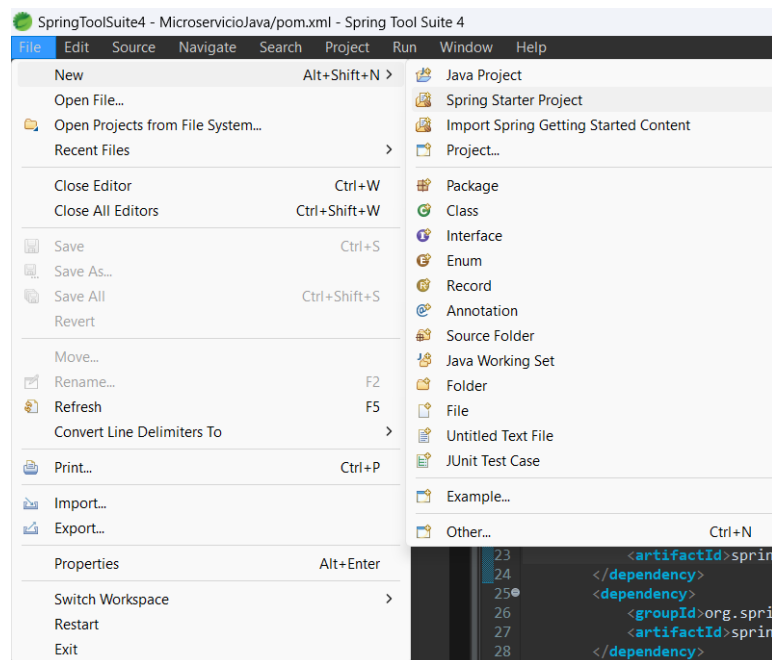
SERVICIO REST CON JAVA, AGREGANDO SWAGGER AL SERVICIO

**Ing. Sebastián Gerardo
Palacios Pérez**

INICIALIZANDO EL PROYECTO

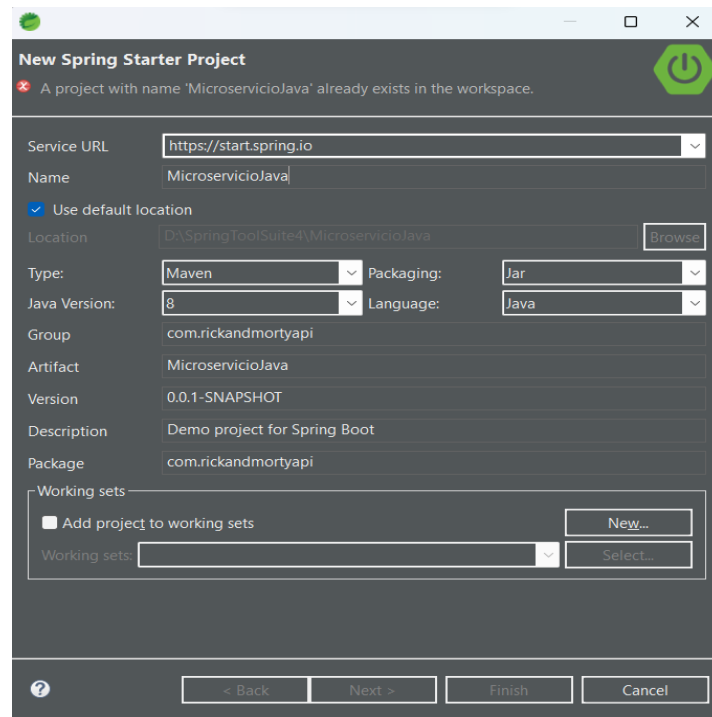
Utilizando el entorno de desarrollo Spring Tool Suite 4 realicé el ejercicio técnico a continuación, con utilización en el framework de spring por la facilidad que me da para emplear objetos de java, además de la herramienta de spring boot para que el microservicio de spring sea más rápido y fácil por su automatización al trabajar en conjunto en java.

Iniciando un proyecto nuevo de Spring Started Project:



Con las características de tipo Maven, versión de java 8 (la más comercial en mi consideración para el ámbito laboral) en el lenguaje Java, se procede a nombrar el paquete con su respectiva clase:

NOTA: el proyecto me sale que ya fue usado el nombre, debido a que documenté después de realizarlo.



New Spring Starter Project

✖ A project with name 'MicroservicioJava' already exists in the workspace.

Service URL:

Name:

☒ Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

Description:

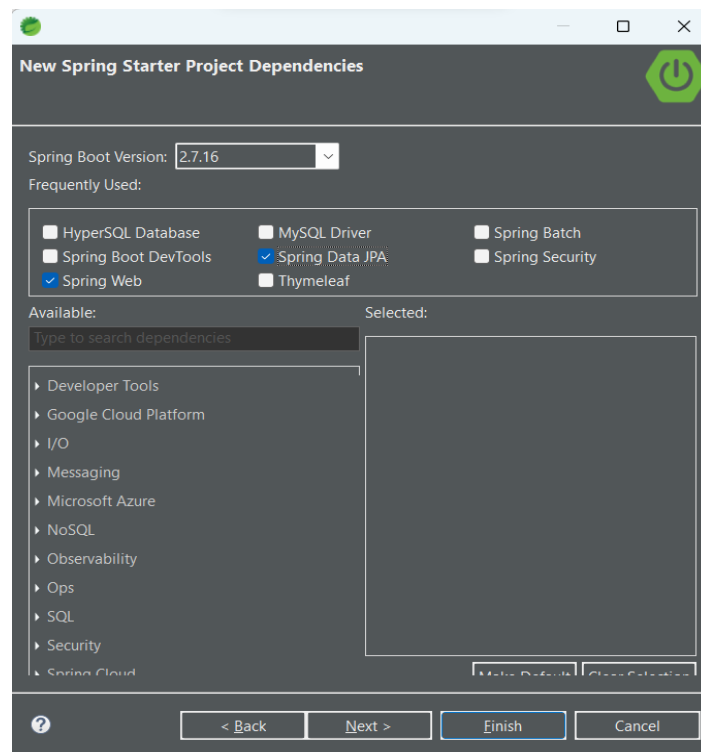
Package:

Working sets

☐ Add project to working sets

Working sets:

Utilizando la versión 2.7.16 de Spring boot, anexo las dependencias de Spring web (que nos da la funcionalidad de carga de archivos y la inicialización del contenedor IoC además que contiene un cliente HTTP) y la de Spring Data JPA (quien facilita la implementación de repositorios basados en JPA).



New Spring Starter Project Dependencies

Spring Boot Version:

Frequently Used:

<input type="checkbox"/> HyperSQL Database	<input type="checkbox"/> MySQL Driver	<input type="checkbox"/> Spring Batch
<input type="checkbox"/> Spring Boot DevTools	<input checked="" type="checkbox"/> Spring Data JPA	<input type="checkbox"/> Spring Security
<input checked="" type="checkbox"/> Spring Web	<input type="checkbox"/> Thymeleaf	

Available:

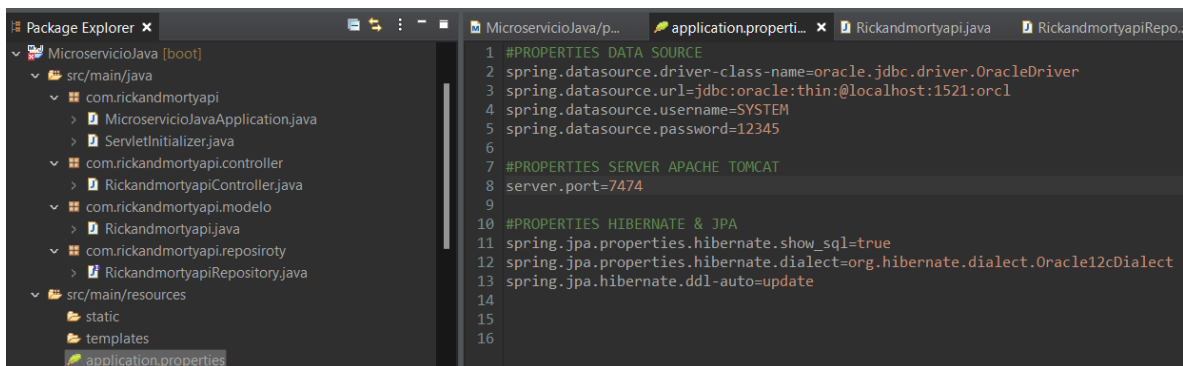
- Developer Tools
- Google Cloud Platform
- I/O
- Messaging
- Microsoft Azure
- NoSQL
- Observability
- Ops
- SQL
- Security
- Spring Cloud

Selected:

En el archivo creado por defecto pom.xml, se debe anexar la dependencia para la base de datos, en mi caso utilicé Oracle pues es la que más me agrada utilizar, para ello también creé la tabla “rickandmortyapi” para almacenar los datos requeridos en el ejercicio y correcta funcionalidad posteriormente.



Colocando las propiedades de la base de datos en el archivo “properties” del proyecto (contraseña, usuario, conexión, etc.), además del servidor apache tomcat para que nos permita que el servidor web maneje el contenido web dinámico y finalmente el mapeo con hibernate.



En src/main/java, se crea una nueva clase con el paquete modelo (com.rickandmortyapi.modelo) con su respectiva clase de rickandmortyapi para establecer los parámetros, usando el framework de entity y table para asignar la tabla creada anteriormente, indicamos el indicador (valga la redundancia) único de la entidad (ID), la columna con su respectivo nombre y generatevalue para generar los valores de la llave primaria, así como la declaración y asignación de cada columna en la tabla con sus respectivas importaciones.

The screenshot shows an IDE with a Package Explorer on the left and a code editor on the right. The Package Explorer shows the project structure for 'MicroservicioJava [boot]'. The code editor displays the 'Rickandmortyapi.java' file, which contains the following code:

```
1 package com.rickandmortyapi.modelo;
2
3 import javax.persistence.Column;
4 import javax.persistence.Entity;
5 import javax.persistence.GeneratedValue;
6 import javax.persistence.Id;
7 import javax.persistence.Table;
8
9 @Entity
10 @Table(name = "rickandmortyapi")
11 public class Rickandmortyapi {
12     @Id
13     @Column(name = "ID")
14     @GeneratedValue
15     private int id;
16     @Column(name = "NOMBRE")
17     private String nombre;
18     @Column(name = "ESTATUS")
19     private String estatus;
20     @Column(name = "ESPECIE")
21     private String especie;
22     @Column(name = "GENERO")
23     private String genero;
24     @Column(name = "IMAGEN")
25     private String imagen;
26     @Column(name = "NUMERO_EPISODIOS")
27     private String numero_episodios;
28     @Column(name = "FECHA_CREACION")
29     private String fecha_creacion;
30
31 }
```

Se generan los getters and setters:

The screenshot shows the 'Rickandmortyapi.java' file with the following code:

```
49 }
50 //GETTERS AND SETTERS
51 public int getId() {
52     return id;
53 }
54 public void setId(int id) {
55     this.id = id;
56 }
57 public String getNombre() {
58     return nombre;
59 }
60 public void setNombre(String nombre) {
61     this.nombre = nombre;
62 }
63 public String getEstatus() {
64     return estatus;
65 }
66 public void setEstatus(String estatus) {
67     this.estatus = estatus;
68 }
69 public String getEspecie() {
70     return especie;
71 }
72 public void setEspecie(String especie) {
73     this.especie = especie;
74 }
75 public String getGenero() {
76     return genero;
77 }
78 public void setGenero(String genero) {
79     this.genero = genero;
80 }
```

```

78 public void setGenero(String genero) {
79     this.genero = genero;
80 }
81 public String getImagen() {
82     return imagen;
83 }
84 public void setImagen(String imagen) {
85     this.imagen = imagen;
86 }
87 public String getNumero_episodios() {
88     return numero_episodios;
89 }
90 public void setNumero_episodios(String numero_episodios) {
91     this.numero_episodios = numero_episodios;
92 }
93 public String getFecha_creacion() {
94     return fecha_creacion;
95 }
96 public void setFecha_creacion(String fecha_creacion) {
97     this.fecha_creacion = fecha_creacion;
98 }
99
100
101 }

```

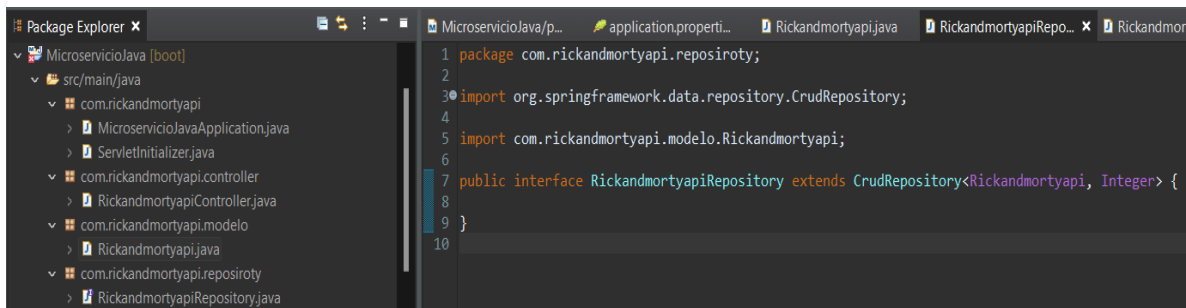
Se generan los fields de todos los elementos y la super clase, esto se genera después, pero se coloca antes de los getters and setters.

```

31
32 //SUPERCLASS
33 public Rickandmortyapi() {
34     super();
35     // TODO Auto-generated constructor stub
36 }
37 //FIELDS
38 public Rickandmortyapi(int id, String nombre, String estatus, String especie, String genero, String imagen,
39     String numero_episodios, String fecha_creacion) {
40     super();
41     this.id = id;
42     this.nombre = nombre;
43     this.estatus = estatus;
44     this.especie = especie;
45     this.genero = genero;
46     this.imagen = imagen;
47     this.numero_episodios = numero_episodios;
48     this.fecha_creacion = fecha_creacion;
49 }
50 //GETTERS AND SETTERS

```

Sobre src/main/java se genera la interface, haciendo clic derecho, new interface y colocando de parámetros el paquete com.rickandmortyapi.reposiroty y la clase Rickandmortyapi.Repository, para crear el repositorio de la capa de persistencia, creando el extends de la tabla creada anteriormente.



```

1 package com.rickandmortyapi.reposiroty;
2
3 import org.springframework.data.repository.CrudRepository;
4
5 import com.rickandmortyapi.modelo.Rickandmortyapi;
6
7 public interface RickandmortyapiRepository extends CrudRepository<Rickandmortyapi, Integer> {
8
9 }
10

```

Nuevamente sobre src/main/java haciendo clic derecho se genera un nuevo paquete con su respectiva clase, los nombre com.rickandmortyapi.controller y su clase Rickandmortyapi.Controller para realizar el controlador, haciendo el frame de “restcontroller” pues es el controlador especial en RESful, el requestmapping para asignar solicitudes web a clases de controlador específicos (como el restcontroller) y el autowired para la inyección automática de dependencias, mandando traer el repositorio creado anteriormente, getmapping para simplificar los diferentes métodos de SpringMVC y los requestmapping que a veces tienden hacerse algo pesados.

```

1 package com.rickandmortyapi.controller;
2
3 import java.util.List;
4 import java.util.Optional;
5
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.web.bind.annotation.DeleteMapping;
8 import org.springframework.web.bind.annotation.GetMapping;
9 import org.springframework.web.bind.annotation.PathVariable;
10 import org.springframework.web.bind.annotation.PostMapping;
11 import org.springframework.web.bind.annotation.PutMapping;
12 import org.springframework.web.bind.annotation.RequestBody;
13 import org.springframework.web.bind.annotation.RequestMapping;
14 import org.springframework.web.bind.annotation.RestController;
15
16 import com.rickandmortyapi.modelo.Rickandmortyapi;
17 import com.rickandmortyapi.reposirotiy.RickandmortyapiRepository;
18
19 @RestController
20 @RequestMapping({"rickandmortyapis"})
21 public class RickandmortyapiController {
22
23     @Autowired
24     private RickandmortyapiRepository repository;
25
26     @GetMapping("listar")
27     public List<Rickandmortyapi> listarRickandmortyapis(){
28         List<Rickandmortyapi> rickandmortyapis = (List<Rickandmortyapi>) repository.findAll();
29         return rickandmortyapis;
30     }
31

```

Utilicé el frame postmapping para asignar solicitudes HTTP POST al método de controlador para agregar datos, putmapping para mapear solicitudes HTTP PUT en métodos de manejadores específicos, en éste caso el editar los datos de la tabla rickandmortyapi, y por último el frame deletemapping, para asignar solicitudes HTTP DELETE para eliminar los datos de la tabla específicamente mandando el ID a borrar.

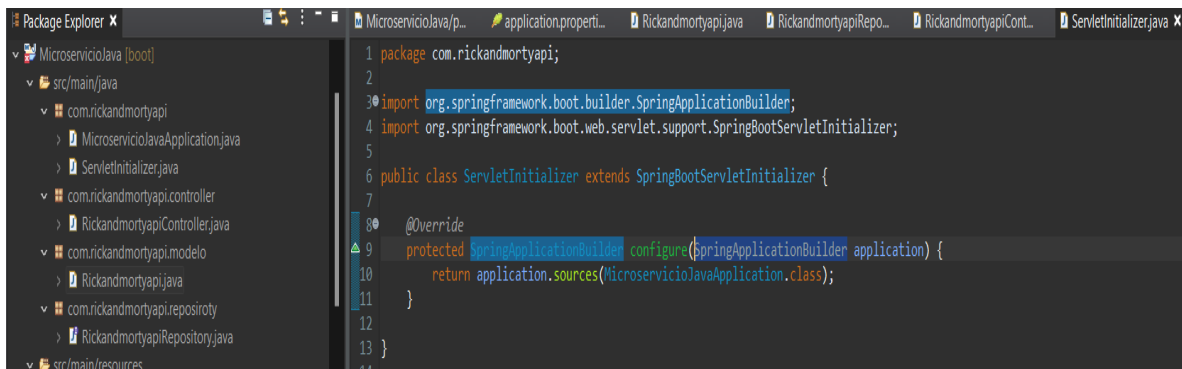
```

31
32 @PostMapping
33 public Rickandmortyapi agregarRickandmortyapi(@RequestBody Rickandmortyapi rickandmortyapi) {
34     return repository.save(rickandmortyapi);
35 }
36
37 @GetMapping("/{id}")
38 public Optional<Rickandmortyapi> getRickandmortyapiById(@PathVariable int id){
39     return repository.findById(id);
40 }
41
42 @PutMapping("/{id}")
43 public Rickandmortyapi modificarRickandmortyapi(@RequestBody Rickandmortyapi rickandmortyapi, @PathVariable int id) {
44     rickandmortyapi.setId(id);
45     return repository.save(rickandmortyapi);
46 }
47
48 @DeleteMapping("/{id}")
49 public void eliminarRickandmortyapi(@PathVariable int id) {
50     repository.deleteById(id);
51 }
52
53
54 }
55

```

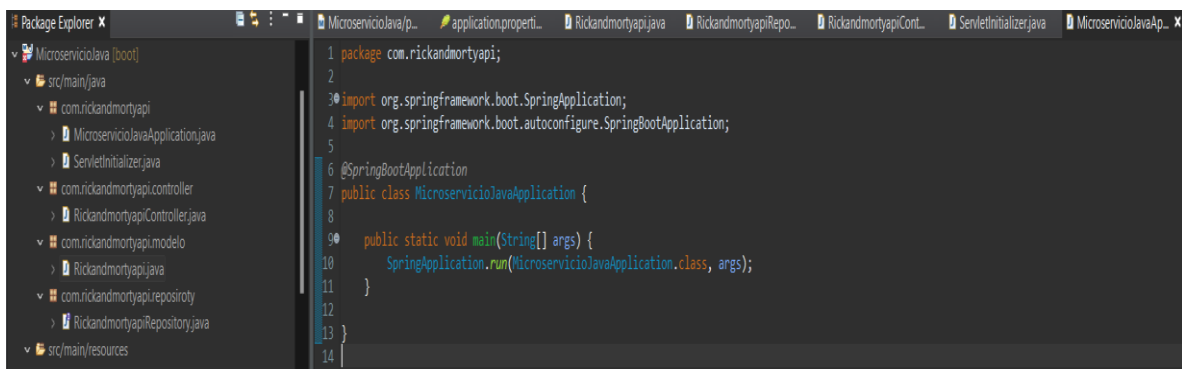
Por último, cree la clase “ServletInitializer” (que es Demon2Application) pues es la que se ejecutaría para arrancar la aplicación pues es hecha en Jar cómo se mostró en la

configuración inicial del proyecto, se hace el extends del webapplicationInitializer (SpringBootServletInitializer) para configurar la aplicación web, hacemos la configuración y despliegue de aplicaciones con SpringApplicationBuilder para el microservicio de java.



```
1 package com.rickandmortyapi;
2
3 import org.springframework.boot.builder.SpringApplicationBuilder;
4 import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
5
6 public class ServletInitializer extends SpringBootServletInitializer {
7
8     @Override
9     protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
10         return application.sources(MicroservicioJavaApplication.class);
11     }
12
13 }
```

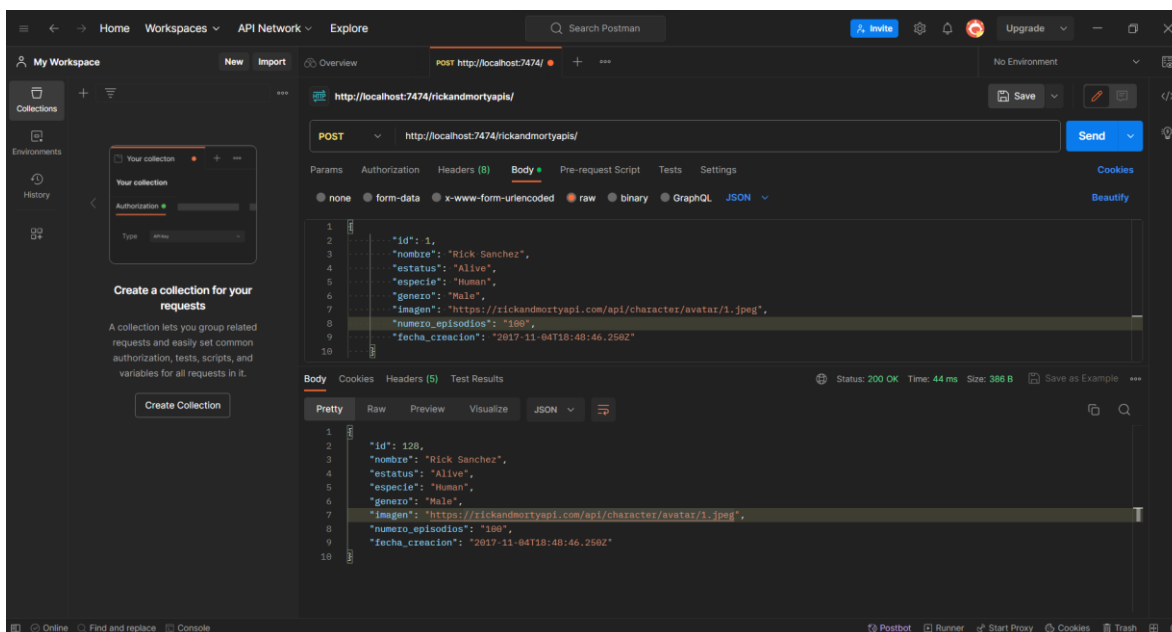
También se crea la clase del microservicio dentro del mismo paquete com.rickandmortyapi que llame MicroservicioJavaApplication, que es la anotación que aparece en la función main de todo proyecto que definamos con Spring Boot, que es el main específicamente.



```
1 package com.rickandmortyapi;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class MicroservicioJavaApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(MicroservicioJavaApplication.class, args);
11     }
12
13 }
```


TESTEO EN POSTMAN

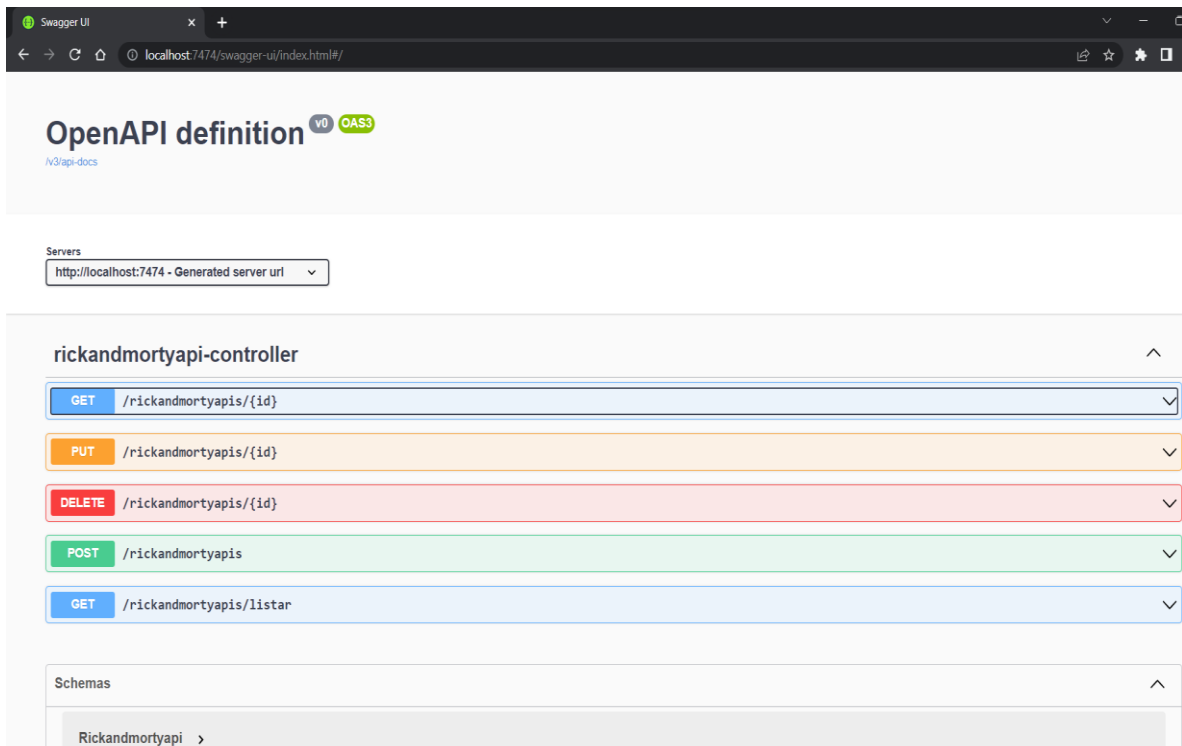
En postman ahora colocamos la dirección `http://localhost:7474/rickandmortyapis/` que es la dirección local para testear, la cual se asignó en las properties inicialmente, esto básicamente en raw, es la que permite configurar rutas de endpoints del API y ejecutarlas, para ejercitar el backend de las aplicaciones. Gracias a Postman podemos guardar todas las request que queramos, para tenerlas preparadas y poder ejecutarlas las veces que haga falta, utilizando el JavaScript JSON, que es el formato de cambio de datos que lo hace posible, ahí metemos los datos que el usuario desee en el método post, el cual es posible gracias al controller usado anteriormente, dando como resultado lo siguiente:



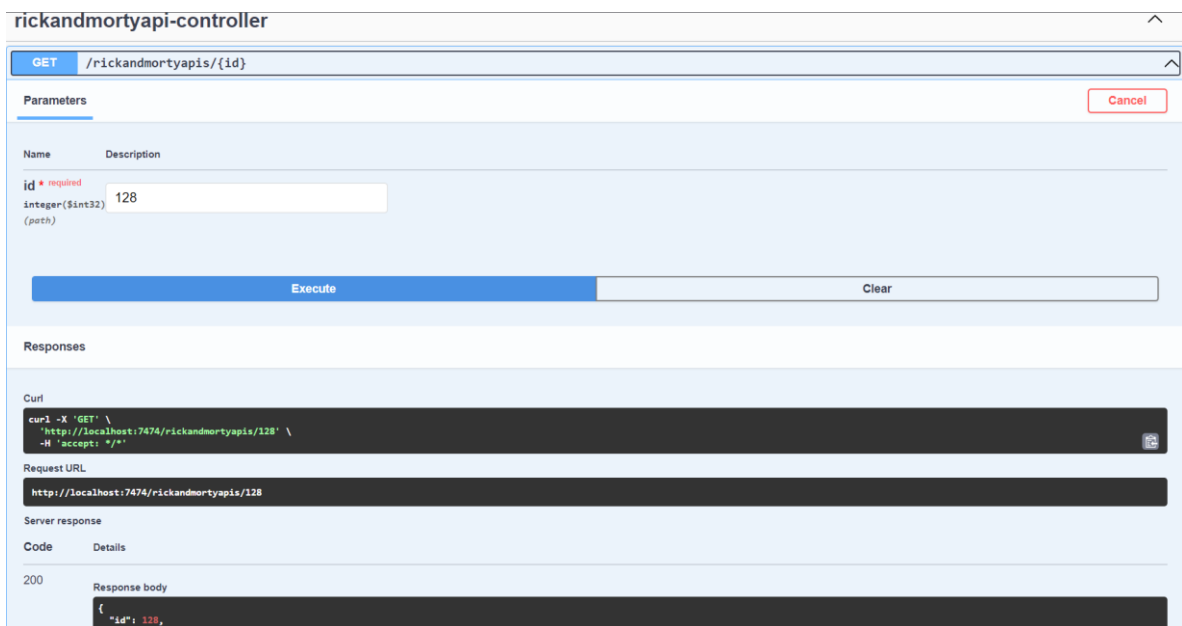
VISUALIZACIÓN EN SWAGGER UI

Usando swagger ui para la visualización y corroborar funcionalidades de los métodos creados, ingresamos el siguiente link:

`http://localhost:7474/swagger-ui/index.html#/rickandmortyapi-controller/getRickandmortyapiById`



Haciendo la prueba para ver los resultados ingresamos en el método get, colocando el id de postman, en éste caso es "128".



Code Details

200

Response body

```
{
  "id": 128,
  "nombre": "Rick Sanchez",
  "estatus": "Alive",
  "especie": "Human",
  "genero": "Male",
  "imagen": "https://rickandmortyapi.com/api/character/avatar/1.jpeg",
  "numero_episodios": "100",
  "fecha_creacion": "2017-11-04T18:48:46.258Z"
}
```

Response headers

```
connection: keep-alive
content-type: application/json
date: Fri, 13 Oct 2023 22:43:47 GMT
keep-alive: timeout=60
transfer-encoding: chunked
```

Responses

Code	Description	Links
200	OK	No links

Media type

/

Controls Accept header

Example Value | Schema

```
{
  "id": 0,
  "nombre": "string",
  "estatus": "string",
  "especie": "string",
  "genero": "string",
  "imagen": "string",
  "numero_episodios": "string",
  "fecha_creacion": "string"
}
```

Con esto finalizo ele ejercicio técnico, gracias por la comprensión y oportunidad.

GRACIAS Y SALUDOS COORDIALES