CURSO PROGRAMACIÓN FULLSTACK

GUÍA SPRING - API REST

EGG

INTRODUCCIÓN

En el año 2000, Roy Fielding propuso una nueva idea para diseñar servicios web llamada el modelo REST. Desde entonces, los servicios web REST se han convertido en el estándar de la industria para construir aplicaciones y servicios web modernos.

Saber cómo diseñar adecuadamente una API REST es una de las habilidades más importantes que un desarrollador de software puede tener.

💡API significa (Application Programing interface) que en español significa interfaz de programación de aplicaciones. Las aplicaciones son cualquier tipo de software, y la interfaz es un contrato de comunicación que una aplicación habilita para permitir que otro software le solicite información o acciones a realizar, de ahí la parte de “programación”.

¿QUÉ ES REST?

REST (Representational State Transfer) es un estilo de arquitectura de software que se basa en principios clave, diseñados para permitir la comunicación entre sistemas a través de internet. Estos son los conceptos fundamentales:

- Recursos: En una API REST, todo es un recurso, como objetos, datos o entidades. Cada recurso se identifica de manera única mediante una URL.
- Verbos HTTP: Las operaciones CRUD se mapean a los métodos HTTP:
    - GET: Obtener información sobre un recurso.
    - POST: Crear un nuevo recurso.
    - PUT: Actualizar un recurso existente.
    - DELETE: Eliminar un recurso.
- Sin estado: Cada solicitud del cliente al servidor debe contener toda la información necesaria para procesar la solicitud, sin depender de estados anteriores. Esto simplifica la escalabilidad.
- Transferencia de Representación: Los recursos se transfieren entre el cliente y el servidor en formatos como JSON, XML o YAML. El cliente y el servidor pueden negociar el formato a través de encabezados HTTP.
- Interfaz uniforme: La interfaz de la API REST debe ser uniforme, lo que significa que las URLs y los métodos HTTP siguen convenciones estandarizadas.

¿QUÉ ES RESTFUL?

Así como REST es el estilo de Arquitectura, RESTful es la implementación de dicha arquitectura.

Entonces API RESTful es un API que fue diseñada siguiendo los principios de la arquitectura REST.

Hay definiciones de RESTful que involucran los llamados “Niveles de madurez”, Hay expertos que consideran que aplicaciones de nivel 2 y 3 son RESTful y otros que proponen que solo las aplicaciones de nivel 3 lo son.

¿QUÉ SON LOS NIVELES DE MADUREZ?

Las API REST pueden categorizarse en diferentes niveles de madurez, siendo el nivel 3 el más completo. Los niveles son los siguientes:

Nivel 0 - transporte HTTP: En este nivel simplemente se usa HTTP como medio de transporte para hacer llamadas remotas sin usar la semántica de la web.En este nivel de madurez las URLs suelen incluir verbos que es una mala práctica, como “/addMessage”, “/deleteMessage” y “/getMessage”.

Nivel 1 - Recursos: En este nivel se aplican varias convenciones en las URLs, una “/” representa una relación jerárquica entre diferentes recursos, y se el plural para los nombres, como por ejemplo “/message”.

Nivel 2 - Verbos HTTP: Las operaciones que se realizan sobre los recursos son las operaciones de creación, obtención, actualización y eliminación o CRUD. Usando los diferentes verbos del protocolo HTTP es posible asignar a cada uno de ellos las diferentes operaciones básicas de manipulación de datos. Se usan los códigos de estado HTTP, que son números que indican el resultado de la operación:

- 200: la operación se ha procesado correctamente.
- 201, CREATED: un nuevo recurso ha sido creado.
- 400, BAD REQUEST: la respuesta es inválida y no puede ser procesada. La descripción del mensaje de error puede ser devuelta en lo datos retornados.
- 401, UNAUTHORIZED: acceder o manipular el recurso requiere autenticación.
- 403, FORBIDDEN: el servidor entiende la petición pero las credenciales proporcionadas no permiten el acceso.
- 404, NOT FOUND: el recurso de la URL no existe.
- 500, INTERNAL SERVER ERROR: se ha producido un error interno al procesar la petición por un fallo de programación. En la respuesta no siempre se devuelve una descripción del error, sin embargo en las trazas del servidor debería haber información detallada del error.

Tanto para enviar datos como obtener datos el formato utilizado es JSON.

Nivel 3 - HATEOAS (Hypermedia as the Engine of Application State / Hipertexto como motor del estado de la aplicación): Este nivel consta de dos partes: negociación de contenido y descubrimiento de enlaces en recursos.

La negociación de contenido permite que el cliente especifique el formato en el que desea recibir los datos. Esto se hace a través de la cabecera "Accept" en la solicitud. Por ejemplo, un cliente que

consume un servicio REST y desea los datos en formato JSON debe incluir la cabecera "Accept: application/json". Si prefiere los datos en formato XML, utilizará "Accept: application/xml". Si el servicio no es capaz de proporcionar los datos en el formato solicitado o no admite ese tipo de datos, se devolverá el código de estado 415, indicando que el formato de datos no es compatible.

En la web, las páginas están interconectadas mediante enlaces. El principio de HATEOAS se aplica al incluir enlaces en los datos de las entidades. Esto permite a los clientes navegar entre las entidades y descubrir las acciones disponibles. En otras palabras, los enlaces proporcionan un camino para explorar y acceder a más información y funcionalidad.

Muchas implementaciones de servicios REST optan por no llegar a este nivel debido a su complejidad. Además, si se ignoraran o no usarán los links a otros recursos significa que no se aprovecharán sus beneficios.

API REST NIVEL 2 CON SPRING

Nosotros nos centraremos en construir aplicaciones de nivel 2 porque es lo más común dentro del mercado.

Para eso explicaremos más detenidamente las convenciones que debes seguir, pero comenzaremos desde el principio.

Al igual como venías trabajando primero debes definir tu modelo de datos, es decir, las entidades que se usaran en tu aplicación.

Luego crearas la capa de acceso a datos con Spring Data JPA y la capa de servicios donde aplicaras las “reglas del negocio”.

Esas capas se mantienen igual a como veníamos trabajando. La diferencia estará en los controladores que ya no devolverán una página HTML, sino que deberán recibir y proveer información en formato JSON.

Parece que la complejidad aumentó, sin embargo, estamos usando el framework spring que ya resolvió la mayor parte de la complejidad para los desarolladores puedan crear sus API REST sin problemas. Veamos como lo resolvió:

@RESTCONTROLLER

Spring tiene la anotación @RestController que suplanta a @Controller, esta anotación permite convertir automaticamente objetos de Java a JSON, de esta manera en nuestros métodos solo debemos poner el tipo de dato que queremos devolver y Spring se encargará de devolver la respuesta en formato JSON:

```
@RestController
@RequestMapping("/api/users")
public class MyRestController {

    @GetMapping()
    public User getUser() {
        User user = new User();
        user.setName("Martín");
        return user;
    }
}
```

Además puedes agregar la anotación @ResponseStatus(HttpStatus.OK) al método para especificar el código de respuesta HTTP cuando la solicitud sea procesada correctamente, en este caso “200”.

```
@RestController
@RequestMapping("/api/users")
public class MyRestController {

    @GetMapping()
    @ResponseStatus(HttpStatus.OK)
    public User getUser() {
        User user = new User();
        user.setName("Martín");
        return user;
    }
}
```

@REQUESTBODY

En el caso de que queramos recibir datos para guardar o modificar una entidad ahora deberemos usar la anotación @RequestBody acompañado del tipo de objeto java que esperamos recibir, spring se encargará automáticamente de serializar el JSON recibido y convertirlo a un objeto Java. En el caso de que el body JSON no cumpla con las propiedades del objeto Java esperado spring automáticamente devolverá una respuesta de “BAD REQUEST 400” al cliente, que es el mensaje de error esperado cuando el body de la solicitud es incorrecto.

```
@RestController
@RequestMapping("/api/users")
public class MyRestController {

    @Autowired
    private UserService userService;

    @GetMapping()
    @ResponseStatus(HttpStatus.OK)
    public List<User> getUsers() {
        return userService.findAll();
    }

    @PostMapping()
    @ResponseStatus(HttpStatus.CREATED)
    public User postUser(@RequestBody User user) {
        return userservice.save(user);
    }
}
```

Observa como el @ResponseStatus() usa la constante “CREATED” en lugar de “OK”, esto es porque al crear un nuevo registro en la base de datos, por convención, se devuelve el código de respuesta “201”.

Aunque se pueden usar las entidades que mapean la base de datos como body de las solicitudes, no es lo recomendable, para eso se suelen crear DTOs.

¿QUÉ SON LOS DTOS?

Los DTO (data transfer object) son objetos diseñados específicamente para transportar datos entre diferentes capas de una aplicación o entre distintas aplicaciones. Proporcionan una manera eficiente y estructurada de mover datos, evitando la exposición innecesaria de detalles internos que hacen referencia a la base de datos.

Los DTO en JAVA son simplemente POJOs (Plain Old Java Objects) que son clases simples de java con atributos y sus correspondientes métodos getters, setters, equals y hashcode.

Se suelen poner juntos en un paquete y por convención suelen tener el sufijo DTO. En nuestro caso se llamaría UserDTO:

```
...
    @GetMapping()
    @ResponseStatus(HttpStatus.OK)
    public List<UserDTO> getUsers() {
        return userService.findAll();
    }

    @PostMapping()
    @ResponseStatus(HttpStatus.CREATED)
    public UserDTO postUser(@RequestBody UserDTO user) {
        return userservice.save(user);
    }

...
```

Al usar DTOs también deberemos usar clases “converter” que se encargarán de tener métodos que conviertan un UserDTO en un User y viceversa.

CONVENCIONES SOBRE LAS URLS

Las URLs en una aplicación REST son importantes porque proporcionan un medio para identificar y acceder a los recursos, y deben seguir ciertas convenciones para mantener la consistencia y la claridad en el diseño de la API.

Convenciones para Crear Estructuras de URL en Aplicaciones REST:

- Mantén las URLs significativas y descriptivas: Deben representar los recursos de manera clara y comprensible. Por ejemplo, en una API de gestión de usuarios, una URL como /users es más intuitiva que /data o /info.
- Utiliza sustantivos en plural para los recursos: Por convención, los sustantivos en plural se utilizan para identificar los recursos. Por ejemplo, /users en lugar de /user para representar una colección de usuarios.
- Utiliza rutas anidadas para relaciones: Si tienes recursos relacionados, puedes utilizar rutas anidadas para representar estas relaciones. Por ejemplo, /users/123/orders para

obtener los pedidos del usuario con el ID 123 o tambien products/456/reviews para ver las revisiones de un producto con el ID 456. Esta convención facilita la navegación de la API y permite a los clientes acceder de manera intuitiva a los recursos relacionados sin necesidad de rutas complicadas.

- Sigue una estructura jerárquica: Al seguir una estructura jerárquica en las URLs, se refleja la organización de los recursos de una manera que imita la jerarquía en la vida real o en la lógica de tu aplicación. Esto significa que los recursos se organizan en una estructura similar a un árbol, con recursos principales y subrecursos anidados. Por ejemplo, “departments/123/employees/456”, en este caso, la estructura refleja una jerarquía organizativa donde los empleados están vinculados a departamentos. Esta organización jerárquica puede ayudar a los clientes a navegar la API y comprender las relaciones entre los recursos.
- Utiliza guiones bajos o guiones para separar palabras en una URL: Es común utilizar guiones bajos (_) o guiones (-) para separar palabras en una URL, lo que hace que las URL sean más legibles. Por ejemplo, /product-reviews o /product_reviews para acceder a revisiones de productos.

Cosas a Evitar - Malas Prácticas:

- Longitudes excesivas de URL: Evita URLs extremadamente largas, ya que pueden ser difíciles de gestionar y pueden causar problemas de legibilidad.
- Uso de verbos en las URLs: En REST, los verbos (como GET, POST, PUT, DELETE) deben representar las acciones, no las URL. Evita URLs como /get-user/123 o /delete-order/456, y en su lugar utiliza el verbo GET junto con la url /users/123 para obtener un usuario y el verbo DELETE y la url /orders/456 para eliminar un pedido.
- Parámetros complejos en la URL: Evita incluir parámetros complejos en la URL “/resource/category/beauty/soirt/price/descendant”. Es preferible utilizar parámetros de query, como “/resource?category=beauty&sort=price,desc” en lugar de rutas complicadas.
- Usar sustantivos en singular: No utilices sustantivos en singular para identificar colecciones de recursos, ya que esto puede resultar en una falta de claridad y consistencia.
- No usar sufijos de archivo en las URL: No es necesario incluir extensiones de archivo, como .html o .json, en las URLs de una API REST. La representación del recurso debería ser manejada a través de encabezados o parámetros de solicitud.
- Evita el uso de mayúsculas en las URLs: Por convención, se recomienda utilizar letras minúsculas en las URLs para mantener la consistencia y evitar problemas de sensibilidad a mayúsculas y minúsculas.
- Evita anidamientos o jerarquias de mas de dos niveles: En lugar de tener una URL “/users/123/orders/321/products/456/reviews” mantén las url de hasta dos niveles de anidamiento, en este caso “/users/123/orders”, “/orders/321/products” y “products/456/reviews”.

@PATHVARIABLE

Como ya vimos en la construcción de URLs para las aplicaciones REST hay datos de la URL que varían como el ID del recurso al que se quiere acceder, para poder construir estas rutas de manera programática en Spring se utiliza el @PathVariable:

```
...
    @GetMapping("/{id}")
    @ResponseStatus(HttpStatus.OK)
    public UserDTO getUser(@PathVariable String id) {
        return userService.getById(id);
    }
...
```

Observa como la variable de ruta se encuentra en llaves, además ten en cuenta que si el nombre del PathVariable en la ruta no coincide con el nombre de la variable definida como argumento del método debes mapear el nombre de la variable:

```
...
    @GetMapping("/{id}")
    @ResponseStatus(HttpStatus.OK)
    public UserDTO getUser(@PathVariable("id") String idUser) {
        return userService.getById(idUser);
    }
...
```

Esto se aplica al resto de los verbos también:

```
...
    @PutMapping("/{id}")
    @ResponseStatus(HttpStatus.OK)
    public UserDTO updateUser(@PathVariable("id") String idUser,
    @BodyRequest UserDTO user) {
        return userService.update(idUser, user);
    }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteUser(@PathVariable("id") String idUser) {
        userService.deleteById(idUser);
    }
...
```

Nota: Se recomienda no eliminar registros de la base de datos, en su lugar tener un atributo como por ejemplo “active” para darlo de baja y en las consultas a las bases de datos ignorar los inactivos.

ANIDAMIENTO Y JERARQUÍA

En el caso de que queramos hacer un endpoint para obtener una lista de objetos relacionados con nuestro recurso podemos usar el anidamiento

```
...
    @GetMapping("/{id}/projects")
    @ResponseStatus(HttpStatus.OK)
    public List<ProjectsDTO> getProjectsByUser(@PathVariable("id")
     String idUser) {
        return userService.getProjectsByUser(idUser);
    }
...
```

También puedes devolver un objeto DTO que dentro tenga la lista de proyectos

```
...
    @GetMapping("/{id}/projects")
    @ResponseStatus(HttpStatus.OK)
    public UserProjectsDTO getProjectsByUser(@PathVariable("id")
     String idUser) {
        return userService.getProjectsByUser(idUser);
    }
...
```

Recuerda no abusar de las jerarquías, si quieres que las proyectos sean accesibles sólo por medio del restController de los usuarios es una decisión de negocio y puede tener lo siguiente

```
...
    @GetMapping("/{id}/projects/{idProject}")
    @ResponseStatus(HttpStatus.OK)
    public Project getProjectsByIdAndUser(@PathVariable("id")
     String idUser, @PathVariable("idProject") String idProject ) {
        return userService.getProjectsByIdAndUser(idProject, idUser);
    }
...
```

Pero por buena práctica no anides más de dos niveles, en el caso de que quieras acceder a las tareas del proyecto crea un nuevo RestController de proyectos.

MANEJO DE EXCEPCIONES

Para el manejo de excepciones se pueda usar un bloque try catch en los controladores para atrapar las excepciones que se producen.

Se recomienda utilizar la clase ResponseStatusException de Spring que te permite lanzar excepciones con códigos de estado HTTP. Aquí hay un ejemplo de cómo crear y lanzar una excepción ResponseStatusException:

```
...
    public UserDTO getUserById(String id) {
        User user = userService.getUserById(id);
        if (user == null) {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND,
              "Usuario no encontrado");
        }
        return userConverter(user);
    }
...
```

Aquí puedes ver como capturar las distintas excepciones en el controlador:

```
...
    @GetMapping("/{id}")
    @ResponseStatus(HttpStatus.OK)
    public UserDTO getUser(@PathVariable("id") String idUser) {
        try {
             return userService.getById(idUser);
        } catch (ResponseStatusException ex) {
            e.printStackTrace();
            throw new ResponseStatusException(e.getStatus(),
               e.getMessage());
        } catch (Exception ex) {
            e.printStackTrace();
            throw new ResponseStatusException(
               HttpStatus.INTERNAL_SERVER_ERROR, "Ocurrió un error");
    }
...
```

En este ejemplo, si el usuario con el ID especificado no se encuentra, se lanza una excepción ResponseStatusException con el código de estado HTTP 404 (NOT FOUND) y un mensaje de error personalizado.

@RestControllerAdvice:

@RestControllerAdvice es una anotación en Spring que te permite definir una claseque manejan excepciones globalmente en toda tu aplicación. Puedes crear una clase anotada con @RestControllerAdvice y métodos anotados con @ExceptionHandler para manejar excepciones específicas. Aquí hay un ejemplo:

```
...
@RestControllerAdvice
public class RestControllerExceptionHandler {

    @ExceptionHandler(ResponseStatusException.class)
    public ResponseEntity<ErrorMessage>
       responseStatusExceptionHandler(ResponseStatusException ex) {
        ErrorMessage message = new ErrorMessage(
                ex.getStatus().value(),
                ex.getMessage());
        return new ResponseEntity<>(message, ex.getStatus());
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorMessage> globalExceptionHandler(Exception
     ex) {
        ErrorMessage message = new ErrorMessage(
                HttpStatus.INTERNAL_SERVER_ERROR.value(),
                ex.getMessage());
        return new ResponseEntity<>(message,
                 HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
...
```

La clase ErrorMessage es una clase DTO que tiene los atributos statusCode de tipo int y message de tipo String. ResponseEntity es una clase de Spring que permite controlar y personalizar la respuesta que se envía al cliente cuando se procesa una solicitud web, incluyendo el estado HTTP, el cuerpo de la respuesta y las cabeceras, en este caso solo estamos usando el cuerpo y el estado HTTP.

En el ejemplo, la clase GlobalExceptionHandler maneja excepciones ResponseStatusException y devuelve una respuesta HTTP apropiada, además hay un método que atrapa a cualquier excepción que no sea controlada y devuelve un código de status 500. Puedes definir métodos y clases de excepción personalizadas para manejar diferentes tipos de excepciones en esta clase global, lo que simplifica la gestión de errores en toda tu aplicación.

CORS

CORS significa "Cross-Origin Resource Sharing" (Compartir recursos entre distintos orígenes) y es un mecanismo de seguridad que se aplica en los navegadores web para controlar las solicitudes y respuestas entre dos dominios diferentes. CORS es necesario para garantizar que los navegadores restrinjan las solicitudes de recursos (como datos JSON o imágenes) desde un dominio web a otro, a menos que se especifiquen explícitamente como seguras y permitidas.

Cómo se configura CORS en Spring:

Para habilitar CORS en una aplicación Spring, puedes usar la anotación @CrossOrigin o configurar CORS globalmente en la aplicación. Aquí hay un ejemplo de ambas opciones:

Opción 1: Anotación @CrossOrigin en un controlador:

```
@RestController
@RequestMapping("/api")
@CrossOrigin(origins = "*")
public class ApiController {


    @GetMapping("/resource")
    public ResponseEntity<String> getResource() {
        return ResponseEntity.ok("Recibido desde un dominio permitido");
    }
}
```

En este ejemplo, se permite el acceso a la ruta /api/resource desde cualquier origen. En el caso de que quieras configurar más de una ruta, debes usar llaves y separar los elementos por comas:

```
...
@RestController
@RequestMapping("/api")
@CrossOrigin(origins = {"http://dominio1", "http://dominio2"})
public class ApiController {
...
```

Puedes configurar varios atributos en @CrossOrigin según tus necesidades. Aquí tienes el link a la documentación:

[CrossOrigin documentación]

Opción 2: Configuración global de CORS:

Puedes configurar CORS globalmente en tu aplicación Spring utilizando la clase WebMvcConfigurer:

```
@Configuration
public class CorsConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
                .allowedOriginsPattern("*");
    }
}
```

Esta configuración permite el acceso a todas las rutas bajo /api desde cualquier origen, si solo quieres permitir que se pueda acceder desde un solo dominio debes modificar el asterisco dentro de del método por el dominio que tú quieras.

```
...
        registry.addMapping("/api/**")
                .allowedOriginsPattern("https://dominio1.com",
                                       "https://dominio2.com");
...
```

¿CÓMO PROBAMOS NUESTRA API REST?

Probar una API REST es una parte crucial para comprobar el funcionamiento de nuestros endpoints. Si bien se puede iniciar una solicitud HTTP en un navegador web para interactuar con una API, esta forma de prueba tiene limitaciones significativas. A continuación se introducen herramientas como cURL y Postman.

CURL

CURL es una herramienta de línea de comandos utilizada para realizar solicitudes HTTP y transferir datos con varias URL. Es una herramienta muy versátil y potente que permite interactuar con una API REST de manera programática y realizar solicitudes de diferentes tipos, como GET, POST, PUT y DELETE, entre otros.

Verificar que lo tenemos:

Puedes verificar si tienes cURL instalado en tu sistema ejecutando el siguiente comando en la línea de comandos:

```
curl --version
```

Cómo instalarlo:

En sistemas Unix (Linux o macOS), cURL generalmente ya está instalado de forma predeterminada. Puedes comprobarlo con el comando anterior.

En Windows, puedes descargar cURL desde el sitio web oficial de cURL:

https://curl.se/download.html

Cómo usarlo con sus reglas y comandos:

cURL se utiliza a través de comandos en la línea de comandos. Aquí hay un ejemplo de cómo realizar una solicitud GET a una API REST con cURL:

```
curl -X GET https://api.example.com/users
```

-X especifica el método HTTP (GET en este caso).

La URL (https://api.example.com/users) es la dirección de la API que deseas acceder.

Puedes realizar solicitudes POST, PUT, DELETE también, agregando un body si es necesario:

```
curl -X POST -d '{"nombre": "Ejemplo", "edad": 30}'
https://api.example.com/endpoint
```

-X POST especifica que estás realizando una solicitud POST.

-d o --data se utiliza para proporcionar los datos del cuerpo en formato JSON. Debes encerrar los datos JSON entre comillas simples o dobles, como se muestra en el ejemplo.

https://api.example.com/endpoint es la URL a la que estás enviando la solicitud.

POSTMAN

Postman es una herramienta más avanzada para probar y documentar API. A diferencia de cURL, Postman proporciona una interfaz gráfica de usuario que facilita la creación, envío y administración de solicitudes HTTP a API RESTful. Además, Postman ofrece características como la organización de solicitudes en colecciones para poder agrupar los endpoints de distintas apis.

Si te cansas de usar la línea de comandos puedes usar Postman para realizar las mismas solicitudes HTTP de manera más intuitiva. Aquí hay una guía rápida para comenzar con Postman:

Primeros Pasos con Postman:

- Ve al sitio oficial de Postman
- Descarga e instala la aplicación según tu sistema operativo.
- Crear una Cuenta (Opcional): Aunque no es obligatorio, crear una cuenta en Postman te permite sincronizar tus solicitudes y colecciones en la nube.
- Inicia la aplicación Postman.
- Crear una Colección: En la barra lateral izquierda, haz clic en "Collections" y luego en el botón “+” (Create new Collection). Asigna un nombre a tu colección y guarda.
- Crear una Solicitud: Dentro de tu colección, haz clic en "Add Request". Asigna un nombre a tu solicitud.
- Configurar la Solicitud: Selecciona el método HTTP (GET, POST, PUT, DELETE, etc.) que deseas probar. Ingresa la URL del endpoint que deseas llamar.
- Enviar la Solicitud: Haz clic en "Send" para enviar la solicitud.
- Observa la respuesta en la parte inferior de la ventana, que incluirá el código de estado, el cuerpo de la respuesta y las cabeceras.

EJERCICIOS

1) Crea una entidad llamada Cliente con los siguientes atributos: id (como clave principal), nombre, email, teléfono, y cualquier otro atributo que desees.
2) Crea una entidad llamada Order con los siguientes atributos: id (como clave principal), fecha, cliente (una relación con la entidad Cliente), y cualquier otro atributo que consideres necesario.
3) Crea una entidad llamada Product con los siguientes atributos: id (como clave principal), nombre, precio, y cualquier otro atributo que desees.
4) Crea un repositorio para la entidad Cliente que permita realizar operaciones CRUD (Create, Read, Update, Delete) en la base de datos.
5) Crea un repositorio para la entidad Order que permita realizar operaciones CRUD.
6) Crea un repositorio para la entidad Product que permita realizar operaciones CRUD.
7) Crea un servicio para la entidad Cliente que contenga métodos para crear, leer, actualizar y eliminar clientes. Utiliza DTOs para mapear datos entre el controlador y el servicio.
8) Crea un servicio para la entidad Order con métodos para gestionar pedidos. Utiliza DTOs para transferir datos entre el controlador y el servicio.
9) Crea un servicio para la entidad Product con métodos para administrar productos. Emplea DTOs para la transferencia de datos.
10) Implementa convertidores (mappers) para convertir entre las entidades y los DTOs en cada servicio. Asegúrate de que los datos se transfieran de manera correcta.

11) Crea un controlador REST para la entidad Cliente con métodos para realizar operaciones CRUD, es decir, GET (obtener todos los clientes, obtener un cliente por ID), POST (crear un nuevo cliente), PUT (actualizar un cliente existente) y DELETE (eliminar un cliente).
12) Crea un controlador REST para la entidad Order con métodos para operaciones CRUD.
13) Crea un controlador REST para la entidad Product con métodos para operaciones CRUD.
14) Asegúrate de anotar tus clases y métodos con las anotaciones adecuadas, como @RestController, @RequestMapping, @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, etc.
15) Prueba tus endpoints con CURL o Postman.