DAW
Desarrollo de Aplicaciones Web
2º Curso

DWES
Desarrollo Web Entorno Servidor

UD 5 Spring Boot

4. Spring APIREST

IES BALMIS
Dpto Informática
Curso 2025-2026
Versión 2 (01/2026)

UD5.4 – Spring APIREST

ÍNDICE

1. Uso de API Rest

1.1 Introducción

API (Application Programming Interface o Interfaz de Programación de Aplicaciones) es un conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca (librería de software) para ser utilizado por otro software como una capa de abstracción.

El término REST (Representational State Transfer) se originó en el año 2000, descrito en la tesis de Roy Fielding, padre de la especificación HTTP.

REST es un modelo de arquitectura web basado en el protocolo HTTP para mejorar las comunicaciones cliente-servidor que utiliza un conjunto de restricciones con las que podemos crear un estilo de arquitectura software.

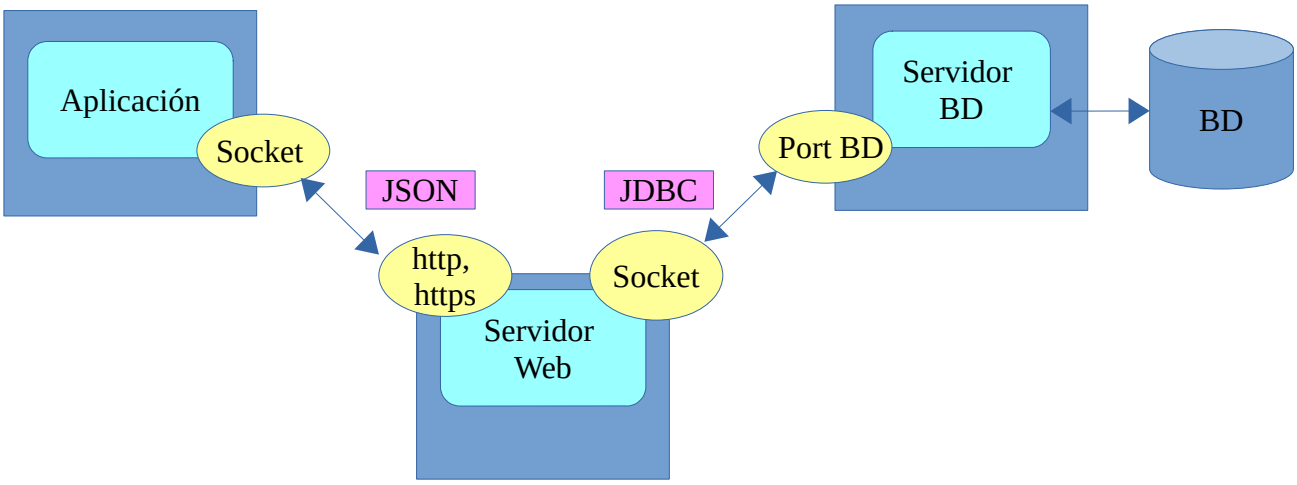
API Rest o Restful es un servicio web que implementa la arquitectura REST y que responde a diferentes peticiones utilizando generalmente un intercambio de datos en formato JSON o XML

Algunos frameworks con los que podremos implementar nuestras API Rest son:

- Spring Boot o JAX-RS (Java API for RESTful Web Services) para Java,
- Django REST framework para Python,
- Laravel para PHP o
- Restify para Node.js

Hoy en día la mayoría de las empresas utilizan API REST para crear servicios. Esto se debe a que es un estándar lógico y eficiente para la creación de servicios web.

Cuando ofrecemos este servicio conectando con un servidor de BD tenemos el siguiente esquema:



Como vimos en la UD1, existen diferentes tecnologías para ofrecer datos usando conexión TCP como:

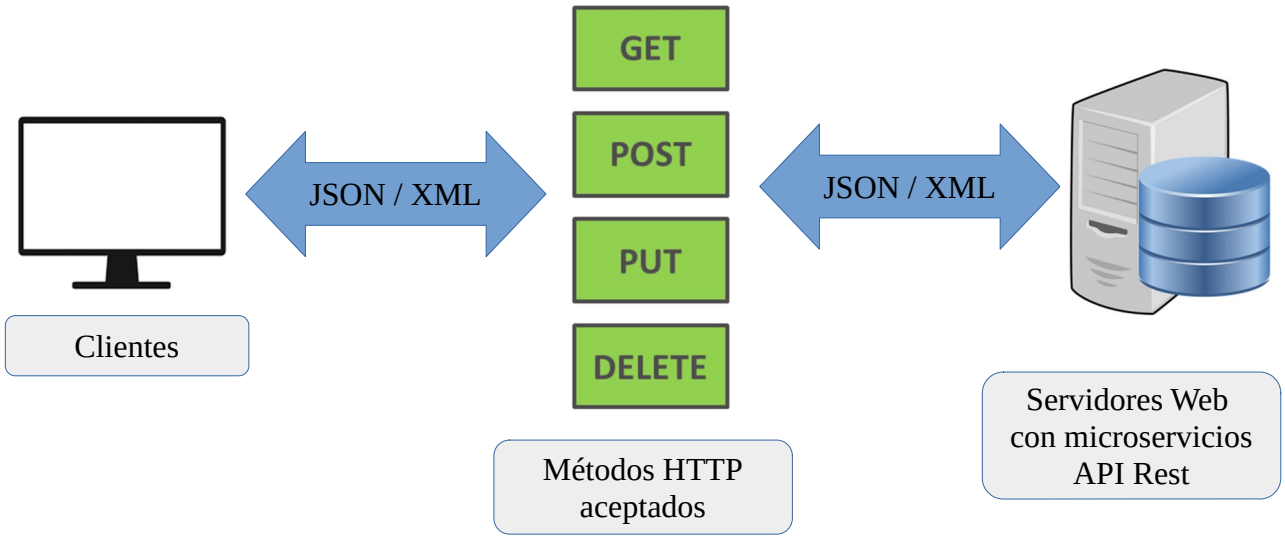
- SOAP (1998)
- API REST (2000)
- GraphQL (2015)

El concepto de **Open Data (Datos abiertos)** es una filosofía y práctica que persigue que determinados tipos de datos estén disponibles de forma libre para todo el mundo, sin restricciones de derechos de autor, de patentes o de otros mecanismos de control. Tiene una ética similar a otros movimientos y comunidades abiertos, como el software libre, el código abierto (open source) y el acceso libre (open access).

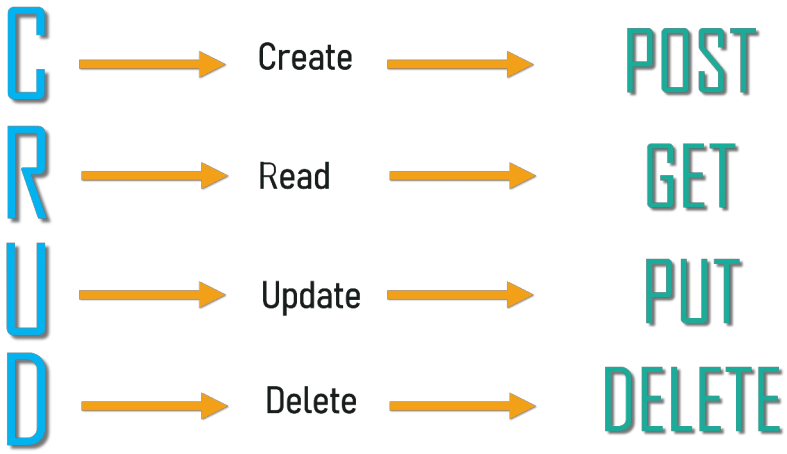
Hay muchos servicios de Open Data y la mayoría se ofrecen a través de servicios API Rest. Algunos ejemplos importantes:

- **OpenStreetMap** => https://wiki.openstreetmap.org/wiki/API
  - Ejemplo: https://www.openstreetmap.org/api/0.6/way/81131976
- **Aemet OpenData** => https://opendata.aemet.es/centrodedescargas/inicio
  - Especificación de métodos:
    - https://opendata.aemet.es/AEMET_OpenData_specification.json
  - Ejemplo de link en Aemet con JSON:
    - https://opendata.aemet.es/opendata/api/maestro/municipios/?api_key=XXX
- **Open Food** => https://es.openfoodfacts.org/ y https://es.openfoodfacts.org/data
  - Ejemplo de link en Open Food con JSON con el código de barra de Nutela:
    - https://es.openfoodfacts.org/producto/80135463/nutella-ferrero
    - https://world.openfoodfacts.org/api/v0/product/80135463.json

Las operaciones más importantes que nos permitirán manipular los recursos son:.



Cada método se usa para realizar una acción en la base de datos:



Aquitectura API Rest
https://juanda.gitbooks.io/webapps/content/api/arquitectura-api-rest.html

Algunos de estos servicios se ofrecen con un **API KEY**, es decir, se necesita registrarse para que el sistema conozca las peticiones de cada usuario. Esto no implica identificación de usuario y contraseña, sino control de peticiones de cada registro.

Algunos ejemplos

API Rest para practicar

- https://jsonplaceholder.typicode.com
- https://reqres.in/
- https://gorest.co.in/
- https://www.mockable.io/
- https://designer.mocky.io/
- http://httpbin.org/#/Request_inspection

API Rest Públicos (la mayoría solo GET)

- https://openlibra.com/es/page/public-api
- https://datos.gob.es/es/apidata
- https://dog.ceo/dog-api/documentation/
- https://api.poems.one/
- https://es.openfoodfacts.org/data
  - (https://es.openfoodfacts.org/)
- https://jikan.moe/
- https://api.nasa.gov/ (con api_key=DEMO_KEY)
- https://openweathermap.org/api
- https://api.mymemory.translated.net
- http://www.omdbapi.com/ (https://es.omdb.org/)

→ https://www.youtube.com/watch?v=xOsqjAyIWwI
dog.ceo, jikan, jsonplaceholder,
regres.in, omdb, openweathermap

Repositorios de API REST Públicos (la mayoría solo GET)

- https://ichi.pro/es/una-lista-seleccionada-de-100-api-publicas-geniales-y-divertidas-para-inspirar-su-proximo-proyecto-8109114517939
- https://any-api.com/

Herramientas gráficas para probar un API REST

- https://es.sensedia.com/post/rest-api-understand-the-step-by-step-to-perform-tests

Para probar podemos usar la utilidad **curl**. Puedes consultar algunos ejemplos prácticos de CURL en:

- https://geekflare.com/es/curl-command-usage-with-example/
- https://terminalcheatsheet.com/es/guides/curl-rest-api

Descargar utilidad CURL
https://curl.se/windows/

Por ejemplo, abriendo una ventana de línea de comandos CMD:

```
C:\> curl -X GET https://jsonplaceholder.typicode.com/albums/2
C:\> curl https://jsonplaceholder.typicode.com/albums/2
{
  "userId": 1,
  "id": 2,
  "title": "sunt qui excepturi placeat culpa"
}
```

```
C:\> curl https://jsonplaceholder.typicode.com/users/2
{
  "id": 2,
  "name": "Ervin Howell",
  "username": "Antonette",
  "email": "Shanna@melissa.tv",
  "address": {
    "street": "Victor Plains",
    "suite": "Suite 879",
    "city": "Wisokyburgh",
    "zipcode": "90566-7771",
    "geo": {
      "lat": "-43.9509",
      "lng": "-34.4618"
    }
  },
  "phone": "010-692-6593 x09125",
  "website": "anastasia.net",
  "company": {
    "name": "Deckow-Crist",
    "catchPhrase": "Proactive didactic contingency",
    "bs": "synergize scalable supply-chains"
  }
}
```

1.2 Probar el funcionamiento de un servicio API Rest

Entraremos en la URL Base del API Rest de ejemplo:

`http://riconet.es/fp/apirest`

Probar con el navegador el método GET

Sin instalar ningún plugin en el navegador, podemos probar el método **GET**. Basta con introducir la URL de los ejemplos:

```
http://riconet.es/fp/apirest/datos
http://riconet.es/fp/apirest/libros
http://riconet.es/fp/apirest/libros/3
http://riconet.es/fp/apirest/libros/count
http://riconet.es/fp/apirest/libros/titulo/?desde=M
```

El formato obtenido por defecto es HTML para la primera y JSON para el resto.

Probar API Rest con Postman

Para probar un API Rest, lo ideal es utilizar una aplicación específica que nos permita parametrizar todos los datos necesarios y **Postman** es una de las más utilizadas.
La versiones 10 y posteriores necesitan de login en la nube para activar las colecciones, mientras que la 9 lo permite sin identificación.

Postman – Web oficial	
https://www.postman.com/downloads/	Última versión
https://dl.pstmn.io/download/version/9.31.28/win64	Versión 9

Generalmente la información que vamos a utilizar en el envío son:

Método	Aunque hay más, se suelen usar GET, POST, PUT y DELETE
URL Base Install (target)	Es la parte inicial de la URL de tipo http que contiene hasta la carpeta donde instalamos nuestro API Rest
URL API Path (path)	Es la parte final de la URL que se reescribe y procesa para procesar la petición (REQUEST). A este proceso también se le llama "URL Amigable".
Datos de QUERY	Es la parte de la URL posterior al carácter **`'?'`** donde se añaden parejas de **parámetro=valor** separadas por el carácter **`'&'`**
Header => Content-Type	Es el tipo de los datos de envío. En API Rest se suele usar json o xml y se indica con el valor "**application/json**" y "**application/xml**" respectivamente
Body => Datos de envío	Es una cadena de caracteres en el formato indicado por "**Content-Type**"
Header => Accept	Es el formato en el se desea que el servidor nos devuelva los datos

La respuesta contendrá al menos la siguiente información:

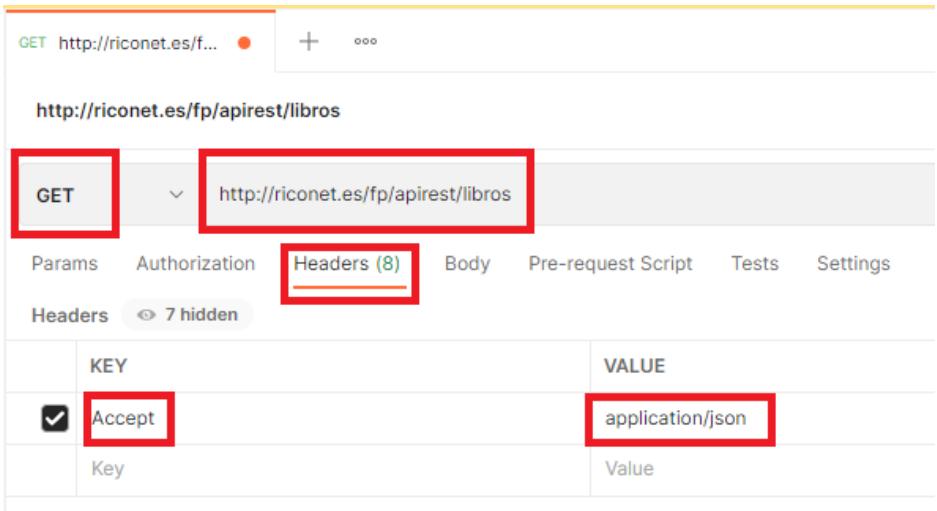
Response => Data	Es una cadena de caracteres en el formato indicado por "**Accept**" en la petición (REQUEST)
Response => Status	Código de estado de la respuesta (**HTTP Status Code**). Es un número entero de 3 dígitos donde el primer dígito del Código de estado define la clase de respuesta y los dos últimos dígitos el tipo de respuesta https://www.tutorialspoint.com/http/http_status_codes.htm

Veamos el formato general:

`http://dominio/target/path?queryString`

Probaremos algunos ejemplos con **Postman** para comprobar el funcionamiento.
Crea una colección para almacenar las llamadas o bien, importa la colección proporcionada por el profesor denominada **POSTMAN-riconet.json**:

Obtener en JSON todos los registros de la tabla libros	
Método	**`GET`**
URL	**`http://riconet.es/fp/apirest/libros`**
Headers => Content-Type	
Body => Datos de envío	
Headers => Accept	**`application/json`**



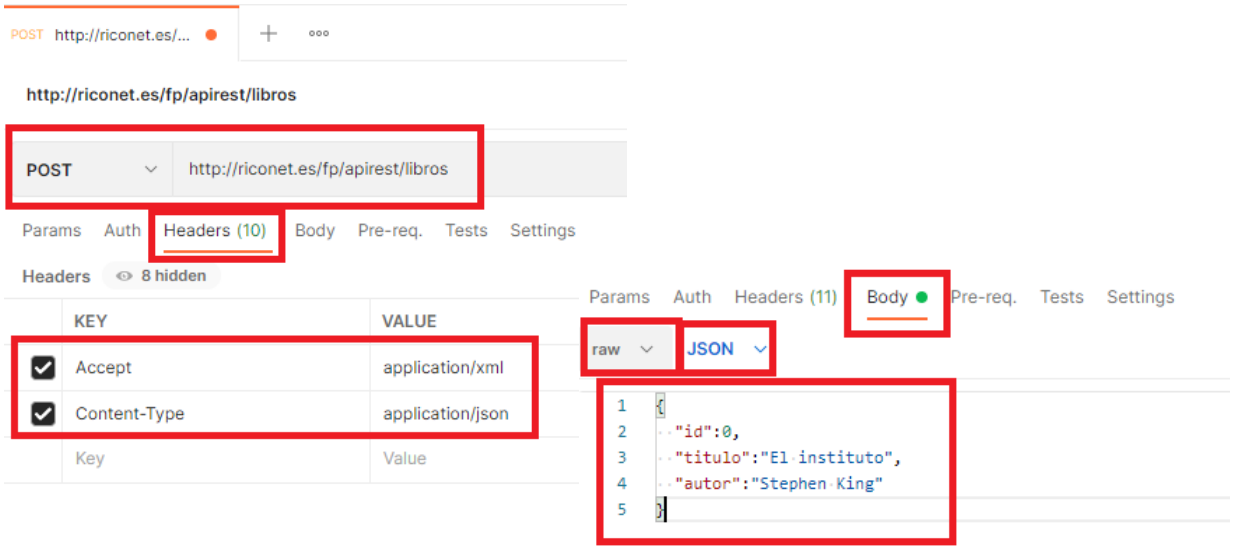
Obtener en XML el registro con id=3 de la tabla libros	
Método	`GET`
URL	`http://riconet.es/fp/apirest/libros/3`
Headers => Content-Type	
Body => Datos de envío	
Headers => Accept	`application/xml`

Insertar en la tabla libros el equivalente a:
`INSERT INTO libros (id, titulo, autor)`
`VALUES (0, 'El instituto', 'Stephen King');`
enviando los datos en JSON y recibiendo la respuesta en XML	

Método	`POST`
URL	`http://riconet.es/fp/apirest/libros`
Headers => Content-Type	`application/json`

Body => Datos de envío	`Body => Raw => JSON`
	`{`
	`"id":0,`
	`"titulo":"El instituto",`
	`"autor":"Stephen King"`
	`}`

Headers => Accept	`application/xml`



Se puede consultar la información de especificación completa del API Rest de ejemplo pulsando en el botón de "**Descargar PDF**":

http://riconet.es/fp/apirest

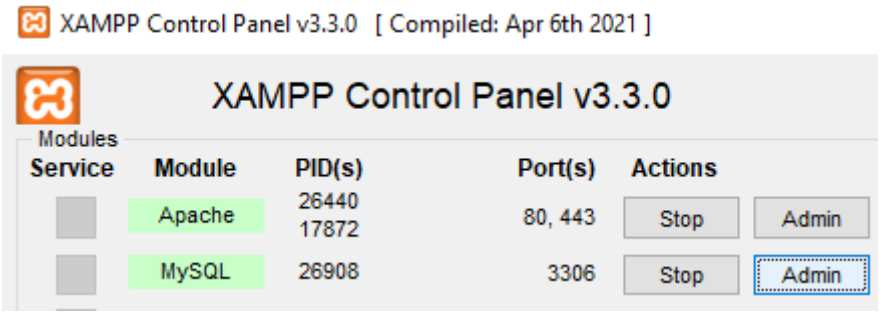
1.3 Instalación de un API Rest con PHP

El profesor entregará el código desarrollado de un **API Rest en PHP** que funciona son un servidor web **Apache + PHP** como el incluido en **XAMPP**.

Este código analiza la base de datos con la que se conecta y muestra un sencillo **API** para acceder a la información de cada tabla de forma independiente, es decir, sin relaciones. Para la **instalación** seguiremos los siguientes pasos:

Paso 1) Crear la BD en MariaDB con phpMyAdmin

En nuestro ejemplo, deberemos iniciar los servicios con **xampp-control.exe**
Recuerda que si tenemos MySQL80 iniciado es necesario detenerlo.



y crear la BD **bibliotecah2** ejecutando las siguientes instrucciones con **phpMyAdmin**:

```
CREATE SCHEMA bibliotecah2 DEFAULT CHARACTER SET utf8mb4 COLLATE
utf8mb4_general_ci;

USE bibliotecah2;

CREATE TABLE autores (
  cod     VARCHAR(5) PRIMARY KEY,
  nombre  VARCHAR(60)
);

CREATE TABLE libros (
  id           INTEGER PRIMARY KEY AUTO_INCREMENT,
  titulo       VARCHAR(60),
  codautor     VARCHAR(5),
  FOREIGN KEY (codautor) REFERENCES autores(cod)
);

INSERT INTO autores (cod, nombre) VALUES
  ('WSHAK', 'William Shakespeare'),
  ('MCERV', 'Miguel de Cervantes'),
  ('FROJA', 'Fernando de Rojas');
```

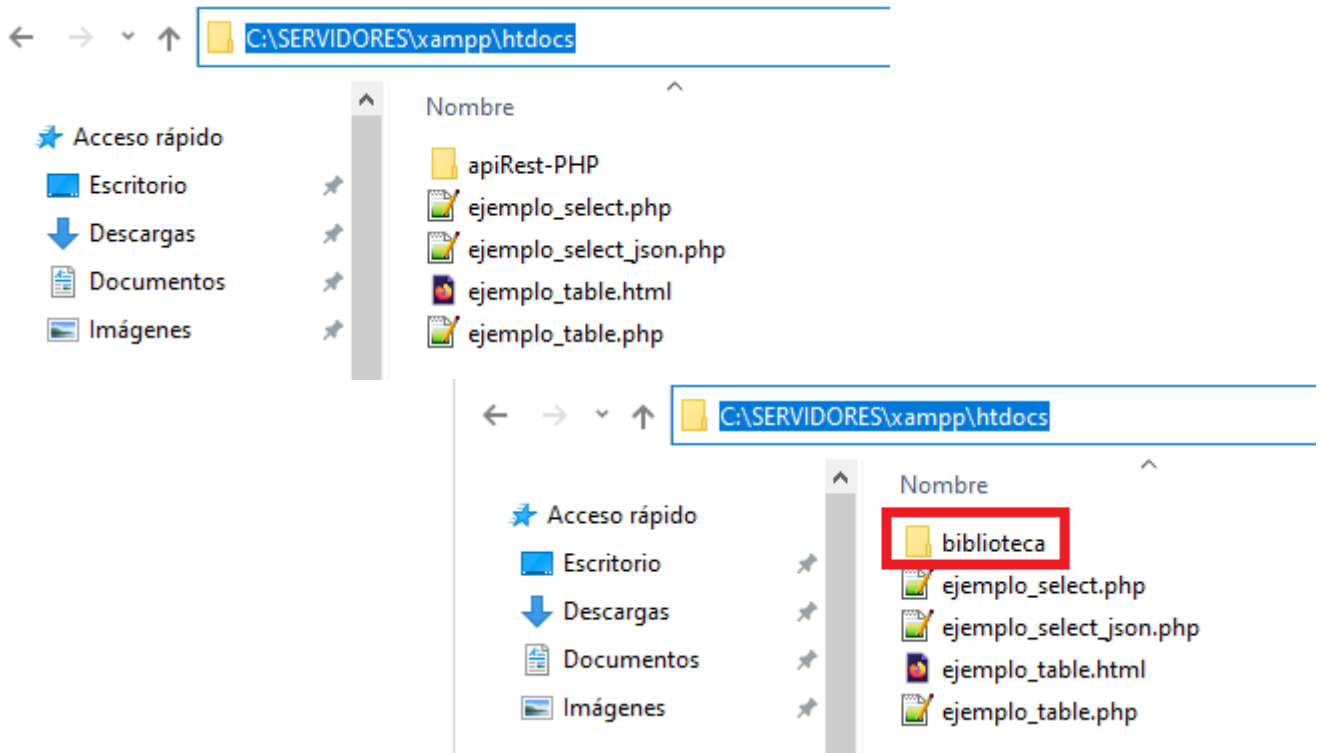
```
USE bibliotecah2;

INSERT INTO libros (id, titulo, codautor) VALUES
  (1, 'Macbeth', 'WSHAK'),
  (2, 'La Celestina (Tragicomedia de Calisto y Melibea)',
'FROJA'),
  (3, 'Don Quijote de la Mancha', 'MCERV'),
  (4, 'La tempestad', 'WSHAK'),
  (5, 'La Galatea', 'MCERV'),
  (6, 'Los trabajos de Persiles y Sigismunda', 'MCERV'),
  (7, 'Novelas ejemplares', 'MCERV');

SELECT *
  FROM autores, libros
  WHERE autores.cod = libros.codautor;
```

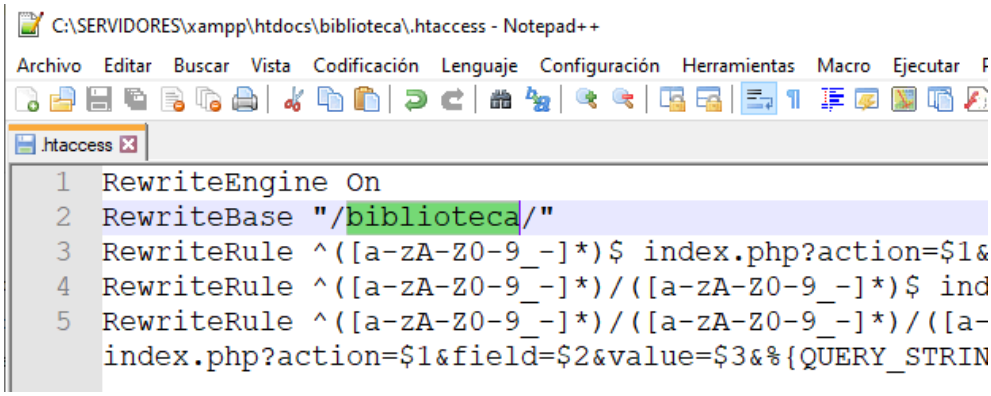
Paso 2) Copiar la aplicación API Rest en Apache

Copiaremos el código proporcionado por el profesor en la carpeta htdocs y cambiaremos su nombre al **Context Path**, por ejemplo **biblioteca**:



Paso 3) Configurar API Rest - Context Path

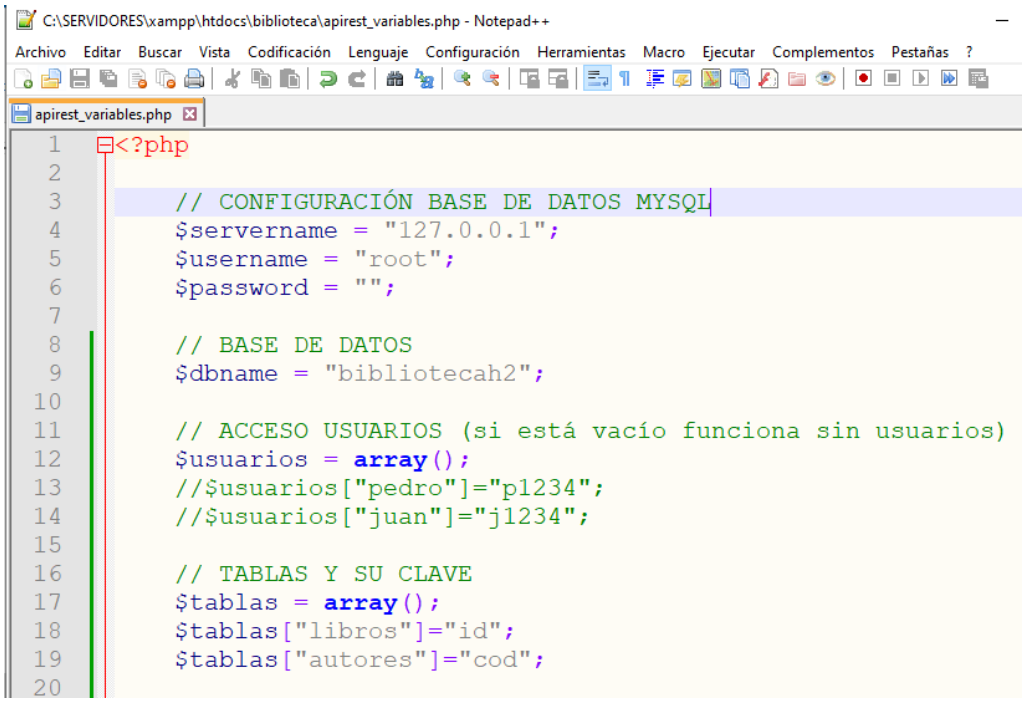
Editaremos el fichero **.htaccess** incluido en la aplicación, donde configuraremos el **Context Path**, que debe coincidir con el nombre de la carpeta, que en nuestro ejemplo es biblioteca:



Paso 4) Configurar API Rest - Conexión a MariaDB/MySQL

Editaremos el fichero **apirest_variables.php** y completaremos los datos. El archivo contiene 4 secciones:

- **CONFIGURACIÓN**: Datos de conexión a la BD
- **BASE DE DATOS:** Base de datos a mostrar por el API Rest
- **USUARIOS:** Array de usuarios y contraseñas. Si se deja vacío es público
- **TABLAS:** Las tablas indicando su clave



Paso 5) Probar funcionamiento

Abrir en el navegador la dirección del API Rest:

http://localhost/biblioteca/

2. Crear APIREST con Spring Boot

2.1 ResponseEntity y anotaciones de clase

Tal y como hemos visto, el modelo de arquitectura REST devuelve un objeto Response con dos partes:

- **Response => Data**
- **Response => Status**

Spring Boot nos ofrece una clase **ResponseEntity** para crear este objeto con los **datos** a mapear a JSON (Data) y es **status** que indica el resultado.

Para comprobar este nuevo funcionamiento, se ha copiado y modificado el proyecto **user-h2-jpa-v1** para crear **api-user-h2-jpa-v1**

api-user-h2-jpa-v1

-Cambiamos variables en el **application.properties** para el nuevo proyecto con el context-path a **/apiuserh2jpav1**

-Añadir al Path en el UserController **@RequestMapping("/rest")**

-Añadir en **UserController** la anotación **@CrossOrigin(origins = "*")**

-Cambiar **UserController** para devolver **ResponseEntity<User>** o **ResponseEntity<List<User>>, en vez de User o List<User>**

La URL para probar será:

http://localhost:8080/apiuserh2jpav1/rest/users
http://localhost:8080/apiuserh2jpav1/rest/users/2
http://localhost:8080/apiuserh2jpav1/rest/users/mayor/7
http://localhost:8080/apiuserh2jpav1/rest/users/count

Para poder devolver un **ResponseEntity** con **status** y **contenido** en UserController usaremos:

return ResponseEntity.status(HttpStatus.XXXXXX).body(XXXXXX);

Existe otras formas de de devolver los valores (PERO NOSOTROS NO LAS USAREMOS):

- Método que implica status: ResponseEntity.**ok**(userService.findAll());
- Build: ResponseEntity.**notFound**().build()
- ...

La clase **GlobalExceptionHandler** no es necesario modificarla porque ya devuelve **ResponseEntity**.

Uso de @CrossOrigin(origins = "")*

CORS (Cross-Origin Resource Sharing) es un mecanismo de seguridad del navegador que restringe las solicitudes HTTP entre diferentes orígenes (domains, ports, protocols).

Sin CORS, un navegador bloquearía las solicitudes AJAX/Fetch de tu frontend (ej: http://localhost:3000) a tu backend (ej: http://localhost:8080).

Esta anotación permite que tu API REST acepte solicitudes desde cualquier origen (dominio). Es útil durante desarrollo, pero en producción deberías restringir el acceso solo a los orígenes permitidos y no a todos con el *.

2.2 Anotaciones @RequestHeader y @CookieValue

Para comprobar este nuevo funcionamiento, se ha copiado y modificado el proyecto **api-user-h2-jpa-v1** para crear **api-user-h2-jpa-v2**

api-user-h2-jpa-v2

-Cambiamos variables en el **application.properties** para el nuevo proyecto con el context-path a **/apiuserh2jpav2**

-Modificar el método **count** para devolverlo solo si se solicita en **JSON** usando **@RequestHeader**

-Añadir métodos para gestionar cookies con **@CookieValue**

Uso de @RequestHeader

En ocasiones podemos necesitar consultar alguna de variables de HEADER enviadas por el cliente. Por ejemplo, en el método **count** se ha añadido la comprobación del valor de **Accept** y crear el **ResponseEntity** indicando el formato **JSON**.

Uso de @CookieValue

En ocasiones necesitamos almacenar información de estado a través de cookies.

En este ejemplo almacenaremos en una cookie par indicar el lenguaje en el que deseamos trabajar.

Se han añadido 3 métodos nuevos:

- **getCookieInfo**: mostrar valores de
  - la cookie de session asignada por el servidor a nuestra conexión
  - el lenguaje a usar
- **setCookieLang**: actualizar el lenguaje (es o en)
- **getMenoresAdultos**: mostrar cuántos menores y adultos tenemos

Resumen

@RequestMapping	Para añadir parte del **path** a laURL
ResponseEntity	Para devolver el contenido con el **status**
@CrossOrigin	Para permitir que acepte solicitudes desde cualquier **origen**
@RequestHeader	Para gestionar las variables y sus valores del **header** de la solicitud
@CookieValue	Para gestionar **cookies** en el servidor asociados a la **session**

2.3 Actualizaciones

Para que un API REST esté completo faltarían las actualizaciones.

Para ello usaremos las anotaciones:

@PostMapping	Para realizar un INSERT
@PutMapping	Para realizar un UPDATE
@DeleteMapping	Para realizar un DELETE

Además, para garantizar que todas las instrucciones de grabación se realizan dentro de la misma transacción, usaremos otra anotación:

@Transactional	Todas las instrucciones realizadas en la Base de Datos se realizarán de forma transaccional, es decir, se realizan todas o ninguna, pero nunca solo algunas de ellas.

Para comprobar este nuevo funcionamiento, se ha copiado y modificado el proyecto **api-user-h2-jpa-v2** para crear **api-user-h2-jpa-v3**

api-user-h2-jpa-v3

-Cambiamos variables en el **application.properties** para el nuevo proyecto con el context-path a **/apiuserh2jpav1**

-Añadir los métodos **save** y **deleteById** en **UserRepository** (LOS HEREDADOS NO HACE FALTA)

-Añadir los métodos **save, update** y **deleteById** en **UserService**

-Añadir la anotación **@Transactional** para optimizar las instrucciones SQL sobre la BD

-Añadir en **UserController** métodos para el **POST, PUT y DELETE**

-Añadir a **data.sql** la actualización del siguiente ID con los datos existentes:

```
ALTER TABLE users ALTER COLUMN id
        RESTART WITH (SELECT COALESCE(MAX(id), 0) + 1 FROM users);
```

-Añadir en la clase **GlobalExceptionHandler** ls gestión de dos excepciones nuevas:

- **HttpMessageNotReadableException**: Error al serializar
- **MethodArgumentNotValidException**: Error al validar un campo por anotaciones en la clase de datos

-Añadir las anotaciones **@Valid** en **UserController**

A continuación mostramos las características incluidas en cada proyecto:

	PROYECTOS		
	api-user-h2-jpa-v1	**api-user-h2-jpa-v2**	**api-user-h2-jpa-v3**
Thymeleaf	❌	❌	❌
Lombok	✅	✅	✅
Controlador	✅ **@RestController Map<String, Object>**	✅ **@RestController Map<String, Object>**	✅ **@RestController Map<String, Object>**
Service con Repository	✅	✅	✅
GlobalExceptionHandler	✅	✅	✅
Documentación HTML	✅	✅	✅
Acceso a Datos	✅ **JpaRepository SQL**	❌ **JpaRepository SQL**	❌ **JpaRepository SQL**
Base de Datos	✅ **H2 Database**	✅ **H2 Database**	✅ **H2 Database**
Persistencia	✅ FILE	✅ FILE	✅ FILE
Carga de Datos inicial	✅ **shema.sql + data.sql**	✅ **shema.sql + data.sql**	✅ **shema.sql + data.sql** **ALTER COLUMN id**
@RequestMapping @CrossOrigin	✅	✅	✅
ResponseEntity	✅	✅	✅
@RequestHeader		✅	✅
@CookieValue		✅	✅
POST, PUT, DELETE @Transactional			✅
@Valid			✅

2.4 APIREST con dos tablas

Para comprobar este nuevo funcionamiento, se ha copiado y modificado el proyecto **user-h2-jpa2-v1** para crear **api-h2-jpa2-v1**

api-h2-jpa2-v1

-Cambiamos variables en el **application.properties** para el nuevo proyecto con el context-path a **/apiuserh2jpa2v1**

-Añadir los métodos **save** y **deleteById** en **UserRepository** (LOS HEREDADOS NO HACE FALTA)

-Añadir los métodos **save, update** y **deleteById** en **UserService**

-Añadir la anotación **@Transactional** para optimizar las instrucciones SQL sobre la BD

-Añadir en **UserController y RoleController** métodos para el **POST, PUT y DELETE**
-Añadir las anotaciones **@Valid** en **UserController y RoleController**

-Añadir a **data.sql** la actualización del siguiente ID con los datos existentes:

```
ALTER TABLE users ALTER COLUMN id
        RESTART WITH (SELECT COALESCE(MAX(id), 0) + 1 FROM users);
```

-Comprobar en la clase **GlobalExceptionHandler** la gestión de dos excepciones:
- **HttpMessageNotReadableException**: Error al serializar
- **MethodArgumentNotValidException**: Error al validar un campo por anotaciones en la clase de datos

-Añadir los **métodos PUT** para actualizar las relaciones

Métodos de actualización

En **UserRepository** y **RoleRepository** no necesitamos añadir ningún método extra ya que son heredados de **JpaRepository**.

En **UserService** y **RoleService** añadiremos los métodos **save**, **update** y **deleteById** para disponer de las 3 principales actualizaciones.
Además, añadiremos:

- **@Transactional(readOnly = true)** en los métodos de consultas para no bloquear las tablas ya se realizarían en modo lectura.
- **@Transactional** en los métodos de actualización para que todos los UPDATE, INSERT o DELETE derivados de estas acciones se realicen de forma transaccional, es decir, se ejecutan todas o ninguna, pero no a medias (unas sí y otras no).

En **UserController** y **RoleController** añadiremos los métodos para el POST, PUT y DELETE que:

- recibirán las URL y los datos,
  - **@Valid** comprobará las restricciones indicadas en la clases de datos **User** y **Role** de tipo **jakarta.validation.constraints.***
  - **@RequestBody** recogerá y mapeará el **JSON** recibido en **body**
- procesarán las acciones
- controlarán los errores
- devolverán un **ResponseEntity** con los **status** y **body** correspondientes

Modificación del data.sql

Como hemos indicado en **application.properties** que estos **INSERT** se realicen simpre con **spring.sql.init.mode=always**, aunque den error porque el **ID** ya existe el **AUTO_INCREMENT** aumenta por cada instrucción.

Para evitarlo actualizaremos los valores de **AUTO_INCREMENT** almacenados en **id_RESTART** con:

```
ALTER TABLE roles
     ALTER COLUMN id RESTART
         WITH (SELECT COALESCE(MAX(id), 0) + 1 FROM roles);

ALTER TABLE users
     ALTER COLUMN id RESTART
         WITH (SELECT COALESCE(MAX(id), 0) + 1 FROM users);
```

GlobalExceptionHandler

Podemos comprobar que en la clase **GlobalExceptionHandler** se capturan dos excepciones que pueden producirse en estos proyectos:

- **HttpMessageNotReadableException**: Error al serializar
  - Al recibir en el **@RequestBody** un **JSON** incorrecto que no se pueda mapear en la clase indicada, se generará esta excepción
- **MethodArgumentNotValidException**: Error al validar un campo
  - Al indicar **@Valid**, si por anotaciones en la clase de datos no se cumple algún requisito, por ejemplo **@NotBlank** o **@Size**, se generará esta excepción

RECOMENDACIÓN EN FORMATO DE URL		
Context Path	`/basededatos/`	
ApplicationConfig	`/api o /rest o /apirest`	
Controlador	`/tabla`	
Métodos Controlador		**EJEMPLO**
Acción sobre la tabla	/count	/count
Todos	/	/
ID=valor	/{valor}	/2
campo=valor	/campo/{valor}	/edad/35
ID operador valor	/operador/{valor}	/mayor/5
campo operador valor	/campo/operador/{valor}	/edad/mayor/20 /nombre/contiene/za

Métodos PUT para actualizar las relaciones

En el caso de las actualizaciones con dos tablas debemos tener en cuenta algunos detalles:

- De forma general, en la serialización del RequestBody en un POST o PUT los datos de la tabla relacionada no serán tratados
- Para actualizar los datos de una relación podremos usar:
  - **En un @ManyToOne**: un método PUT específico que no llevará RequestBody sino que recibirá los datos en le URL
  - **En @OneToMany**: un método PUT específico que llevará un RequestBody con la lista de datos a relacionar.

En nuestro ejemplo se han añadido dos nuevos métodos PUT en UserController:

- **@PutMapping("/users/{userId}/asignar/roles/{roleId}")**
  - Asigna al usuario con id=userid el role con id=roleId
- **@PutMapping("/users/{userId}/desasignar/role")**
  - Asigna al usuario con id=userid el role null

3. Documentación con SWAGGER

3.1 Generar documentación automática con Swagger

api-h2-jpa2-v2 (PROYECTO MODELO RECOMENDADO)

-Cambiamos variables en el **application.properties** para el nuevo proyecto con el context-path a **/apiuserh2jpa2v2**

-Añadir dependencia de **Swagger** en el pom.xml

-Añadir configuración a **application.properties** de Swagger

-Añadir la clase **OpenApiConfig** con los datos de la aplicación

-Añadir a las clases de datos (**User** y **Role**) descripciones y ejemplos a las **clases** y los **campos** con **@Schema**

-Añadir a los controladores (**UserController** y **RoleController**) la descripción con la anotación **@Tag**

-Añadir a los controladores (**UserController** y **RoleController**) información por cada método con **@Operation** y **@ApiResponses**

-Añadir controller (**SwaggerController**) para redirigir la URL base a la documentación generada por Swagger, y eliminar los archivos anteriores de **/static/*.html**

Dependencia y configuración de Swagger

Para disponer de estas clases que generan la documentación dela APIREST automáticamente debemos incluir la siguiente dependencia de Swagger OpenAPI 3.0

```
<!-- Swagger/OpenAPI 3 -->
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>3.0.0</version>
</dependency>
```

En el **application.properties** debemos indicar las siguiente variables:

- **springdoc.api-docs.path:** Define la ruta URL donde se expone el documento OpenAPI JSON.
- **springdoc.swagger-ui.path:** Define la ruta de la interfaz web de Swagger UI.
- **springdoc.swagger-ui.operationsSorter:** Ordena las operaciones (endpoints) en Swagger UI. Los valores posibles son:
  - **method (por defecto):** Ordena por tipo de método (GET, POST, PUT, DELETE, etc.)
  - **alpha:** Ordena alfabéticamente por nombre de operación
  - **none:** Sin orden específico
- **springdoc.swagger-ui.tagsSorter:** Ordena los tags (grupos de endpoints) en Swagger UI. Los posibles valores son:
  - **alpha:** Ordena alfabéticamente (por defecto)
  - Una función personalizada

En nuestro proyecto son:

```
# Configuración de Swagger/OpenAPI
springdoc.api-docs.path=/api-docs
springdoc.swagger-ui.path=/swagger-ui.html
springdoc.swagger-ui.operationsSorter=method
springdoc.swagger-ui.tagsSorter=alpha
```

Clase OpenApiConfig

Esta clase define los textos de encabezamiento de la documentación.
Aparecen al principio de la página web.

@Schema

En las clases de datos, **User** y **Role** en nuestro ejemplo, añadiremos con **@Schema** una **descripción** y un **nombre** a la clase, y también a cada método.

Estos textos se mostrarán en la documentación.

@Tag, @Operation y @ApiResponses

En los controladores, **UserController** y **RoleController** en nuestro ejemplo, añadiremos con **@Tag** una **descripción** y un **nombre** a la clase.

Además, a cada método le añadiremos información de lo que hace con **@Operation** y una lista de los **status** que devuelve con **@ApiResponses**.

SwaggerController

Por último añadiremos este controlador para regirigir la URL base del proyecto a la documentación web.

El último proyecto de ejemplo facilitado añade a la documentación **Swagger** ejemplos de código JSON tanto para lo que devuelve como para lo que se espera.

api-h2-jpa2-v3 (AMPLIACIÓN - SOLO SI SE ESTIMA CONVENIENTE)

-Cambiamos variables en el **application.properties** para el nuevo proyecto con el context-path a **/apiuserh2jpa2v2**

-Añadir contenido JSON de ejemplo con **@Content** en **@ApiResponses** para los diferentes **status** devueltos

@Content en @ApiResponses

En cada método de los controladores podemos añadir si lo estimamos conveniente para aclarar su funcionamiento, un ejemplo de código **JSON** devuelto.

En algunos casos es adecuado indicar que no se devuelve nada con:

```
content = @Content()
```

En otros, se indicará el tipo de datos devuelto con **mediaType** y un ejemplo con **examples**.

3.2 Resumen para activar Swagger

Para realizar este ejercicio se deberá:

- activar en el **pom.xml** la dependencia de **Swagger**
- añadir la clase **OpenApiConfig** en el Java Package **config**
- añadir la clase **SwaggerController** en el Java Package **controller**
- añadir anotaciones en las clases de datos dentro de **model**
  - **@Schema**
- añadir anotaciones en las clases clases controladores dentro de **controller**
  - **@Tag**
  - **@Operation**
  - **@ApiResponses**
- eliminar la documentación inicial de **/static/index.html**
- configurar variables de swagger en **application.properties**
  - **springdoc.api-docs.path=/api-docs**
  - **springdoc.swagger-ui.path=/swagger-ui.html**
  - **springdoc.swagger-ui.operationsSorter=method**
  - **springdoc.swagger-ui.tagsSorter=alpha**

A continuación mostramos las características incluidas en cada proyecto:

	PROYECTOS		
	api-h2-jpa2-v1	**api-h2-jpa2-v2**	**api-h2-jpa2-v3**
Thymeleaf	❌	❌	❌
Lombok	✅	✅	✅
Controlador	✅ **@RestController Map<String, Object>**	✅ **@RestController Map<String, Object>**	✅ **@RestController Map<String, Object>**
Service con Repository	✅	✅	✅
GlobalExceptionHandler	✅	✅	✅
Documentación HTML	✅	✅	✅
Acceso a Datos	✅ **JpaRepository SQL**	❌ **JpaRepository SQL**	❌ **JpaRepository SQL**
Base de Datos	✅ **H2 Database**	✅ **H2 Database**	✅ **H2 Database**
Persistencia	✅ FILE	✅ FILE	✅ FILE
Carga de Datos inicial	✅ **shema.sql + data.sql ALTER COLUMN id**	✅ **shema.sql + data.sql ALTER COLUMN id**	✅ **shema.sql + data.sql ALTER COLUMN id**
@RequestMapping @CrossOrigin	✅	✅	✅
ResponseEntity	✅	✅	✅
POST, PUT, DELETE @Transactional	✅	✅	✅
@Valid	✅	✅	✅
SWAGGER: pom.xml, application.properties, OpenApiConfig		✅	✅
SWAGGER: @Schema @Tag @Operation @ApiResponses		✅	✅
SWAGGER: @content			✅