Sistemas Distribuidos

Bloque II

Desarrollo de sistemas distribuidos y aplicaciones web.

Tema 2.3:

API REST

URJC Escuela Técnica Superior de Ingeniería Informática

Nicolás Rodríguez
nicolas.rodriguez@urjc.es

Índice

- **API REST**
- Recursos
- URIs
- Métodos HTTP
- Códigos de estado
- JSON
- Postman
- Componentes

API (Interfaz de Programación de Aplicaciones)

- Una API es un conjunto de reglas y definiciones que permite que diferentes sistemas de software se comuniquen entre sí.
- Funciona como un intermediario que permite a dos aplicaciones interactuar o intercambiar datos.
- Las APIs definen la manera en que los desarrolladores deben realizar solicitudes y estructurar las respuestas para que las aplicaciones puedan comunicarse entre sí. Esto incluye especificaciones de rutas (endpoints), métodos de solicitud (como GET o POST), formatos de datos, etc.

REST (Transferencia de Estado Representacional)

- Estilo arquitectónico:
  - REST es un estilo arquitectónico para el diseño de servicios de red. Fue descrito por Roy Fielding en su tesis doctoral en el año 2000.
- Fundamentos de REST:
  - Se basa en un conjunto de principios y restricciones que, cuando se aplican correctamente, llevan a un sistema distribuido eficiente, escalable y flexible.
- Uso de recursos:
  - En REST, la interacción se realiza a través de recursos, que son cualquier tipo de objeto, dato o servicio que puede ser nombrado. Cada recurso es identificable por una URI (Identificador Uniforme de Recursos).
- Representación del estado:
  - Lo que se transfiere entre el cliente y el servidor no es el recurso en sí, sino una representación del estado del recurso, como HTML, XML, JSON, etc.

API REST

- Combinación de API y REST:
  - Una API REST es una API que sigue los principios del estilo arquitectónico REST. Ofrece una forma estandarizada de integrar sistemas distribuidos, aprovechando los métodos y protocolos de la web, principalmente HTTP.
- Operaciones a través de Métodos HTTP:
  - Utiliza métodos GET, POST, PUT, DELETE, etc., para realizar operaciones en los recursos.
- Independencia del formato de datos:
  - Aunque el formato más común es JSON, una API REST puede usar cualquier formato de representación de datos que sea entendible tanto por el cliente como por el servidor.
- Sin Estado y Sin sesión:
  - REST es sin estado, lo que significa que cada solicitud de un cliente debe contener toda la información necesaria para procesarla, sin depender de ninguna información almacenada en el servidor respecto a sesiones anteriores.

Índice

- API REST
- **Recursos**
- URIs
- Métodos HTTP
- Códigos de estado
- JSON
- Postman
- Componentes

Recursos

- Un recurso en REST es cualquier elemento o entidad que puede ser nombrado, identificado, y sobre el cual se pueden realizar operaciones. Puede ser un objeto físico, un documento, una imagen, un servicio, etc.
- Características:
  - Representación: Un recurso puede tener varias representaciones, como JSON, XML, HTML, etc.
  - Estado: Los recursos tienen un estado que puede cambiar con el tiempo.
  - Identificación Única: Cada recurso es único y accesible a través de su URI.
- Ejemplos:
  - Un artículo en un blog, un usuario en un sistema, un producto en un catálogo.

Índice

- API REST
- Recursos
- **URIs**
- Métodos HTTP
- Códigos de estado
- JSON
- Postman
- Componentes

URIs

- Las URIs son la forma en que se identifican y localizan los recursos en una API REST. Son análogas a las direcciones web que utilizamos para acceder a páginas específicas en Internet.
- Características:
  - Uniformidad: Siguen un formato estándar que permite localizar recursos en la red.
  - Inmutabilidad: Idealmente, una URI no debe cambiar con el tiempo. Si un recurso cambia, su URI debe permanecer constante.
  - Legibilidad: Aunque no es un requisito, es una buena práctica diseñar URIs que sean fáciles de entender y predecir por humanos.

URIs

- Claridad y Simplicidad:
  - Las URIs deben ser intuitivas y fáciles de comprender. Por ejemplo, usar /usuarios para acceder a recursos de usuarios.
- Uso de nombres en plural:
  - Es común usar nombres en plural para colecciones de recursos (p. ej., /usuarios en lugar de /usuario).
- Estructura Jerárquica:
  - Las URIs pueden representar relaciones jerárquicas. Por ejemplo, /usuarios/123/posts podría representar todos los posts del usuario con ID 123.
- Evitar uso de verbos:
  - Las URIs deben centrarse en identificar recursos, no acciones. Las acciones se definen mediante los métodos HTTP (GET, POST, PUT, DELETE).

URIs

- Una URI típica para un recurso REST podría ser algo así como https://www.ejemplo.com/recursos/123:
  - https://www.ejemplo.com/ es el dominio base,
  - /recursos/ es el path que indica el tipo de recurso,
  - y 123 es un identificador único para un recurso específico.
- Los recursos representan entidades sobre las que se pueden realizar operaciones, y las URIs proporcionan una forma estandarizada y coherente de identificar y acceder a estos recursos.

Índice

Métodos HTTP

1. GET

- Uso: Solicitar datos de un recurso específico o un conjunto de recursos.
- Características:
  - Es un método seguro, lo que significa que no modifica el estado del recurso.
  - Es idempotente, lo que implica que realizar la misma solicitud GET varias veces produce el mismo resultado.
- Ejemplo de Uso:
  - Obtener la información de un usuario (GET /usuarios/123) o listar todos los usuarios (GET /usuarios).

Métodos HTTP

2. POST

- Uso: Enviar datos para crear un nuevo recurso.
- Características:
  - No es ni seguro ni idempotente. Cada solicitud puede resultar en la creación de un nuevo recurso o en un cambio en el servidor.
- Ejemplo de Uso:
  - Crear un nuevo usuario con datos en el cuerpo de la solicitud (POST /usuarios).

Métodos HTTP

3. PUT

- Uso: Actualizar un recurso existente o crear un recurso en una URI específica si no existe.
- Características:
  - Es idempotente. Realizar la misma solicitud PUT varias veces tiene el mismo efecto que hacerlo una sola vez.
- Ejemplo de Uso:
  - Actualizar la información de un usuario (PUT /usuarios/123).

Métodos HTTP

4. DELETE

- Uso: Eliminar un recurso específico.
- Características:
  - Es idempotente. Una vez que el recurso se elimina, realizar la misma solicitud no tiene ningún efecto adicional.
- Ejemplo de Uso:
  - Eliminar un usuario (DELETE /usuarios/123).

Métodos HTTP

5. PATCH

- Uso: Aplicar actualizaciones parciales a un recurso.
- Características:
  - Puede ser idempotente o no, dependiendo de cómo se implemente.
- Ejemplo de Uso:
  - Actualizar ciertos campos de un usuario sin modificar todo el recurso (PATCH /usuarios/123).

Índice

- API REST
- Recursos
- URIs
- Métodos HTTP
- **Códigos de estado**
- JSON
- Postman
- Componentes

Códigos de estado

- En las APIs REST, las respuestas y los códigos de estado HTTP son esenciales para comunicar el resultado de las solicitudes de los clientes.
- Estos códigos informan al cliente sobre el éxito, el fallo o la necesidad de tomar una acción adicional.
- 1xx - Respuestas Informativas:
  - Indican que la solicitud fue recibida y está siendo procesada.
  - Ejemplo: 100 Continue

Códigos de estado

- 2xx - Éxito:
  - Significan que la solicitud fue recibida, entendida y aceptada correctamente.
  - 200 OK: Respuesta estándar para solicitudes exitosas.
  - 201 Created: Indica que un recurso fue creado exitosamente (generalmente en respuesta a solicitudes POST o PUT).
  - 204 No Content: La solicitud fue procesada con éxito, pero no hay contenido para devolver.
- 3xx - Redirecciones:
  - Indican que se deben tomar acciones adicionales para completar la solicitud.
  - Ejemplo: 301 Moved Permanently

Códigos de estado

- 4xx - Errores del cliente:
  - Se usan cuando la solicitud no pudo ser procesada debido a errores del lado del cliente.
  - 400 Bad Request: La solicitud no se pudo entender o procesar.
  - 401 Unauthorized: Autenticación requerida o fallida.
  - 403 Forbidden: El servidor entiende la solicitud, pero se niega a autorizarla.
  - 404 Not Found: El recurso solicitado no fue encontrado.
  - 405 Method Not Allowed: El método HTTP utilizado no está permitido para el recurso.
- 5xx - Errores del servidor:
  - Indican fallos del lado del servidor.
  - 500 Internal Server Error: Error genérico cuando el servidor falla al procesar la solicitud.
  - 503 Service Unavailable: El servidor no está disponible, generalmente por mantenimiento o sobrecarga.

Índice

- API REST
- Recursos
- URIs
- Métodos HTTP
- Códigos de estado
- **JSON**
- Postman
- Componentes

JSON – Características principales

- JSON (JavaScript Object Notation) es un formato de representación de datos ampliamente utilizado en el desarrollo de APIs, especialmente en APIs REST.
- Su popularidad se debe a su simplicidad, eficiencia y facilidad de uso en aplicaciones web y móviles
- Basado en Texto:
  - JSON es un formato de datos basado en texto, lo que lo hace fácilmente legible por humanos y máquinas.

JSON – Características principales

- Estructura de Datos Ligera:
  - Ofrece una forma concisa y fácil de representar estructuras de datos como objetos y arrays.
  - Ejemplo de un objeto JSON: { "nombre": "Ana", "edad": 25 }
- Independiente del Lenguaje:
  - A pesar de estar basado en la sintaxis de JavaScript, es independiente de cualquier lenguaje de programación.
  - La mayoría de los lenguajes modernos soportan JSON a través de bibliotecas o funciones nativas.
- Fácil de Analizar:
  - La simplicidad de su estructura lo hace fácil de analizar y generar, lo que reduce la complejidad del código en el cliente y el servidor.

JSON – Uso

- Intercambio de Datos:
  - JSON es el formato más común para el envío y recepción de datos en APIs REST.
  - Ejemplo: Un cliente envía una solicitud POST con un cuerpo JSON para crear un nuevo recurso.
- Representación de Recursos:
  - Permite representar los recursos de una API de manera estructurada y clara.
  - Ejemplo: Un recurso de usuario se puede representar como un objeto JSON con propiedades como nombre, correo electrónico, etc.
- Negociación de Contenido:
  - Las APIs REST pueden usar JSON como uno de los formatos en la negociación de contenido, respondiendo con datos en formato JSON si el cliente lo solicita.

JSON – Estructura y sintaxis

- Objetos:
  - Representados por llaves { }, contienen un conjunto de pares de clave-valor.
  - Ejemplo: { "nombre": "Carlos", "edad": 40 }
- Arrays:
  - Representados por corchetes [ ], son colecciones ordenadas de valores.
  - Ejemplo: ["manzanas", "naranjas", "peras"]
- Valores:
  - Pueden ser cadenas de texto (strings), números, booleanos (true/false), arrays o incluso otros objetos.
  - Ejemplo: { "nombres": ["Ana", "Luis"], "activo": true }

JSON – Estructura y sintaxis

- Cadenas de Texto (Strings):
  - Deben estar entre comillas dobles.
  - Ejemplo: "nombre": "Juan"
- Números:
  - Pueden ser enteros o flotantes.
  - Ejemplo: "edad": 30
- Limitaciones:
  - No tiene soporte nativo para comentarios, lo que puede ser un inconveniente para la documentación en el código.
  - Limitado en términos de tipos de datos (por ejemplo, no hay distinción entre tipos de números).

JSON – Ejemplo

```
{
  "id": "12345",
  "nombre": "Nicolás Rodríguez",
  "edad": 33,
  "email": "nicolas.rodriguez@urjc.es",
  “enActivo": true,
  "roles": ["usuario", "profesor"],
  "direccion": {
    "calle": "Calle Falsa 123",
    "ciudad": "Alcorcón",
    "codigoPostal": "28922"
  }
}
```

- Strings: id, nombre, email
- Número: edad
- Boolean : enActivo
- Lista: roles
- Objeto: dirección

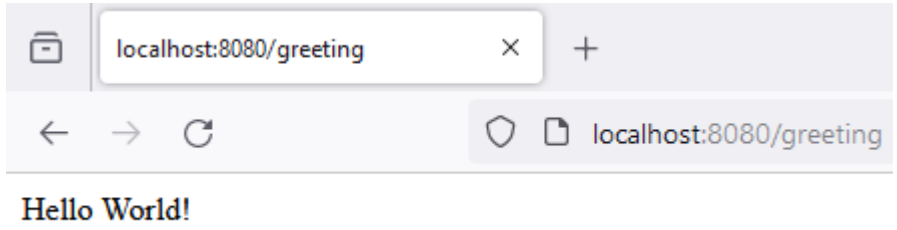
Ejemplo

- Aplicación en SpringBoot con la dependencia de SpringWeb para saludar desde una API REST

GreetingRESTController

```
@RestController
public class GreetingRESTController {
    @GetMapping("/greeting")
    public String greeting() {
        return "Hello World!";
    }
}
```

Salida:



Índice

- API REST
- Recursos
- URIs
- Métodos HTTP
- Códigos de estado
- JSON
- **Postman**
- Componentes

Postman

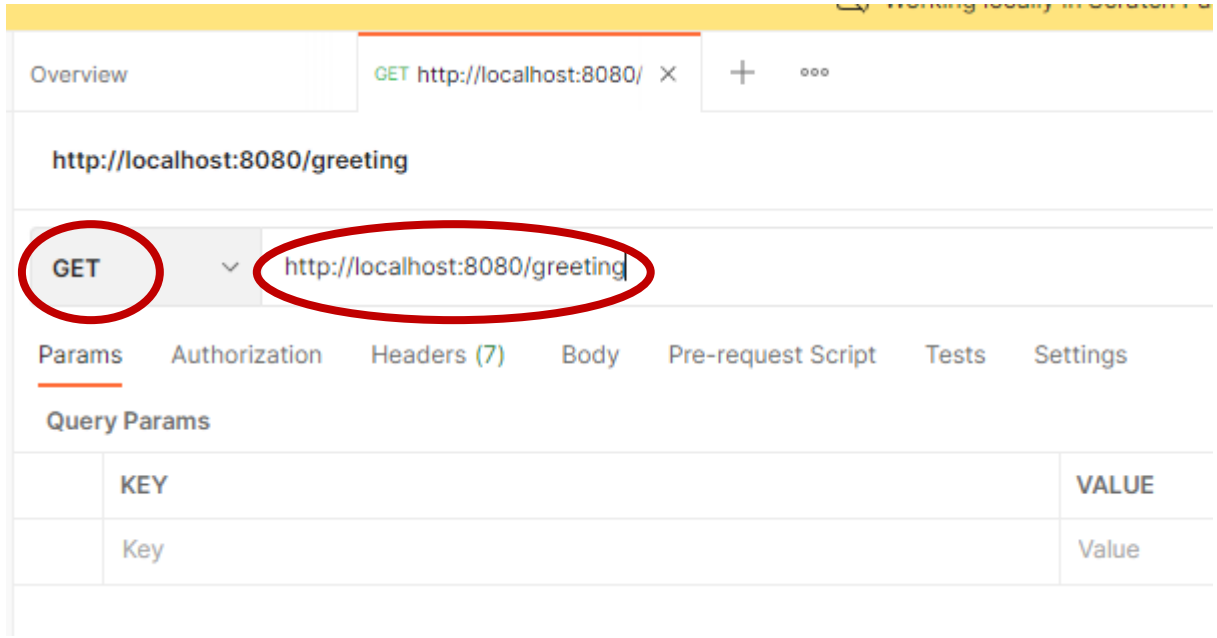
- Postman es una plataforma popular para el desarrollo y pruebas de APIs.
- Ofrece un conjunto de herramientas para construir, probar y documentar APIs, facilitando el trabajo de desarrolladores en el diseño de APIs, la verificación de respuestas y la colaboración en equipo.
- Construcción y Prueba de Solicitudes:
  - Permite a los usuarios crear solicitudes HTTP a APIs, incluyendo todos los métodos (GET, POST, PUT, DELETE, etc.).
  - Soporta la personalización de cabeceras, parámetros, cuerpos de solicitud y autenticación.

Postman

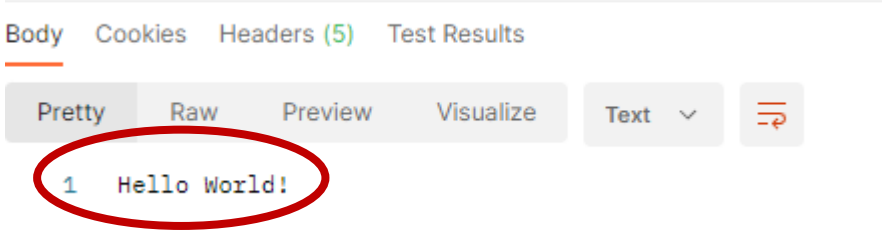
- Colecciones y Ambientes:
  - Las colecciones permiten organizar y guardar solicitudes, lo que es útil para agrupar y compartir sets de llamadas a APIs.
  - Los ambientes son conjuntos de variables que permiten cambiar fácilmente entre diferentes configuraciones, como desarrollo, pruebas y producción.
- Documentación:
  - Ofrece la capacidad de generar y publicar documentación de APIs a partir de las colecciones, facilitando la comunicación y colaboración entre equipos.
- https://www.postman.com/

Postman

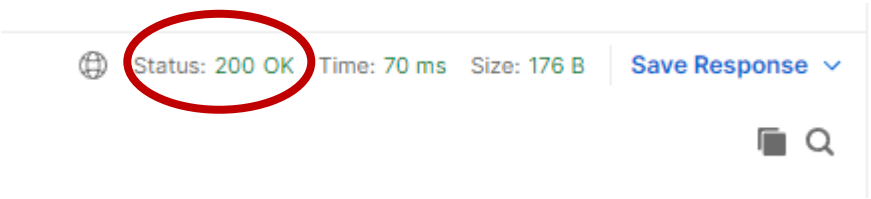
Petición vía Postman



Respuesta vía Postman



Respuesta HTTP Postman



Postman

Petición vía Postman

https://swapi.dev/api/people/1

- A la izquierda se puede observar la respuesta obtenida de realizar una petición GET a la URL arriba indicada.
- A la hora de obtener una respuesta, se puede elegir el formato de respuesta. En este caso, hemos elegido JSON.

Ejemplo

- Aplicación en SpringBoot con la dependencia de SpringWeb para añadir un Usuario a través de la API REST
- Se utiliza un mapa para almenar un usuario y su id.
- Este id es autoincremental.

```
public class Usuario {
    private String nombre;
    private int edad;

    // Constructor, getters y setters para 'nombre' y 'edad'
}
```

Ejemplo

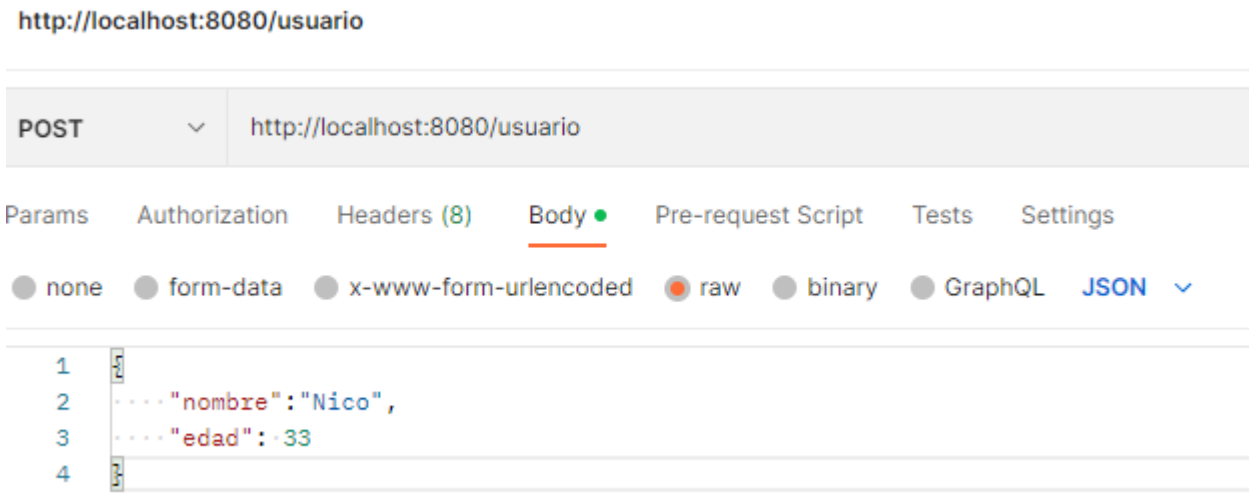
UsuarioRESTController.java

```
@PostMapping("/usuario")
public ResponseEntity<String> agregarUsuario(@RequestBody Usuario usuario) {
    int userId = idCounter.incrementAndGet();
    usuarios.put(userId, usuario);
    return ResponseEntity.status(201).body("Usuario agregado con ID: " + userId);
}
```

- Se utiliza el PostMapping para crear un Usuario. Por tanto, la petición es de tipo POST
- @RequestBody recibe una instancia de la clase Usuario, y la genera automáticamente.
- Se devuelve un ResponseEntity, que permite añadir código de respuesta y cuerpo de respuesta.

Ejemplo

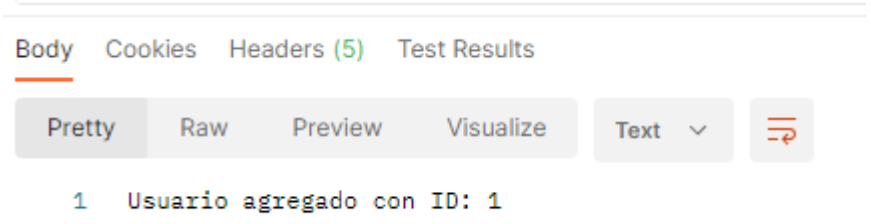
Petición vía Postman



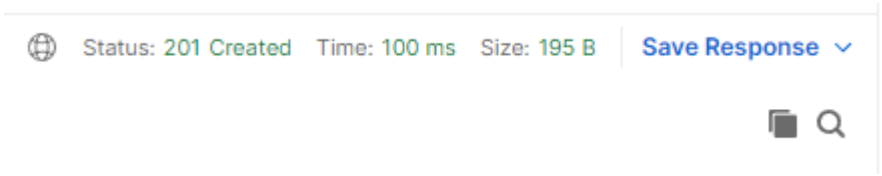
- La petición ahora es de tipo POST.
- Añadimos la URL del endpoint
- Tenemos que enviar la “instancia” de la clase Usuario
- Body/raw/JSON y escribimos la instancia

Ejemplo

Respuesta vía Postman



Respuesta vía Postman



Ejercicio

- Implementar todas las operaciones CRUD + PATCH sobre la entidad Cerveza con una API REST
- Generar una colección de Postman para todas los endpoint

Ejercicio

- Crear una nueva cerveza
  - Endpoint: /cervezas
  - Método HTTP: POST
  - Descripción: Este endpoint acepta un objeto Cerveza en el cuerpo de la solicitud y lo agrega al mapa de cervezas. Genera un ID único para cada nueva cerveza.
- Obtener una cerveza por ID
  - Endpoint: /cervezas/{id}
  - Método HTTP: GET
  - Descripción: Retorna los detalles de la cerveza con el ID especificado. Si no existe una cerveza con ese ID, devuelve un estado 404 (Not Found).

Ejercicio

- Actualizar una cerveza existente
  - Endpoint: /cervezas/{id}
  - Método HTTP: PUT
  - Descripción: Reemplaza completamente la cerveza con el ID especificado con la nueva información proporcionada en el cuerpo de la solicitud. Si la cerveza no existe, devuelve un estado 404 (Not Found).
- Eliminar una cerveza
  - Endpoint: /cervezas/{id}
  - Método HTTP: DELETE
  - Descripción: Elimina la cerveza con el ID especificado. Si no existe una cerveza con ese ID, devuelve un estado 404 (Not Found).

Ejercicio

- Actualizar parcialmente una cerveza
  - Endpoint: /cervezas/{id}
  - Método HTTP: PATCH
  - Descripción: Permite actualizar parcialmente los atributos de una cerveza existente (como el nombre, tipo o porcentaje de alcohol). Solo los campos proporcionados en el cuerpo de la solicitud serán actualizados.
- Obtener todas las cervezas:
  - URL: /cervezas
  - Método HTTP: GET
  - Descripción: Este endpoint maneja solicitudes GET para listar todas las cervezas almacenadas en la API. Devuelve una lista de objetos Cerveza con sus detalles.

Ejercicio

Elementos a insertar

```
[
    {
      "nombre": "Alhambra 1925",
      "tipo": "Lager",
      "alcohol": 6.4
    },
    {
      "nombre": "Mahou",
      "tipo": "Lager",
      "alcohol": 5.5
    },
    {
      "nombre": "Estrella Galicia",
      "tipo": "Lager",
      "alcohol": 5.5
    }
]
```

Listado de elementos

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON


```
[
    {
        "id": 1,
        "nombre": "Alhambra 1925",
        "tipo": "Lager",
        "alcohol": 6.4
    },
    {
        "id": 2,
        "nombre": "Mahou",
        "tipo": "Lager",
        "alcohol": 5.5
    },
    {
        "id": 3,
        "nombre": "Estrella Galicia",
        "tipo": "Lager",
        "alcohol": 5.5
    }
]
```

Índice

- API REST
- Recursos
- URIs
- Métodos HTTP
- Códigos de estado
- JSON
- Postman
- **Componentes**

Componentes

- Las anotaciones @Service y @Autowired son componentes clave en Spring Framework, y se utilizan para implementar la inyección de dependencias y la separación de la lógica de negocio, lo que facilita la creación de aplicaciones escalables y mantenibles.
- @Service es ideal para clases que implementan la lógica de negocios.
- @Autowired es esencial para inyectar dependencias de manera limpia y desacoplada.

Componentes

- @Service es una anotación en Spring que se utiliza para marcar una clase en Java como un servicio. Es una especialización de la anotación @Component, que indica que una clase es un "bean" de Spring y que debe ser gestionada por el contenedor de Spring.
- Se coloca sobre las clases que implementan la lógica de negocios. En la arquitectura MVC (Modelo-Vista-Controlador), @Service se utiliza en la capa de servicio, que se sitúa entre los controladores y las capas de acceso a datos (repositorios).

Componentes

- @Autowired es una anotación que permite a Spring resolver e inyectar beans colaboradores en tu clase automáticamente. Es decir, Spring buscará y proporcionará las dependencias requeridas para un objeto sin necesidad de configuración manual.
- Se puede usar en constructores, métodos setter y campos para inyectar dependencias. Spring se encargará de buscar en su contenedor el bean adecuado que coincida con la propiedad o parámetro que necesita ser inyectado.

Ejercicio

- Unir el controlador web y el controlador de la API REST de Cerveza, de manera en que si añado una cerveza desde la API REST sea capaz de verla en la web y viceversa.
- Para ello es necesario:
  - Implementar un @Servicio con el mapa.
  - Añadir el @Autowired a ambos controladores.
  - Modificar los métodos dentro de los controladores para que usen el @Autowired.
  - Evitar conflictos de endpoints.
  - Separar las clases en:
    - Entidades.
    - Controladores.
    - Servicios.

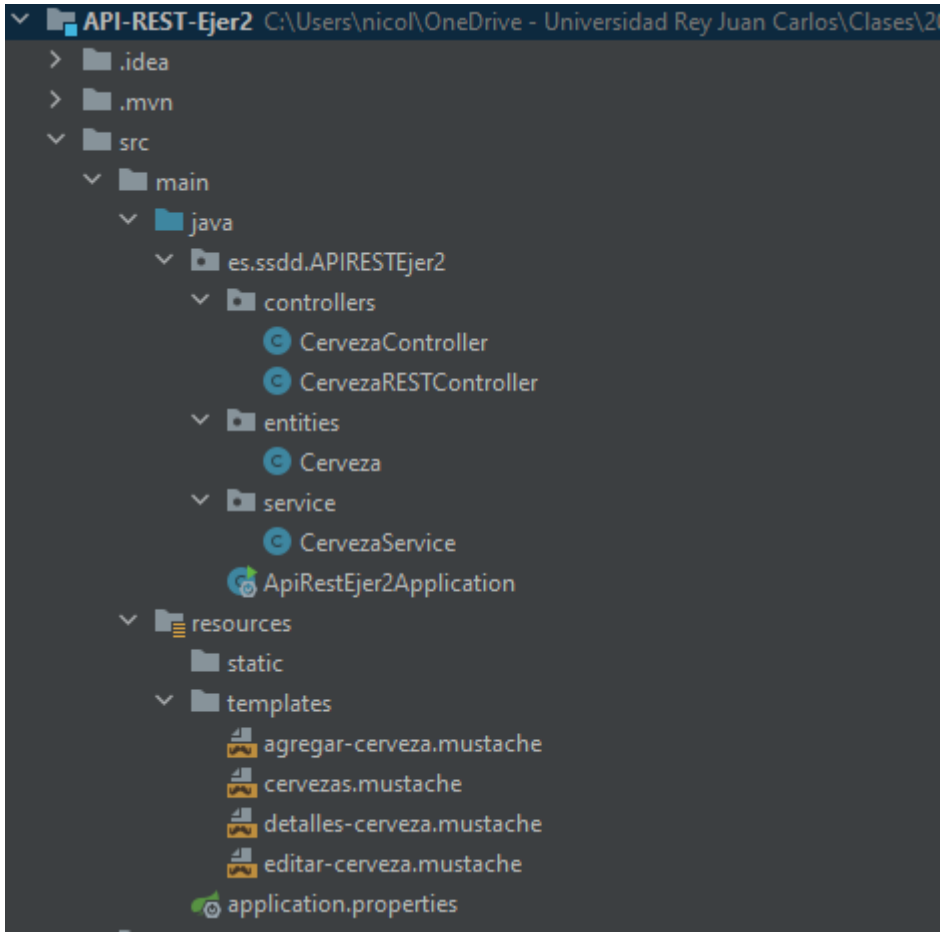
Ejercicio

ServiceCerveza.java

```java
@Service
    public class CervezaService {
    private final Map<Long, Cerveza> cervezas = new HashMap<>();
    private final AtomicLong nextId = new AtomicLong();

    public Cerveza crearCerveza(Cerveza cerveza) {
        long id = nextId.incrementAndGet();
        cerveza.setId(id);
        cervezas.put(id, cerveza);
        return cerveza;
    }
```

Estructura



Ejercicio

CervezaRESTController.java

```
@RestController
@RequestMapping("/api/cervezas")
public class CervezaRESTController {

    @Autowired
    private CervezaService cervezaService;

    @PostMapping
    public ResponseEntity<Cerveza> crearCerveza(@RequestBody Cerveza cerveza) {
        return ResponseEntity.status(201).body(cervezaService.crearCerveza(cerveza));
    }
}
```

- @RequestMapping(“/api/cervezas”) factoriza las URL de la API REST. En este caso, se produce un cambio, y es que, a partir de ese punto, todas las URL de la API REST tienen ese inicio de URL.

Sistemas Distribuidos

Bloque II

Desarrollo de sistemas distribuidos y aplicaciones web.

Tema 2.3:

API REST

Nicolás Rodríguez
nicolas.rodriguez@urjc.es