UD03 – Servicios REST

Conocimientos previos

Antes de comenzar:

Tomcat: Apache Tomcat es un servidor web y un contenedor de servlets de código abierto. Es utilizado para implementar y ejecutar aplicaciones web escritas en Java.

Tomcat implementa varias tecnologías de Java EE como Java Servlet, JavaServer Pages (JSP), Java EL, y WebSocket, y proporciona un entorno de servidor web en el que se pueden ejecutar estas aplicaciones.

Es importante destacar que Tomcat es a menudo referido como un "contenedor de servlets", ya que es capaz de proporcionar las funcionalidades que requieren las aplicaciones web basadas en servlets. Sin embargo, a diferencia de otros servidores de aplicaciones Java EE, Tomcat no soporta EJB (Enterprise JavaBeans) u otras características empresariales de Java EE de manera nativa. Para esas características, se necesitaría un servidor de aplicaciones completo como WildFly, OpenLiberty, o Payara.

Bean: Componente o clase Java, registrada en el contenedor de Spring. Los Beans son la esencia del framework de Spring, ya que controlan la creación de objetos, la configuración de los mismos y la gestión de su ciclo de vida.

Patrones de diseño: (ED) Los patrones de diseño son soluciones probadas a problemas comunes en el diseño de software. Proporcionan un marco reutilizable que puede ser adaptado a las necesidades específicas de su código. Algunos ejemplos comunes incluyen el Singleton, Factory, y Observer.

https://www.youtube.com/watch?v=6BHOeDL8vls

https://www.youtube.com/watch?v=cwfuydUHZ7o&list=PLvimn1Ins-41Uiugt1WbpyFo1XT1WOquL

CamelCase vs snakecase: convenciones de nomenclatura en programación:

CamelCase: En esta convención, la primera letra de cada palabra se escribe en mayúscula y no se utilizan espacios. Por ejemplo, “myVariableName”.

“snake_case”: En esta convención, las palabras se separan con guiones bajos (_) y todo se escribe en minúsculas. Por ejemplo, “my_variable_name”.

La elección entre CamelCase y snake_case puede depender del lenguaje de programación, el equipo de desarrollo, o las guías de estilo de código.

Pojo: POJO es un acrónimo de Plain Old Java Object. Es un término utilizado en Java para enfatizar que un objeto no necesita ser parte de una estructura compleja de herencia o implementar algún marco o interfaz predefinido.

Un POJO es un objeto Java ordinario, no ligado por ninguna restricción especial más allá de las de Java, por lo que no necesita extender una clase predefinida, implementar una interfaz específica o contener anotaciones específicas.

Estos objetos son simples de construir y comprender, y pueden ser útiles para crear modelos de datos.

Lombok: Lombok es una biblioteca de Java que se utiliza para reducir el código repetitivo, como los métodos getter, setter, constructores y código de registro, entre otros. Lombok proporciona anotaciones que puedes usar en tus clases para generar automáticamente este tipo de código.

MVC

MVC es el acrónimo de Modelo-Vista-Controlador, que es un patrón de diseño de software comúnmente utilizado para desarrollar interfaces de usuario.

- Modelo: Representa los datos y las reglas de negocio. Es independiente de la representación de la interfaz de usuario.
- Vista: Es la representación visual de los datos del modelo. Muestra la información al usuario.
- Controlador: Actúa como intermediario entre el modelo y la vista. Maneja la entrada del usuario y actualiza el modelo y/o la vista en consecuencia.

El usuario interactúa con la Vista, que pasa las interacciones del usuario al Controlador. El Controlador procesa estas interacciones, puede hacer cambios en el Modelo y finalmente actualiza la Vista para reflejar cualquier cambio en los datos.

Spring

Spring es un framework, probablemente el framework más importante del mundo Java.

Web del framework:

https://spring.io/

Editores:

https://spring.io/tools

En VSC → extension spring boot extension pack

Generador de proyectos:

https://start.spring.io/

Qué es spring boot

Spring es un framework, spring boot es una herramienta que mejora, extiende spring.

Es rápido, seguro, incluye un tomcat embebido

Se encarga de versiones de dependencia auto-administradas.

Integra herramientas de desarrollo “dev-tools”

Integra métricas de monitoreo, auditing, health.

Listo para producción (jar)

Integra: JPA, Mongo, Seguridad, Testing, Microservicios, Cloud, Reactivo (app asíncronas.)

Esteriotipos de Spring:

Los estereotipos de Spring Framework ayudan a definir el papel de una clase en una aplicación Spring:

@Component: Es una anotación genérica para cualquier componente gestionado por Spring. Las clases anotadas con “@Component” son detectadas automáticamente por Spring durante el escaneo de clases.

@Controller: Es una anotación específica para las clases que actúan como controladores en una aplicación Spring MVC. Estas clases son responsables de procesar las solicitudes del usuario.

@Service: Se utiliza en las clases que proporcionan algún tipo de servicio de negocio (o lógica de negocio). Es una especialización de “@Component” que no añade ningún comportamiento adicional, pero ayuda a indicar la intención de la clase.

@Repository: Se utiliza en las clases que acceden a los datos, por lo general desde una base de datos. Esta anotación es una especialización de “@Component” que proporciona traducción de excepciones de base de datos a las excepciones de Spring.

@Bean: No es un estereotipo, sino una anotación que se utiliza para indicar que un método produce un bean que debe ser gestionado por el contenedor de Spring. Los métodos anotados con “@Bean” se suelen definir dentro de una clase anotada con “@Configuration”.

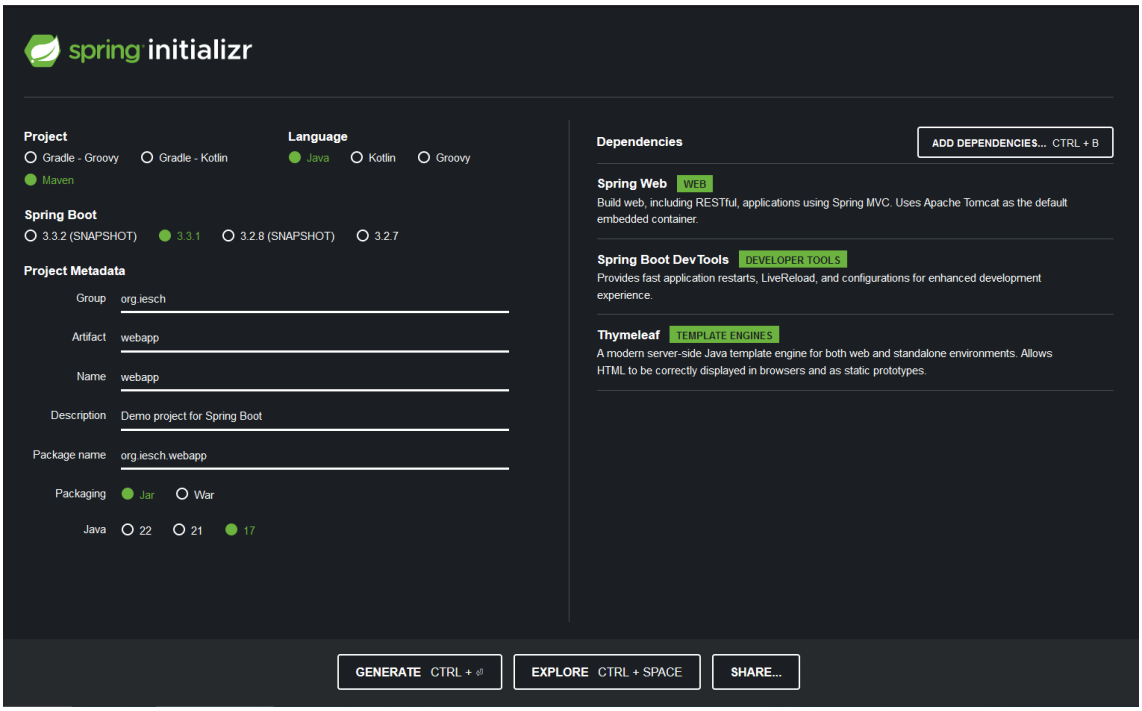
Estas anotaciones ayudan a Spring a entender las responsabilidades de las clases en tu aplicación y a gestionar automáticamente la creación y conexión de objetos.

Creando un primer proyecto MVC web

Nos dirigimos a la web initializer

https://start.spring.io/

Generamos un proyecto siguiendo el modelo:

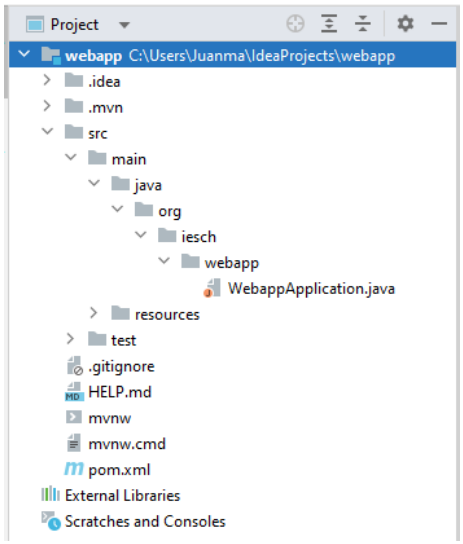


Siendo:

- Spring web una dependencia básica para crear webs, Apis, etc.
- Dev-tools dependencia que entre otras cosas nos ofrece un servidor apache ligth-reload
- Thymelaef, un motor de plantillas.

Nota: el package indicado, es la raíz del proyecto, nada podrá estar fuera de este package

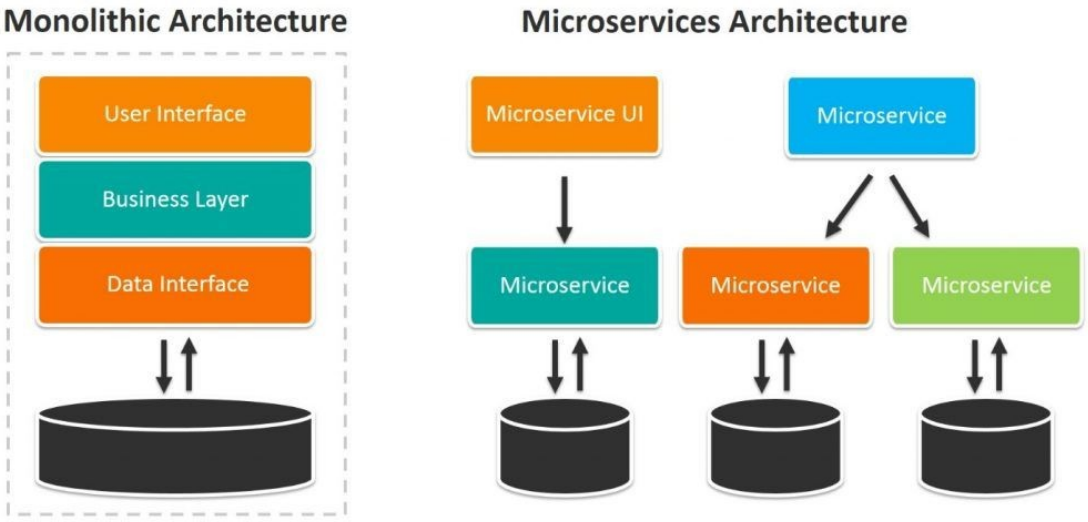
Hacemos clic en generate, esto nos descarga un zip, este zip tiene un proyecto que debemos importar en nuestro Intellij.

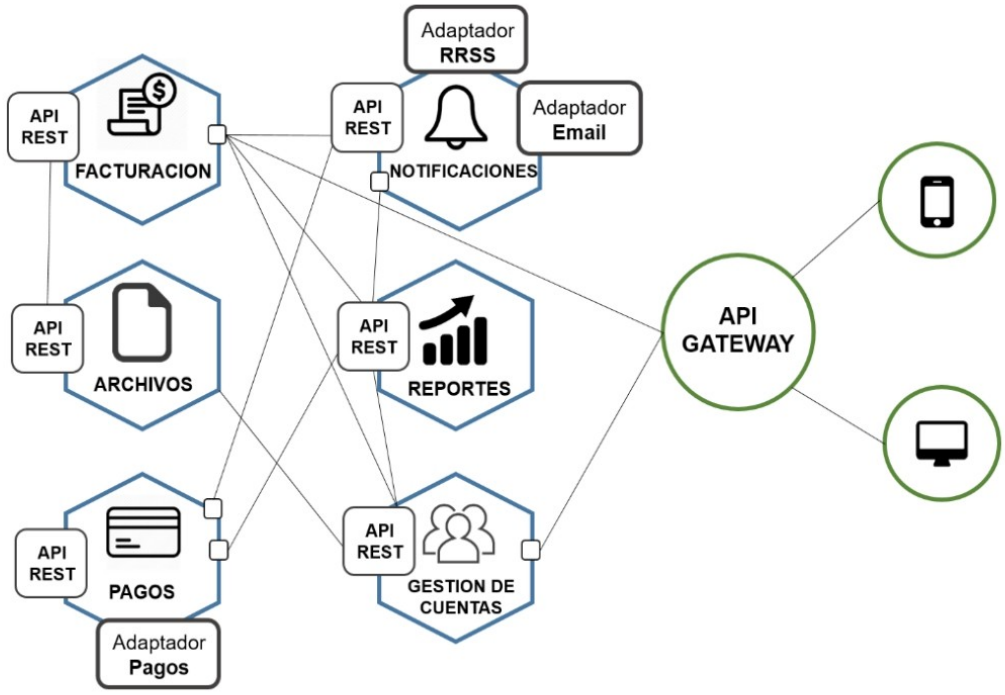


Aplicación monolítica vs microservicios.

Aplicación Monolítica: En una aplicación monolítica, todos los componentes de la aplicación (la interfaz de usuario, el código de procesamiento, la lógica de negocio, el acceso a la base de datos, etc.) están todos interconectados y gestionados como una sola unidad. Si necesitas escalar, tienes que escalar todo el monolito, lo que puede ser costoso y complicado. Además, si una parte de la aplicación falla, puede afectar a toda la aplicación.

Microservicios: En la arquitectura de microservicios, la aplicación se divide en servicios más pequeños e independientes que se ejecutan en su propio proceso y se comunican entre sí a través de APIs. Cada microservicio puede ser desarrollado, desplegado, operado y escalado de forma independiente. Esto permite una mayor flexibilidad y puede hacer que la aplicación sea más resistente, ya que si un servicio falla, los demás pueden seguir funcionando. Sin embargo, los microservicios pueden ser más complejos de gestionar debido a la necesidad de coordinar múltiples servicios y mantener la coherencia de los datos.





Aparece el concepto de Api REST, a partir de ahora vamos a empezar a crear Apis Rest

Mi primera Api Rest

REST, que significa Representational State Transfer, es un estilo de arquitectura para diseñar redes de aplicaciones. Fue definido y nombrado por Roy Fielding, un científico de la computación, en su tesis doctoral en el año 2000.

Una API REST es una interfaz de programación de aplicaciones que sigue los principios de REST. Estas APIs permiten la interacción entre múltiples software a través de HTTP, permitiendo operaciones como CREATE, READ, UPDATE y DELETE (CRUD) en los datos.

Las API REST son stateless, lo que significa que cada solicitud debe contener toda la información necesaria para procesarla. El servidor no guarda ningún contexto entre peticiones.

Las API REST utilizan los métodos HTTP estándar (GET, POST, PUT, DELETE, etc.) para realizar operaciones, y pueden devolver datos en diferentes formatos, aunque el formato JSON es el más común.

Las API REST son populares en el desarrollo web debido a su simplicidad, su eficiencia y su compatibilidad con las tecnologías web existentes. Son utilizadas por muchas grandes empresas como Twitter, YouTube, y Facebook para proporcionar servicios a sus aplicaciones web y móviles.

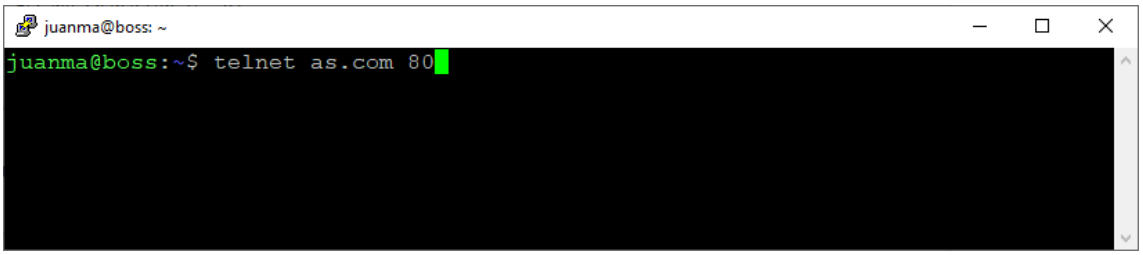
Cómo funciona un navegador internamente:

Para desarrollar una Api, lo primero que debemos comprender es como funciona el protocolo http (parte de PSP).

Un navegador cuando le indicamos una URL, se conecta a ella, y le indica al servidor que quiere obtener dicha página web. Esto lo hace haciendo uso de una operación conocida como “GET” vamos a simularlo desde una consola.

$ telnet [SERVER] [PORT]

telnet as.com 80



Una vez realiza la conexión el servidor se queda esperando a que le digamos que orden queremos realice.

Escribiremos:

GET [WEB PAGE] HTTP/1.1

HOST: [SERVER]

<Press ENTER>

Esto nos devuelve el código html de la url solicitada.

Acabamos de hacer uso de un verbo http, concretamente de GET

Más info: https://www.shellhacks.com/telnet-send-get-head-http-request/

Verbos HTTP o métodos HTTP

Los verbos HTTP, también conocidos como métodos HTTP, son indicadores que representan las acciones que se pueden realizar en un recurso específico en una API REST. Aquí están los más comunes:

- → **GET**: Se utiliza para recuperar datos de un recurso. No modifica ni elimina los datos existentes.
- → **POST**: Se utiliza para enviar datos a un recurso para su creación. También se utiliza a menudo para enviar datos a un servidor para procesarlos.
- → **PUT**: Se utiliza para actualizar un recurso existente. Debe proporcionar toda la representación del recurso, incluso si solo se modifican algunas partes.
- → **PATCH**: Similar a PUT, pero se utiliza para actualizar parcialmente un recurso. Solo necesita proporcionar los campos que se van a modificar.
- → **DELETE**: Se utiliza para eliminar un recurso.
- → **HEAD**: Similar a GET, pero solo recupera los encabezados de respuesta y no el cuerpo de la respuesta.
- → **OPTIONS**: Se utiliza para describir las opciones de comunicación para el recurso de destino.

Cada uno de estos verbos HTTP tiene un propósito específico y debe ser utilizado de acuerdo a las operaciones que se quieran realizar en el recurso.

Software para probar una API:

Postman: https://www.postman.com/

Bruno: https://www.usebruno.com/

HTTPie: https://httpie.io/

Etc.

Nota: consola con colores en intellij, en aplication.properties añade: spring.output.ansi.enabled=ALWAYS

Programando:

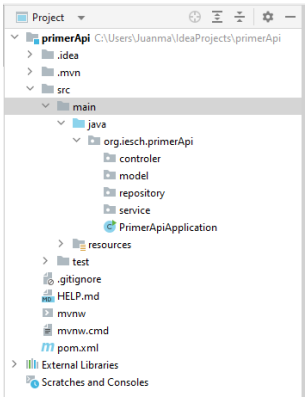
Crea un proyecto Spring con las siguientes dependencias:

Spring web y spring web dev tolos

Siempre que programamos con el patrón mvc, debemos crearnos los paquetes:

- → Controler: Ubicaremos nuestros controladores.
- → Model: Contendrá nuestros modelos.
- → Repository: Aquí desarrollaremos nuestras clases de acceso a datos DAO
- → Service: Se ubica la lógica de negocio.

Esta estructura, a demás de estar “estandarizada” permite que el software se encuentre desacoplado, por lo tanto, permite la reutilización.



Creamos nuestro primer controlador: “HolaMundo.java” evidentemente dentro del paquete controler.

```
package org.iesch.primerApi.controler;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HolaMundo {
    @GetMapping("/hola")
    public String holaMundo(){
        return "Hola Mundo";
    }
}
```

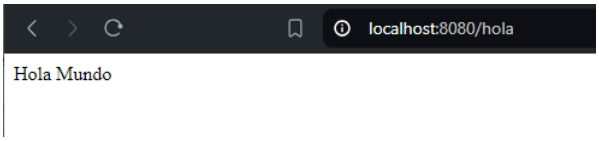
Podemos ver dos anotaciones:

`@RestController` para indicar que es un controlador de tipo REST

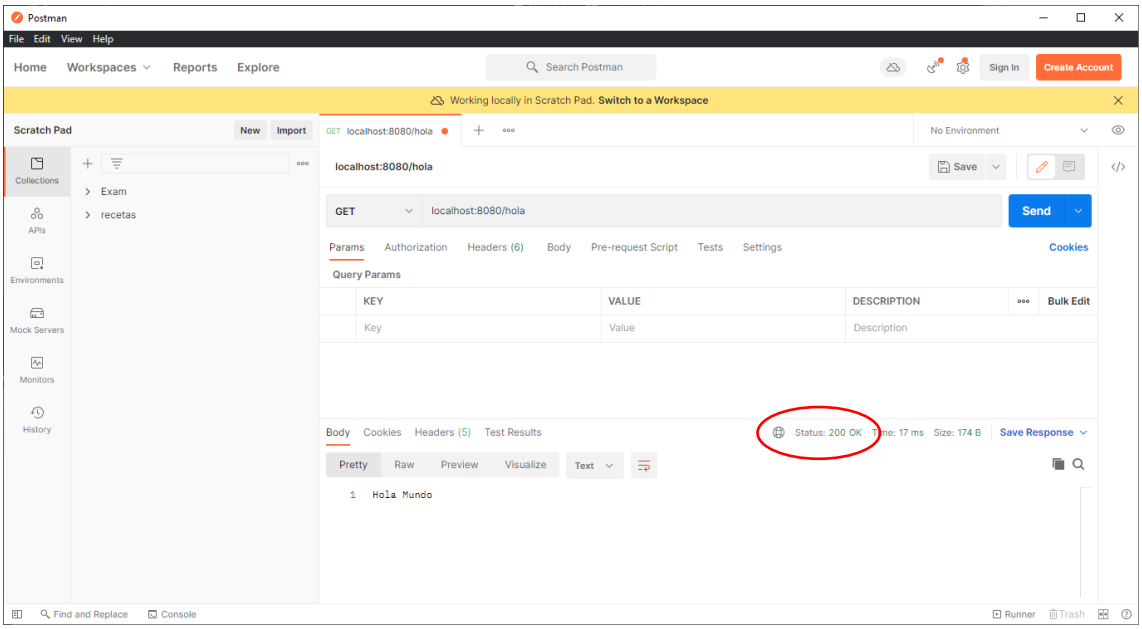
`@GetMapping` para indicar el verbo y la ruta o endpoint.

Lanzamos la aplicación, para ello lanzaremos la clase anotada como “@SpringBootApplication” esto nos autoconfigurará el proyecto y nos levanta y despliega el servidor tomcat.

Si abrimos un navegador y vamos a la url: localhost:8080/hola



También podemos consumir dicho servicio desde un móvil, una aplicación de escritorio, o usando el software específico para ello.



Códigos de estado:

Los códigos de estado HTTP son códigos de tres dígitos que indican el resultado de una solicitud HTTP. Son parte de la respuesta que el servidor envía después de procesar una solicitud. Aquí están algunos de los más comunes:

→ 200 OK: La solicitud ha tenido éxito. El significado de un éxito varía dependiendo del método HTTP: 200 OK para GET significa que el recurso ha sido recuperado y está en el cuerpo del mensaje.
→ 201 Created: La solicitud ha tenido éxito y se ha creado un nuevo recurso como resultado. Esta es típicamente la respuesta enviada después de una solicitud POST.
→ 204 No Content: La solicitud ha tenido éxito, pero no hay representación que devolver (es decir, el cuerpo del mensaje está vacío).
→ 400 Bad Request: La solicitud no se pudo entender o no se pudo cumplir debido a una sintaxis mal formada.
→ 401 Unauthorized: La solicitud requiere autenticación. El cliente debe pasar la autenticación antes de que se pueda conceder la respuesta.
→ 403 Forbidden: El cliente no tiene derechos de acceso al contenido; es decir, está no autorizado, por lo que el servidor está rechazando dar una respuesta apropiada.
→ 404 Not Found: El servidor no pudo encontrar el contenido solicitado. Este código de estado es comúnmente utilizado cuando el servidor no desea revelar exactamente por qué la solicitud ha sido rechazada, o si no hay otra respuesta más apropiada.
→ 500 Internal Server Error: El servidor ha encontrado una situación que no sabe cómo manejar. Es un código de respuesta genérico, utilizado cuando no se dispone de ningún código más específico.

Estos códigos de estado permiten al cliente entender si una solicitud ha sido exitosa, y si no, por qué no.

Anotación @PathVariable

Hasta ahora solo nos hemos conectado una url, y nos ha devuelto un saludo. ¿Pero qué ocurre si queremos pasarle información al servidor?

La anotación “@PathVariable” en Spring se utiliza para manejar variables de plantilla en la ruta de una URL. Se utiliza para extraer los valores de las variables y pasarlos como argumentos a los métodos del controlador.

Por ejemplo, si tienes una URL como “/users/{id}”, puedes usar “@PathVariable” para extraer el valor de “id”:

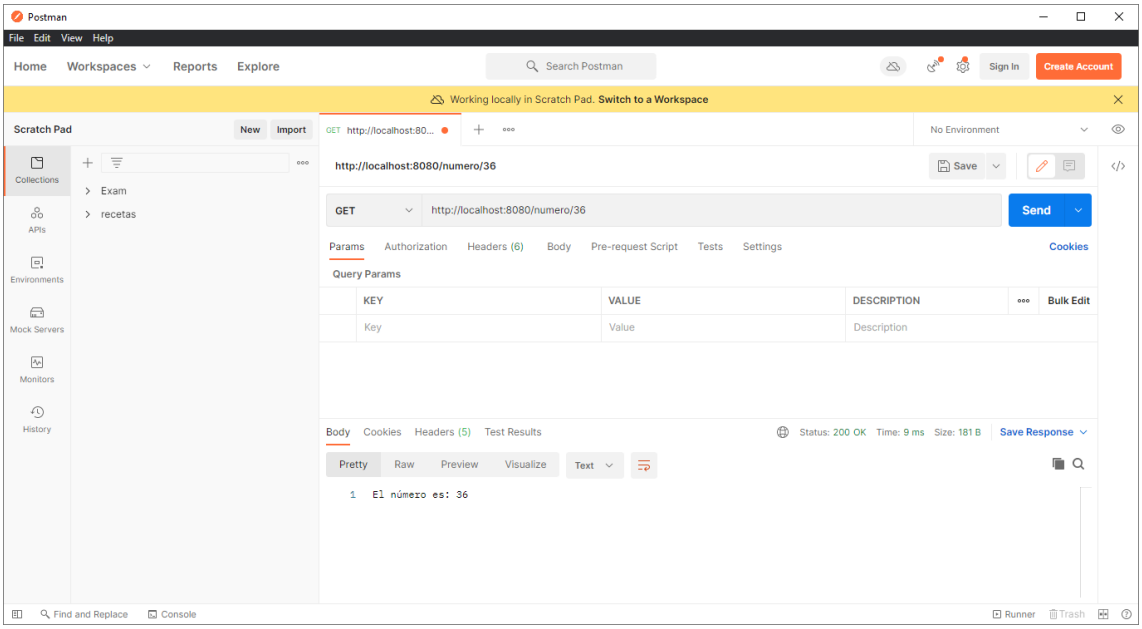
```
@GetMapping("/users/{id}")
public User getUser(@PathVariable String id) {
    // código para obtener el usuario por id
}
```

En este ejemplo, si accedes a “/users/123”, Spring llamará al método “getUser” con “123” como argumento.

“@PathVariable” es muy útil para crear rutas dinámicas donde algunos valores de la ruta pueden cambiar.

Ejemplo:

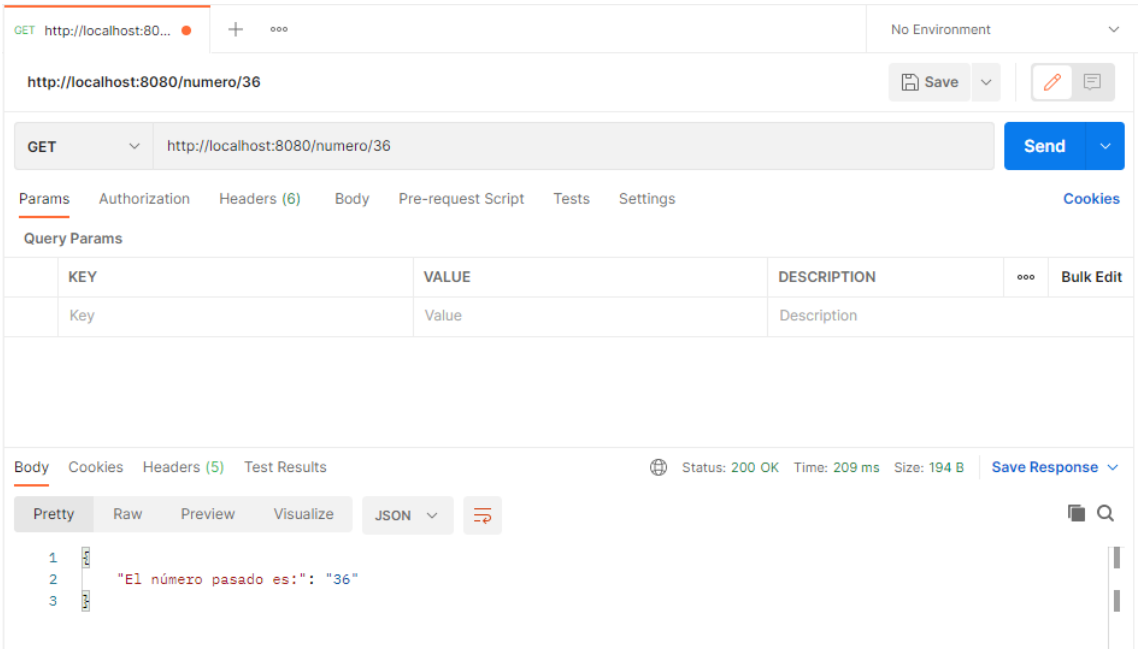
```
@GetMapping("/numero/{id}")
public String getUser(@PathVariable  String id) {
    return "El número es: " + id;
}
```



Hasta ahora hemos devuelto un String, si queremos devolver un json, que como sabes es el estándar, lo lógico es crearnos un modelo (clase), rellenarla y devolverla o más sencillo, crear un map (clave,valor) y devolverlo, veamos como quedaría:

```
@GetMapping("/numero/{id}")
public Map<String,String> getUser(@PathVariable  String id) {
    Map<String,String> map = Map.of("El número pasado es:", id);
```

```
    return map;
}
```



La anotación permite múltiples @PathVariable en una misma url:

```
#Multiples PathVariable
@GetMapping ("/api/mix/{product}/{id}")
public Map<String, Object> mixPathVar(@PathVariable String product,
@PathVariable Long id)
```

Ahora que has visto una introducción, realiza lo siguiente ejercicio, recuerda que la lógica de negocio debe estar en el paquete “service”:

Ejercicio 1

Generador de contraseñas: Implementa un servicio REST con dos endpoints que genere contraseñas aleatorias. El primer endpoint debe generar contraseñas de caracteres [a-z A-Z] y debe recibir como parámetro el número de caracteres de la contraseña a generar. El segundo endpoint debe generar contraseñas alfanuméricas. La longitud de la contraseña debe ser fija.

Método	EndPoint	Descripción	Status code	Respuesta
GET	/generaLetras	Genera contraseñas [a-z A-Z]	200	JSON
GET	/genera	Genera contraseñas alfanuméricas	200	JSON

Ejercicio 2

Servicio REST de conversión de unidades: Crea un servicio que reciba un valor y una unidad de medida, y devuelva la conversión a otra unidad de medida. Por ejemplo, convertir de grados Celsius a Fahrenheit o de kilómetros a millas. Debes pasar los datos a convertir al endpoint a través de un método get y como un valor pasado por variable (@PathVariable).

Método	EndPoint	Descripción	Status code	Respuesta
GET	/conDistancia	Convierte los kilómetros a millas	200	JSON
GET	/conTemp	Convierte de Celsius a Fahrenheit	200	JSON

Solución convertidor de distancias:

Para resolver el ejercicio deberíamos crear una clase en service, con la lógica de negocio:

```
package org.iesch.primerApi.service;

public class Conversor {
    public double convertirAKm(double millas){
        return millas * 1.60934;
    }
    public double convertirAMillas(double km){
        return km / 1.60934;
    }
}
```

Y un nuevo método en el controlador:

```
@GetMapping("/conDistancia/{distancia}")
public Map<String,String> getDistancia(@PathVariable  String
distancia) {
    Conversor conversor = new Conversor();
    double distance =
conversor.convertirAKm(Double.parseDouble(distancia));
    Map<String,String> map = Map.of("La distancia en km es:",
String.valueOf(distance));
    return map;
}
```

Si te fijas, debes crear una instacia de la clase Conversor y llamar al método que hace el cambio. (También podías haber hecho el método convertirAKm static)

Para solucionar este problema, aparece la inyección de dependencias.

Anotación @RequestParam:

@RequestParam

Se usa para extraer parámetros de consulta de la URL.

No forma parte de la ruta, sino que se añade después del símbolo ? en la URL.

/buscar?nombre=Juan

```
@GetMapping("/buscar")
public String buscarUsuario(@RequestParam("nombre") String nombre) {
    return "Nombre del usuario: " + nombre;
}
```

La anotación permite pasarle información como si es requerida o valores por defecto, veamos un ejemplo:

```
@RequestParam(required = false, defaultValue="Hola que tal") String
message)
```

Inyección de dependencias

La inyección de dependencias es un patrón de diseño que permite a una clase delegar la responsabilidad de crear sus dependencias a un contenedor o marco, en lugar de crearlas por sí misma. Esto puede hacer que el código sea más modular, más fácil de probar y más flexible.

En Spring, puedes usar la anotación “@Autowired” para inyectar automáticamente las dependencias en tus clases. Cuando Spring ve la anotación “@Autowired”, intentará encontrar un bean en su contenedor que coincida con la dependencia y lo inyectará en tu clase.

Veamos como quedaría el ejemplo anterior

La clase Conversor debe ser anotado como @Service de los esteriotipos de Spring:

```
import org.springframework.stereotype.Service;

@Service
public class Conversor {
```

Nuestro controlador debe tener una variable no instanciada “Conversor” anotada con “@Autowired”, la inyección de dependencias se encargará de crearla y destruirla según sea necesario, por lo que no debemos instanciarla nosotros:

```
@RestController
public class HolaMundo {
    @Autowired
    Conversor conversor;
    @GetMapping("/conDistancia/{distancia}")
    public Map<String,String> getDistancia(@PathVariable  String
distancia) {
        double distance =
conversor.convertirAKm(Double.parseDouble(distancia));
        Map<String,String> map = Map.of("La distancia en km es:",
String.valueOf(distance));
```

```
        return map;
    }
```

Realiza los dos ejercicios anteriores haciendo uso de inyección de dependencias si fuera necesario.

Enviando json a nuestros servicios REST.

Hasta ahora hemos pasado parámetros a través de la url, esto puede ser útil si la información a pasar es pequeña.

Ahora vamos a pasar json a nuestro servicio rest.

Lo primero a realizar es crear un modelo de los datos que vamos a recibir y después crear un método en nuestro controlador que recibirá dicho objeto.

El método del controlador, debe tener un parámetro anotado como: @RequestBody para indicar que recibirá un json. `(@RequestBody Producto nuevo)`

Ejemplo:

Crea un servicio rest al que le pases un json con el contenido:

```
{
    "nombre": "Juanma",
    "apellido": "Moreno"
}
```

Método	EndPoint	Descripción	Status code	Respuesta
POST	/guardaUser	Guarda el nombre y apellidos recibidos en un fichero de texto	200	JSON

Ejercicios:

Ejercicio 1

Creación de un servicio REST para una calculadora: Implementa endpoints para realizar operaciones matemáticas básicas como suma, resta, multiplicación y división. Los parámetros de entrada y la respuesta pueden ser enviados en formato JSON.

Método	EndPoint	Descripción	Status code	Respuesta
POST	/suma	Suma los valores pasados en un json	200	JSON
POST	/resta	Resta los valores pasados en un json	200	JSON
POST	/multiplicacion	Multiplica los valores pasados en un json	200	JSON
POST	/división	Divide los valores pasados en un json	200	JSON

Ejercicio 2

Generador de contraseñas: Implementa un servicio REST con dos endpoints que genere contraseñas aleatorias. El primer endpoint debe generar contraseñas de caracteres [a-z A-Z] y debe recibir como parámetro el número de caracteres de la

contraseña a generar. El segundo endpoint debe generar contraseñas alfanuméricas. Debe recibir como parámetro el número de caracteres de la contraseña a generar

Método	EndPoint	Descripción	Status code	Respuesta
POST	/generaLetras	Genera contraseñas [a-z A-Z]	200	JSON
POST	/genera	Genera contraseñas alfanuméricas	200	JSON

Modificando el Path completo de nuestra API

Si colocamos la anotación `@RequestMapping` a nivel de clase podemos conseguir aplicar una ruta base a todos los endpoints de un controlador:

```
@RequestMapping("/api")
```

Esto establece /api como la ruta base, por lo que los endpoints estarán accesibles en /api/hola y /api/adios.

Esto es útil si queremos:

1. Agrupar Funcionalidades Relacionadas: Si tienes un conjunto de endpoints que pertenecen a una misma funcionalidad o módulo, puedes agruparlos bajo una ruta base común. Esto mejora la organización y la claridad del código.
2. Versionado de API: Puedes versionar tu API de manera efectiva utilizando rutas base. Esto facilita la administración de múltiples versiones de tu API.
3. Versionado de API: Puedes versionar tu API de manera efectiva utilizando rutas base. Esto facilita la administración de múltiples versiones de tu API.
4. Microservicios y Servicios REST: Para servicios RESTful y microservicios, las rutas base ayudan a mantener la consistencia y estructura en tu API, especialmente cuando tienes múltiples recursos.
5. Módulos Específicos: Cuando tienes módulos específicos en tu aplicación, puedes usar rutas base para organizarlos de manera lógica.

Propiedades personalizadas dentro del fichero Properties

Podemos definir tus propiedades personalizadas en el archivo application.properties:

```
mi.app.nombre=MiAplicacion
mi.app.version=1.0.0
mi.app.autor=Juan Perez
```

Después podemos inyectar propiedades en tu código usando @Value

Ejemplo:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class MiComponente {
```

```
    @Value("${mi.app.nombre}")
    private String nombre;

    @Value("${mi.app.version}")
    private String version;

    @Value("${mi.app.autor}")
    private String autor;

    public void mostrarInfo() {
        System.out.println("Nombre: " + nombre);
        System.out.println("Versión: " + version);
        System.out.println("Autor: " + autor);
    }
}
```

También podemos tener listas de valores:

```
config.listOfValues=Hola,que,tal
```

```
    @Value("$config.listOfValues")
    private String[] listOfValues;
```

También podemos crearnos nuestros propios ficheros de propiedades, por ejemplo app.properties

Para poder leerlo, debemos indicarle a la clase principal que lo busque en el classpath con su correspondiente anotación:

```
@SpringBootApplication
@PropertySource("classpath:values.properties")
public class RestdemoApplication {
```

UTF8 en el archivo de propiedades:

```
@PropertySource("classpath:values.properties", encoding = "UTF-8")
```

Otra forma de leer archivos de propiedades:

También podemos leer archivos de configuración con la clase Enviroment

```
@AutoWired
private Enviroment env;

env.getProperty("config.XXXX")
```

Modificando los códigos de estado.

Spring nos ofrece una clase: ResponseEntity<T> , que nos permite manejar de una forma más conveniente la respuesta que enviamos al cliente.

- Hereda de HttpEntity<T>
- Nos permite indicar el código de respuesta, qué se envía en el cuerpo, responder peticiones sin el mismo

@GetMapping

- Devolvemos 200 OK si localizamos el recurso
- Si no, devolvemos 404 Not Found

@PostMapping

- Devolvemos 201 Created

@PutMapping

- Devolvemos 200 OK si localizamos y modificamos el recurso
- Si no, devolvemos 404 Not Found.

@DeleteMapping

- Devolvemos 204 No Content.

Ejemplo:

```
@GetMapping("/producto")
    public ResponseEntity<?> obtenerTodos() {
```

……(Falta código)

```
      if (result.isEmpty()) {
            return ResponseEntity.notFound().build();
        } else {
            return ResponseEntity.ok(result);
        }
}
```

……(Falta código)

```
@PostMapping("/producto")
public ResponseEntity<?> nuevoProducto(@RequestBody Producto nuevo) {
……
return ResponseEntity.status(HttpStatus.CREATED).body(saved);

@PutMapping("/producto/{id}")
return ResponseEntity.ok(p);
return ResponseEntity.notFound().build();

@DeleteMapping("/producto/{id}")
return ResponseEntity.noContent().build();
```

Aplicación con un main en spring

Clase principal debe implementar la interfaz CommandLineRunner

```
@SpringBootApplication
public class RestdemoApplication implements CommandLineRunner {
```

Log en Aplicaciones Spring

Podemos usar la fachada SLF4J integrada en Spring.

Debemos configurar “Application.properties”

```
logging.level.root=INFO
logging.level.org.springframework.web=DEBUG
logging.file.name=app.log
```

El primer `logging.level.root` hace referencia a tu app

El segundo `logging.level.org.springframework.web` hece referencia al core de Spring.

El tercero, `logging.file.name` es el fichero donde se guardarán los logs.

Recuerda que para poder hacer uso de dicha librería, debemos indicarlo en la clase en cuestión un ejemplo:

```
private static final Logger logger =
LoggerFactory.getLogger(HolaMundo.class);
```

Aplicación UI con Spring.

Spring está pensado para la construcción del lado de backend. Por lo tanto está pensado para realizar aplicaciones que corren en servidor.

Si queremos realizar una aplicación con interfaz de usuario deberíamos indicárselo, en la clase principal de Spring `@SpringBootApplication` debemos indicarle en el main:

```
System.setProperty("java.awt.headless", "false");
```

Y en el método run podemos crear nuestra app:

```
if (!GraphicsEnvironment.isHeadless()) {
      SwingUtilities.invokeLater(() -> MySwingApp.createAndShowGUI());
} else {
      System.out.println("Modo headless detectado, la GUI no se
lanzará.");
       }
```

Contenido de la clase MySwingApp:

```
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MySwingApp extends JFrame {

    public MySwingApp() {
```

```
        // Configurar la ventana
        setTitle("Mi Aplicación Swing");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        // Añadir un componente
        JLabel label = new JLabel("¡Hola, Swing en Spring Boot!");
        add(label);
    }

    public static void createAndShowGUI() {
        // Crear y mostrar la GUI
        MySwingApp frame = new MySwingApp();
        frame.setVisible(true);
    }
}
```

Anotación @Configuration

Esta anotación le indica a Spring que la clase anotada contiene configuraciones que deben ser tratadas como fuentes de beans (objetos gestionados por el contenedor de Spring).

¿Qué hace?

Cuando añades @Configuration a una clase, le estás diciendo a Spring que utilice esa clase para definir uno o más beans que serán gestionados en el contexto de la aplicación.

¿Para qué sirve?

Definir Beans: usando métodos anotados con @Bean.

Configuración centralizada: Centraliza la configuración de beans en una sola clase, haciendo que la configuración sea más organizada y manejable.

Flexibilidad: Puedes configurar dependencias y comportamientos de los beans de manera programática.

¿Cómo se comporta?

Escaneo y Registro: Spring escanea las clases anotadas con @Configuration al arrancar la aplicación y registra los beans definidos en ellas.

Creación de Beans: Los métodos con @Bean dentro de la clase @Configuration se invocan para crear e inicializar los beans.

Concepto de DTO (Data Transfer Object)

Un DTO es un objeto especial que usaremos para mover información de un lado a otro en nuestra aplicación.

¿Para qué sirve?

Transporte: Lleva datos entre diferentes partes de la aplicación, como entre el frontend y el backend, o entre servicios.

Organización: Agrupa datos relacionados para que viajar juntos tenga sentido.

Seguridad: Puede ayudar a asegurar que solo los datos necesarios viajen, protegiendo información sensible.

Ejemplo:

Imagina que tenemos una aplicación con un modelo “Usuario” que tiene millones de atributos, pero nuestra aplicación solo necesita hacer un login, es decir solo necesita los atributos nombre de usuario y password, ¿Tiene sentido mover el millón de atributos para hacer un login? Por esta razón aparecen los DTO, no son más que pojos simples con los datos simples que necesitamos en cada momento.

Excepciones.

Cuando desarrollas una API con Spring y ocurre un error, Spring devuelve una respuesta con un formato de error predeterminado. Este formato de error incluye detalles como el estado HTTP, el error, el mensaje y la ruta de la solicitud. veamos un ejemplo de cómo se ve:

```
{
    "timestamp": "2021-12-01T12:00:00.000+00:00",
    "status": 404,
    "error": "Not Found",
    "message": "No se pudo encontrar el recurso solicitado",
    "path": "/api/resource"
}
```

- **timestamp**: La fecha y hora en que ocurrió el error.
- **status**: El código de estado HTTP.
- **error**: Una descripción corta del código de estado.
- **message**: Un mensaje más detallado sobre el error.
- **path**: La ruta de la solicitud que causó el error.

Este formato de error es útil porque proporciona información detallada sobre el error, lo que puede ayudar a los clientes de la API a entender qué salió mal.

Spring Boot nos recomienda una serie de herramientas que podemos usar, en vez de los simples bloques “try-catch”. Primero empezaremos destacando dos anotaciones que nos permiten manejar la gestión de excepciones:

@ControllerAdvise / @RestControllerAdvise → Al añadir esta anotación a una clase controlador se inyecta código transversal en métodos existentes. También nos permite interceptar y modificar los valores que vamos a retornar de un método del controlador principal.

@ExceptionHandler → Esta anotación es añadida en los métodos que se encuentran dentro de la clase donde realizaremos las excepciones. Al añadirla tenemos que darle como parámetro la excepción o lista de excepciones las cuales, al producirse la excepción, ejecutará el método. Además, el método soporta el retorno de diferentes tipos como ResponseEntity, String o void.

Mirar ejemplo realizado en clase.

¿Qué es CORS?

CORS significa **Cross-Origin Resource Sharing (Intercambio de Recursos de Origen Cruzado)**. Es un mecanismo de seguridad implementado en los navegadores web que controla cómo las aplicaciones web pueden hacer solicitudes HTTP a dominios diferentes del que sirvió la página web original.

CORS es esencial cuando:

- Desarrollas un frontend y backend por separado
- Tu SPA (Single Page Application) consume una API REST

El Problema que Resuelve

Por defecto, los navegadores implementan **la política de mismo origen** (Same-Origin Policy), que impide que JavaScript en una página web haga solicitudes a un dominio diferente. Por ejemplo:

- Tu frontend está en http://localhost:3000

- Tu backend (API REST) está en http://localhost:8080

Sin CORS, el navegador bloqueará las solicitudes del frontend al backend porque son "orígenes" diferentes.

¿Qué se considera un Origen Diferente?

Dos URLs tienen orígenes diferentes si difieren en:

1. Protocolo (“http” vs “https”)

2. Dominio (“example.com” vs “api.example.com”)

3. Puerto (“localhost:3000” vs “localhost:8080”)

Cómo Funciona CORS

CORS funciona mediante **cabeceras HTTP** que el servidor envía al navegador para indicar qué orígenes están permitidos:

- Access-Control-Allow-Origin: http://localhost:3000
- Access-Control-Allow-Methods: GET, POST, PUT, DELETE
- Access-Control-Allow-Headers: Content-Type, Authorization

Ejemplo en Spring Boot

Configuración global:

```
@Configuration
public class CorsConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOrigins("http://localhost:3000")
                .allowedMethods("GET", "POST", "PUT", "DELETE")
                .allowedHeaders("*")
                .allowCredentials(true);
    }
}
```

O directamente en un controlador:

```
@RestController
@CrossOrigin(origins = "http://localhost:3000")
public class EstudianteController {
    // ... tus métodos
}
```

Nota: También se puede aplicar la anotación `@CrossOrigin` a los diferentes métodos, no solo a nivel de clase

Tipos de Solicitudes CORS

1.- Solicitudes Simples: GET, POST con tipos de contenido simples
2.- Solicitudes Preflight: El navegador envía primero una solicitud OPTIONS para verificar permisos antes de enviar la solicitud real (común con PUT, DELETE, o headers personalizados)