



2.1 - Tecnologías de Servicios Web

Tema 3 - APIs REST



Tema 3 – APIs REST

Tema 3.1 – APIs REST con Spring

APIs REST con Spring

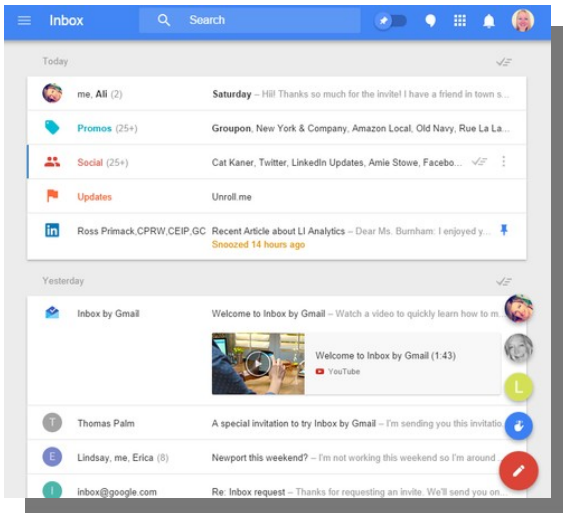
- **Introducción**
- Clientes de APIs REST
- APIs REST con Spring
- Conversión entre objetos y JSON
- Cliente REST en el servidor
- Documentación de APIs REST
- Imágenes

Introducción

- En una aplicación web, el cliente (**navegador**) se comunica con el servidor (**servidor web**) usando el protocolo **http**
- En una aplicación web las peticiones **http** devuelven un **documento HTML** que será **visualizado** por el navegador
- En las aplicaciones con **AJAX** y las aplicaciones **SPA**, las peticiones http se pueden usar intercambiar **información estructurada** entre el navegador y el servidor (pero no HTML)

Introducción

- Ejemplo de aplicación SPA haciendo peticiones **http** a un servidor para obtener **información**



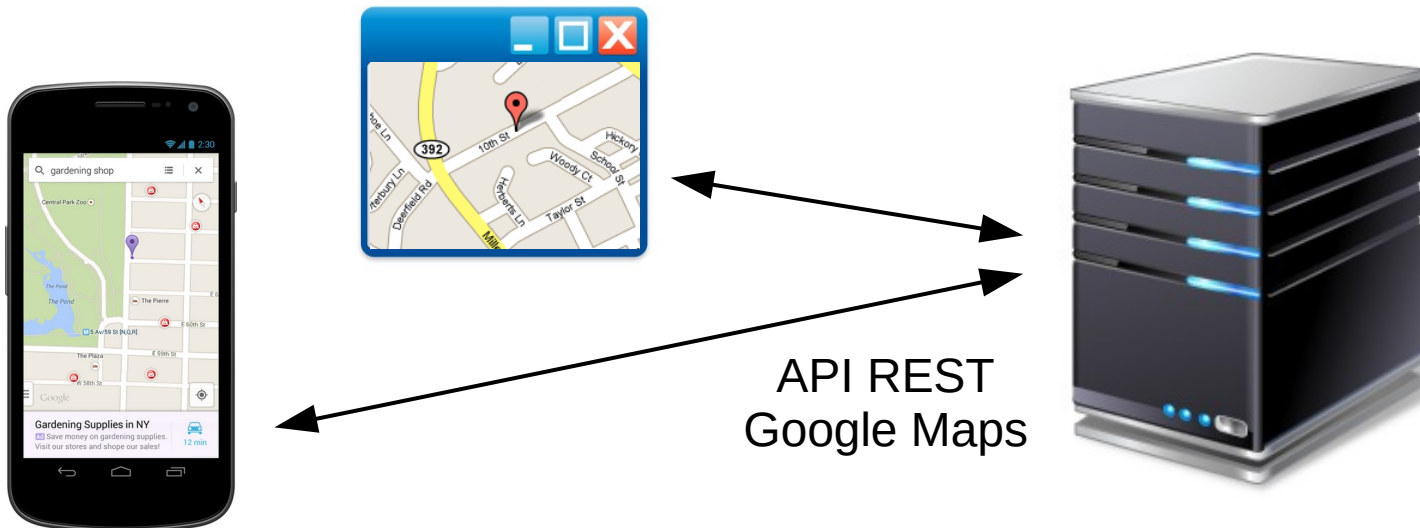
`http://www.miweb.com/users/34`

```
{  
  "name": "Pepe",  
  "surname": "López",  
  "age": 45,  
  "email": "pepe@miweb.com",  
  "friends": [ "María", "Juan" ]  
}
```



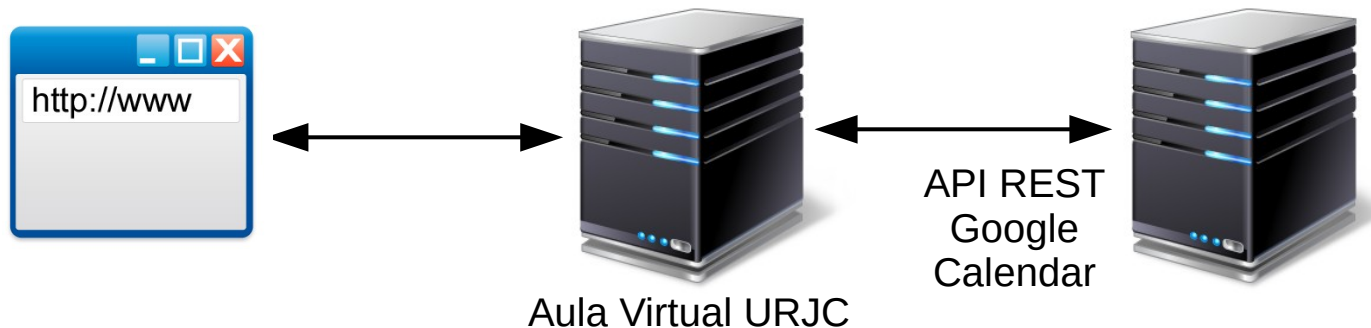
Introducción

- Además de un **navegador web**, otros tipos de aplicaciones también usan las **APIs REST**
 - **Otros clientes:** Apps móviles, TVs, consolas...
 - Ejemplo: La aplicación de **Google Maps** para Android es un cliente de la misma **API REST** de la web SPA



Introducción

- Además de un **navegador web**, otros tipos de aplicaciones también usan las **APIs REST**
 - **Otros servidores:** El backend de una aplicación web puede usar APIs REST además de sus bases de datos para ofrecer sus servicios
 - Ejemplo: El Aula Virtual de la URJC podría usar la API REST de Google Calendar para publicar eventos



APIs REST

- Una API REST es un tipo de servicio web que se basa en los **elementos del protocolo http**
- REST es acrónimo de ***REpresentational State Transfer***, Transferencia de Estado Representacional.
- El término se acuñó en el año 2000, en la tesis doctoral de **Roy Fielding**, uno de los principales autores de la especificación del protocolo **HTTP**

<https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

APIs REST

- **REST vs SOAP (*Simple Object Access Protocol*)**
 - SOAP es otro protocolo para implementar servicios web
 - Usa http como transporte, pero no aprovecha muchas de sus características como hace REST
 - Es más complejo de implementar que REST (clientes y servidores)
 - Se usaba antes de que se popularizaran las APIs REST

<https://en.wikipedia.org/wiki/SOAP>

Características de una API REST

- Existen **muchas formas** de implementar un protocolo sobre http
- Hay enfoques **más puristas** y otros **más prácticos**
- A lo largo de los años se han ido definiendo un **conjunto de buenas prácticas**
- Existen aspectos bastante **consensuados** y otros en los que se admiten **diversas variantes**

Características de una API REST

- Estudiaremos las **buenas prácticas básicas** más usadas en la actualidad

apigee

Google Cloud

Web API Design: The Missing Link

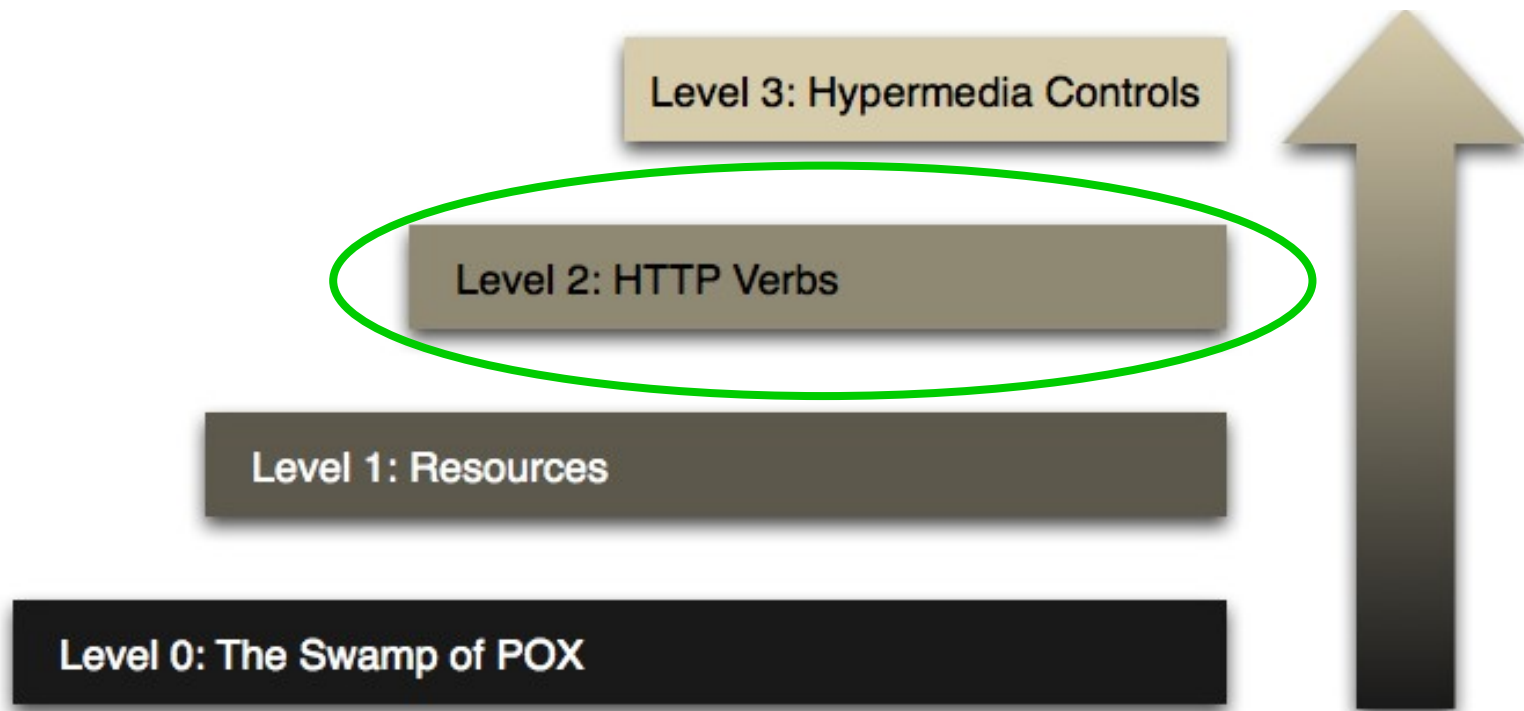
Best Practices for Crafting Interfaces that Developers Love

<https://cloud.google.com/files/apigee/apigee-web-api-design-the-missing-link-ebook.pdf>

<https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api>

Características de una API REST

- Niveles de cumplimiento de los principios REST



<http://martinfowler.com/articles/richardsonMaturityModel.html>

Características de una API REST

- Una **API REST** es un servicio de red que ofrece operaciones **CRUD** (**creación, lectura, actualización y borrado**) sobre recursos (items de información)
- Se aprovecha de todos los aspectos del **protocolo http**: URL, métodos, códigos de estado, cabeceras...
- Los recursos se **representan** en un formato estructurado (campos), habitualmente **JSON** (aunque puede ser XML, Protocol Buffers, ...)

Características de una API REST

- Estudiaremos el **nivel 2** de API REST (el más extendido):
 - La **URI** referencia a uno o más recursos
 - Las **operaciones** que se quieren realizar con ese recurso son los **métodos del protocolo HTTP**
 - Se usan los códigos de **estado http** para notificar errores (p.e. 404 Not found)
 - Los recursos se **representan** en JSON, un **formato estructurado**

Características de una API REST

- La URI referencia a uno o más recursos
 - Deben seguir un **formato** muy concreto
 - Parte de la URL indentifica el **tipo de recurso** (en plural) y otra parte es el **identificador** del recurso concreto
 - Ejemplos:
 - <https://server/posts/vendo-moto-23-10-2014>
 - <https://server/cars/fpx-4567-x>
 - <http://server/posts/453>
 - <http://server/users/bob>

Características de una API REST

- La URI referencia a uno o más recursos
 - Cuando un recurso está asociado a otro recurso se pueden poner ambos en la URL
 - Ejemplos:
 - <http://server/users/bob/posts/comparto-piso>
 - <http://server/users/bob/posts/44>
 - <http://server/posts/542/comments/74>
 - <http://server/sessions/563664/streams/AVXX>

Características de una API REST

- La URI referencia a uno o más recursos
 - A veces las URLs hacen referencia a varios recursos
 - Los parámetros de la URL se pueden usar para filtrar el conjunto total
 - Ejemplos:
 - <http://server/users/>
 - <http://server/users/?active=true>
 - <http://server/users/bob/posts/?visibility=public>
 - <http://server/posts/542/comments/>
 - <http://server/sessions/563664/streams/>

Características de una API REST

- La URI referencia a uno o más recursos
 - La URI está formada por nombres, no verbos
 - Ejemplos:

<http://server/create/user/>

<http://server/deleteUser/345>

<http://server/addNewPost>

<http://server/startRecording>

<http://server/stopRecording>

**URLs NO adecuadas
para una API REST**

Características de una API REST

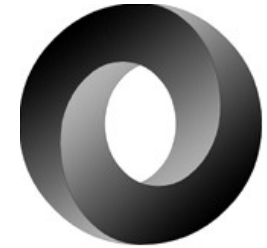
- Las operaciones se codifican como métodos http
 - **GET:** Devuelve el recurso (o recursos) identificado en la URI
 - **DELETE:** Borra el recurso identificado en la URI
 - **POST:** Añade un nuevo recurso. El recurso se envía en el cuerpo de la petición
 - **PUT:** Reemplaza el recurso identificado en la URI. El nuevo recurso se envía en el cuerpo de la petición
 - **PATCH:** Actualizar el recurso identificado en la URI. Los cambios se envían en el cuerpo de la petición
 - Existen más...

Características de una API REST

- Se usan los códigos de estado http para notificar el resultado de la operación:
 - **100-199:** No están definidos. Describen fases de ejecución de la petición.
 - **200-299:** La petición fue procesada correctamente.
 - **300-399:** El cliente debe hacer acciones adicionales para completar la petición, por ejemplo, una redirección a otra página.
 - **400-499:** Se usa en casos en los que el cliente ha realizado la petición incorrectamente (404 No existe).
 - **500-599:** Se usa cuando se produce un error procesando la petición.

Características de una API REST

- Los recursos se representan en JSON
 - Acrónimo de *JavaScript Object Notation*
 - Es fácil de mapear a objetos en memoria
 - Representa **objetos del dominio** del servicio



```
{
  "id": 147633,
  "name": "John Ronald Reuel Tolkien",
  "enabled": true,
  "books": [
    {"title": "El Silmarillion", "year": 1936 },
    {"title": "El hobbit", "year": 1932 },
    {"title": "El Señor de los Anillos", "year": 1954}
  ]
}
```

<http://www.json.org/>

Características de una API REST

- Los recursos se representan en JSON

- Petición:

URL: <http://server/bookmarks/6>

Método: GET

- Respuesta:

Header: content-Type:application/json

Body:

```
{
  "id":6,
  "uri": "http://bookmark.com/web",
  "description": "A description"
}
```

Características de una API REST

- Los recursos se representan en JSON
 - Se pueden usar otros formatos, pero son menos comunes



<https://www.w3.org/XML/>

Protocol Buffers

<https://developers.google.com/protocol-buffers/>

```
<author>
  <id>147633</id>
  <name>John Ronald Reuel Tolkien</name>
  <enabled>true</enabled>
  <books>
    <book>
      <title>"El Silmarillion"</title>
      <year>1936</year>
    </book>
    <book>
      <title>"El hobbit"</title>
      <year>1932</year>
    </book>
    <book>
      <title>"El Señor de los Anillos"</title>
      <year>1954</year>
    </book>
  </books>
</author>
```

APIs REST con Spring

- Introducción
- **Cientes de APIs REST**
- APIs REST con Spring
- Conversión entre objetos y JSON
- Cliente REST en el servidor
- Documentación de APIs REST
- Imágenes

Cientes de APIs REST

- Las APIs REST están diseñadas para ser utilizadas por **aplicaciones** (no por humanos)
- Todos los **lenguajes de programación** disponen de librerías para uso de APIs REST (Java, JavaScript...)
- Como desarrolladores podemos usar **herramientas interactivas** para hacer pruebas (hacer peticiones y ver las respuestas)

Cientes de APIs REST

- Vamos a familiarizarnos con las herramientas interactivas de acceso a APIs REST
- Haremos una petición GET a una API REST pública y analizaremos la información obtenida
- Usaremos la API REST de libros de Google Play

```
https://www.googleapis.com/books/v1/volumes?q=intitle:javascript
```

- Documentación de la API REST
 - <https://developers.google.com/books/docs/v1/using>

Cientes de APIs REST

- **Cliente JavaScript**

- Las aplicaciones web con **AJAX** o con arquitectura **SPA**, implementadas con **JavaScript**, usan servicios **REST** desde el navegador
- Se pueden usar APIs REST usando la **API estándar** del browser o **librerías externas** (jQuery)

Cientes de APIs REST

front-ejem1

- Cliente JavaScript: jQuery
 - Muestra en la consola el resultado de la API REST

script.js

```
let result = await fetch("https://www.googleapis.com/books/v1/volumes?q=intitle:java");
let books = await result.json();
console.log(books);
```

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Cientes de APIs REST

- Cliente JavaScript: jQuery

front-ejem1

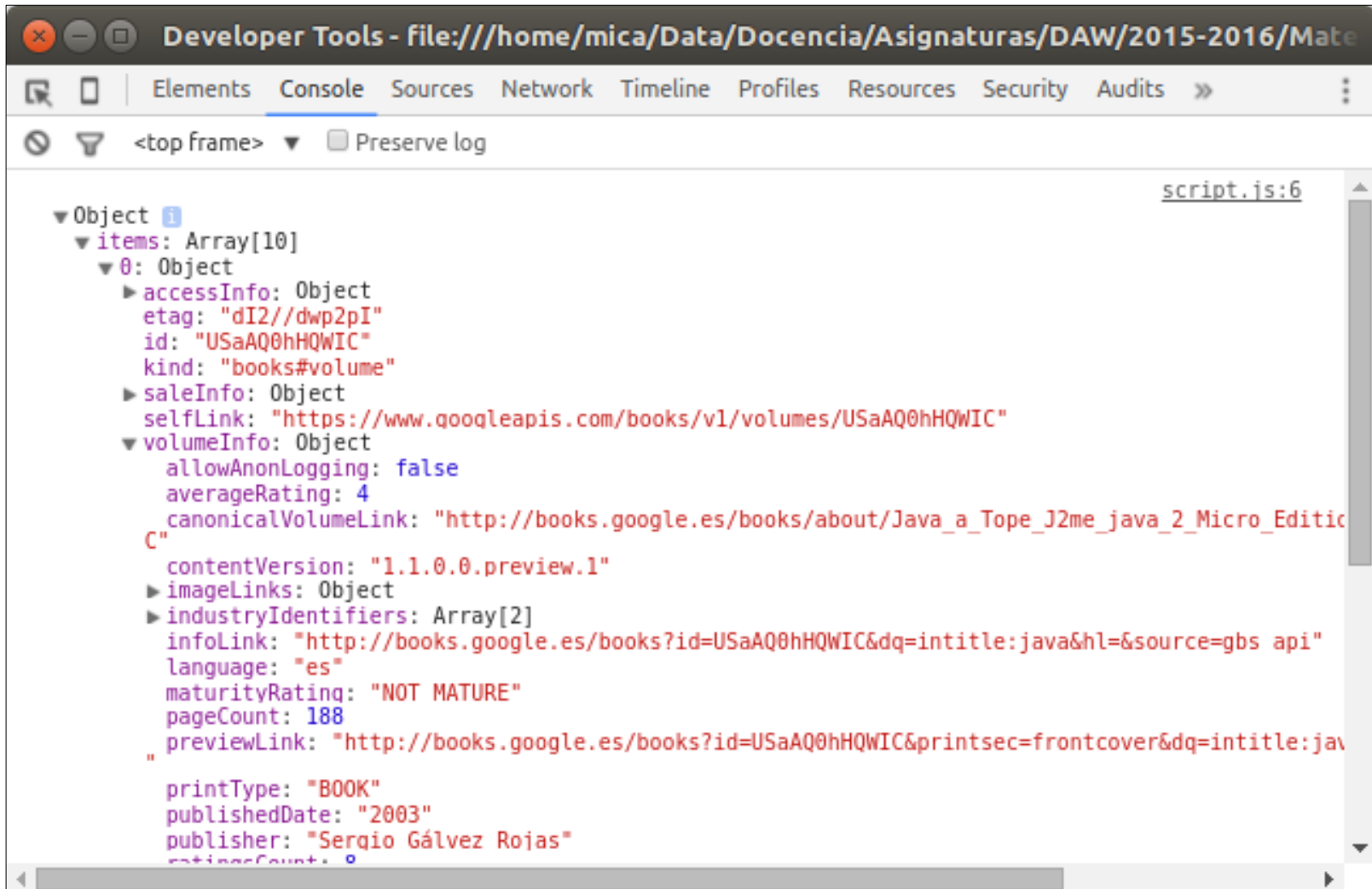
```
<!DOCTYPE html>
<html>

<head>
  <script type="module" src="script.js"></script>
</head>

<body>
</body>

</html>
```

Cientes de APIs REST



The screenshot shows the Chrome Developer Tools console with the 'Console' tab selected. The breadcrumb path is '<top frame>' and the 'Preserve log' checkbox is unchecked. The console displays a log entry for 'script.js:6' showing a JSON object. The object has a property 'items' which is an array of 10 objects. The first object in the array is expanded, showing the following properties:

- `accessInfo`: Object
 - `etag`: "dI2//dwp2pI"
 - `id`: "USaAQ0hHQWIC"
 - `kind`: "books#volume"
- `saleInfo`: Object
 - `selfLink`: "https://www.googleapis.com/books/v1/volumes/USaAQ0hHQWIC"
- `volumeInfo`: Object
 - `allowAnonLogging`: false
 - `averageRating`: 4
 - `canonicalVolumeLink`: "http://books.google.es/books/about/Java_a_Tope_J2me_java_2_Micro_EditicC"
 - `contentVersion`: "1.1.0.0.preview.1"
 - `imageLinks`: Object
 - `industryIdentifiers`: Array[2]
 - `infoLink`: "http://books.google.es/books?id=USaAQ0hHQWIC&dq=intitle:java&hl=&source=gb&api"
 - `language`: "es"
 - `maturityRating`: "NOT MATURE"
 - `pageCount`: 188
 - `previewLink`: "http://books.google.es/books?id=USaAQ0hHQWIC&printsec=frontcover&dq=intitle:jav"
 - `printType`: "BOOK"
 - `publishedDate`: "2003"
 - `publisher`: "Sergio Gálvez Rojas"
 - `ratingsCount`: 0

Cientes de APIs REST

front-ejem2

- Cliente JavaScript: jQuery
 - Muestra en la página los títulos de los libros

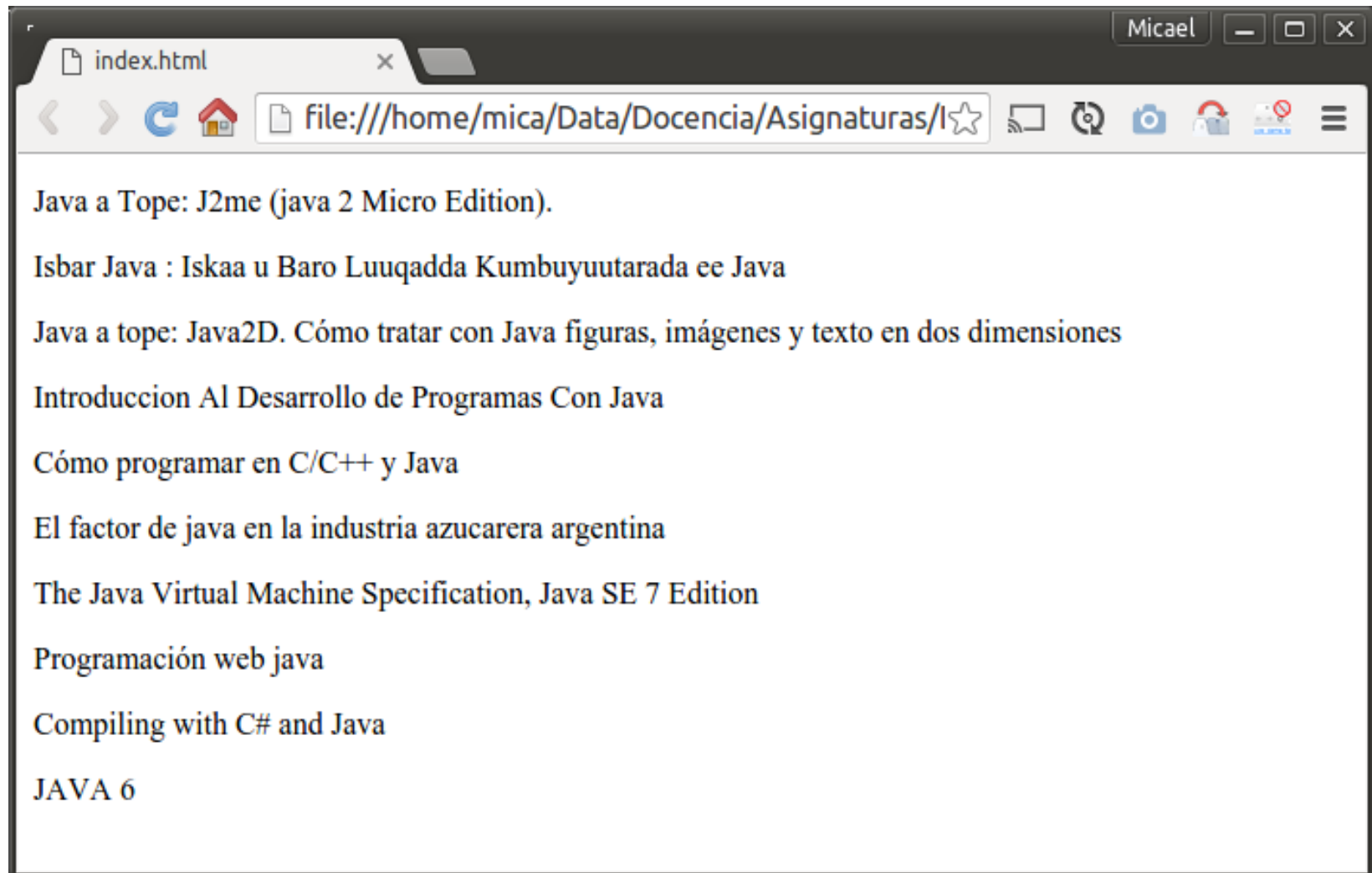
script.js

```
let result = await fetch(
  "https://www.googleapis.com/books/v1/volumes?q=intitle:java");
let books = await result.json();
let body = document.getElementsByTagName("body");
for (let i = 0; i < books.items.length; i++) {
  body[0].innerHTML += `

${books.items[i].volumeInfo.title}</p>`;
}

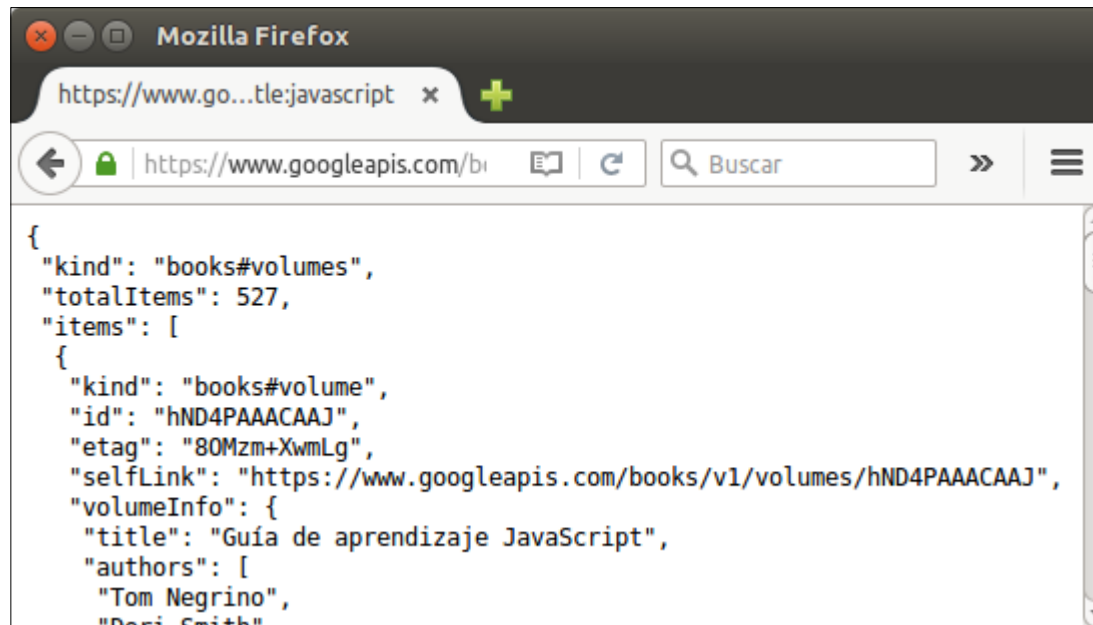

```

Cientes de APIs REST



Cientes de APIs REST

- Herramientas interactivas
 - El navegador web es una herramienta básica que se puede usar para hacer peticiones GET a APIs REST



Cientes de APIs REST

- **Herramientas interactivas**
 - Tipos
 - Integradas en el entorno de desarrollo
 - Extensiones del navegador
 - Permiten
 - Realizar peticiones REST con cualquier método (GET, POST, PUT...)
 - Especificar URL, cabeceras (headers)...
 - Analizar la respuesta: Cuerpo, status http...

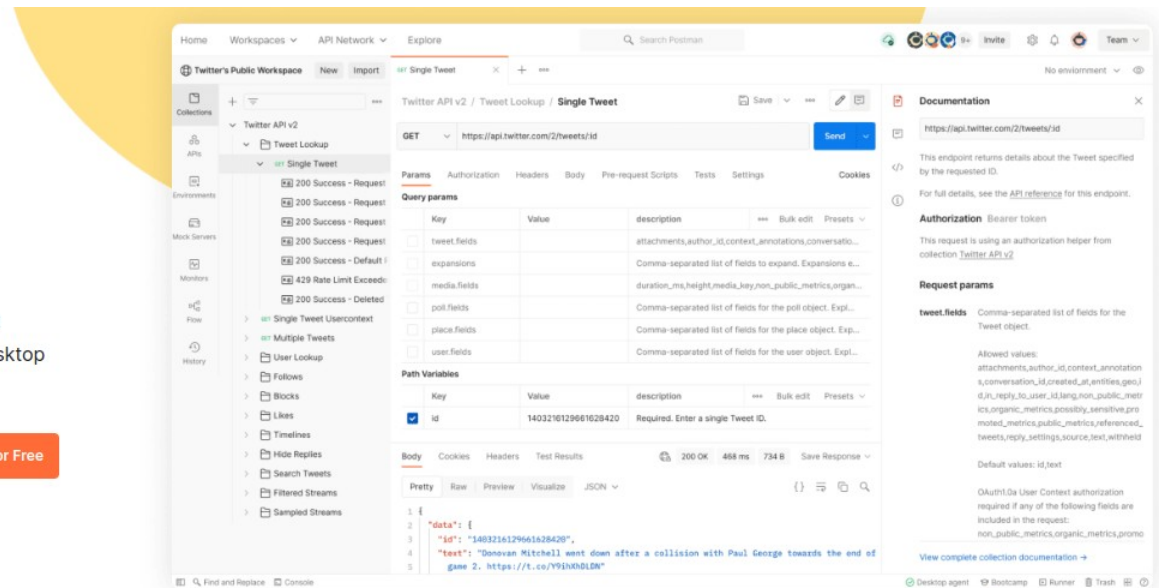
Cientes de APIs REST

- Postman - REST Client

M_ APIs together

Over 20 million developers use Postman. Get started by signing up or downloading the desktop app.

Download the desktop app



What is Postman?

Postman is an API platform for building and using APIs. Postman simplifies each step of the API lifecycle and streamlines collaboration so you can create better APIs—faster.



<https://www.getpostman.com/>

Postman

File Edit View Help

Home Workspaces ▾ Explore

Search Postman

Sign In Create Account

Scratch Pad New Import

Overview GET https://www.googleapi

No Environment

https://www.googleapis.com/books/v1/volumes?q=intitle:javascript

Save

Send

GET https://www.googleapis.com/books/v1/volumes?q=intitle:javascript

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	q	intitle:javascript			
	Key	Value	Description		

Body Cookies Headers (13) Test Results

200 OK 933 ms 35.86 KB Save Response ▾

Pretty Raw Preview Visualize JSON

```
1
2  "kind": "books#volumes",
3  "totalItems": 200,
4  "items": [
5    {
6      "kind": "books#volume",
7      "id": "XGvjswEACAAJ",
8      "etag": "tjsYSkFYwjM",
9      "selfLink": "https://www.googleapis.com/books/v1/volumes/XGvjswEACAAJ",
10     "volumeInfo": {
11       "title": "El gran libro de HTML5, CSS3 y JavaScript",
12       "authors": [
13         "J. D. Gauchat"
14       ],
15       "publishedDate": "2017",
16     }
```

https://www.googleapis.com/books/v1/volumes?q=intitle:javascript

Cientes de APIs REST

ejem1

- **API REST Posts**
 - Ofrece de operaciones de consulta, creacion, modificación y borrado
 - Las propiedades de un post son: user, title y text
 - La API rest está implementada en el proyecto **ejem1**

 [MasterCloudApps / 2.1.Tecnologias-de-servicios-web](#)

https://github.com/MasterCloudApps/2.1.Tecnologias-de-servicios-web/tree/master/tema3-rest/Spring/rest_ejem1

Cientes de APIs REST

ejem1

- **API REST Posts**
 - Consulta de posts
 - Method: GET
 - URL: `http://127.0.0.1:8080/posts/`
 - Result:

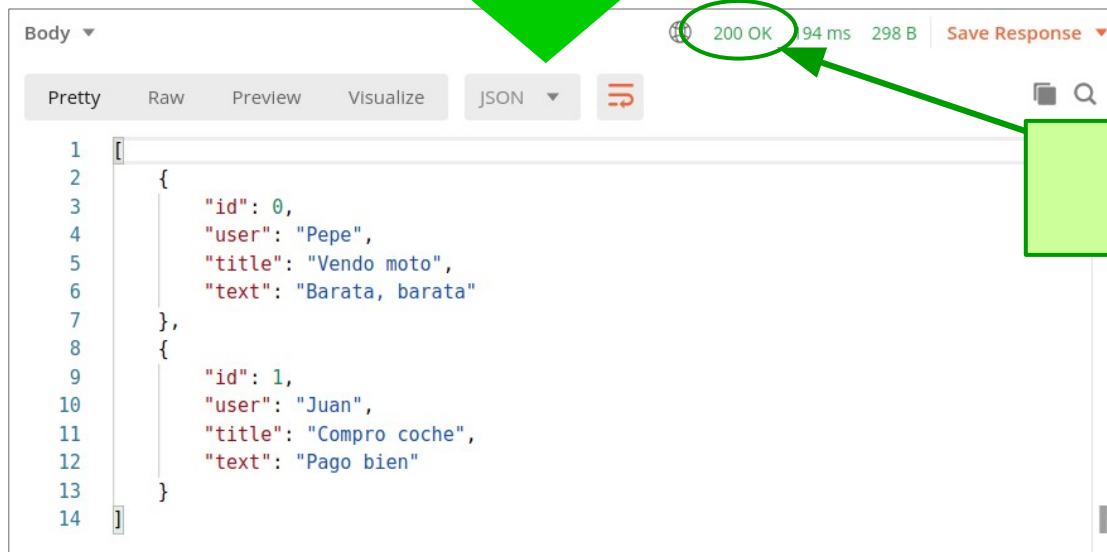
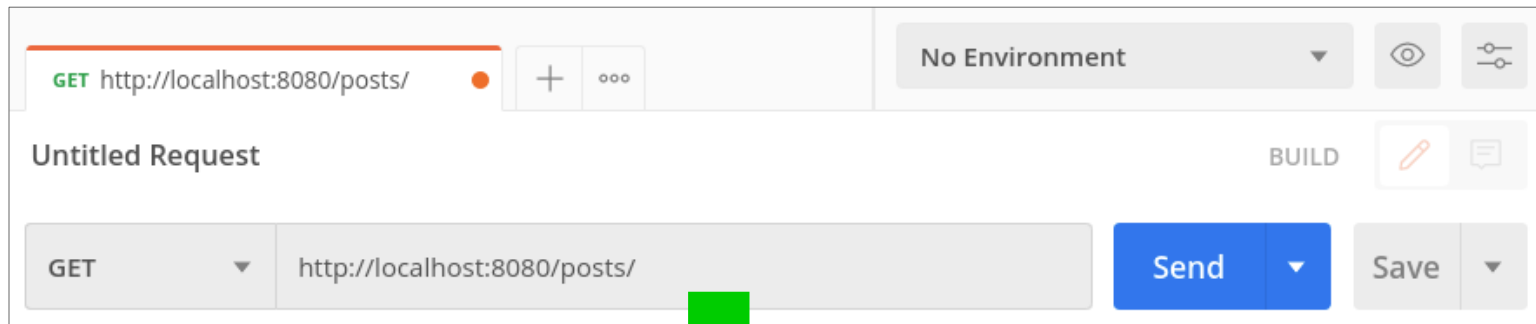
```
[
  { "id": 0, "user": "Pepe", "title": "Vendo moto", "text": "Barata, barata"},
  { "id": 1, "user": "Juan", "title": "Compro coche", "text": "Pago bien"}
]
```

- Status code: 200 (OK)

Cientes de APIs REST

ejem1

- GET /posts/



Status de la
petición

Cientes de APIs REST

ejem1

- **API REST Posts**
 - Consulta de un post concreto
 - Method: GET
 - URL: `http://127.0.0.1:8080/posts/1`
 - Result:

```
{ "id": 1, "user": "Pepe", "title": "Vendo moto", "text": "Barata, barata"}
```

- Status code: 200 (OK)

Cientes de APIs REST

ejem1

- **API REST Posts**

- Creación de un post
 - Method: POST
 - URL: `http://127.0.0.1:8080/posts/`
 - Headers: Content-Type: `application/json`
 - Body:

```
{ "user": "Alberto", "title": "Cambio bici", "text": "MTB por carretera"}
```

- Result:

```
{"id":2, "user": "Alberto", "title": "Cambio bici", "text": "MTB por carretera"}
```

- Status code: 201 (Created)
- Headers: Location: `http://127.0.0.1:8080/posts/2`

Cientes de APIs REST

ejem1

- **POST /posts/**

Para POST y PUT es necesario especificar que el cuerpo (**body**) es de tipo **raw** y en concreto, **JSON**

The screenshot shows a REST client interface. At the top, there are two tabs: 'GET http://localhost:8080/posts/' and 'POST http://localhost:8080/posts/'. The 'POST' tab is active. Below the tabs, the 'POST' method is selected, and the URL 'http://localhost:8080/posts/' is entered. The 'Body' tab is selected, and the body is set to 'raw' (circled in green). The 'JSON' format is selected (circled in green). The body content is a JSON object:

```
{  "user": "Alberto",  "title": "Cambio bici",  "text": "MTB por carretera"}
```

. A large green arrow points from the 'POST' tab to the 'Body' tab. Below the 'Body' tab, the response is shown. The status is '201 Created' (circled in green), with a response time of '55 ms' and a size of '284 B'. The response body is also JSON:

```
{  "id": 2,  "user": "Alberto",  "title": "Cambio bici",  "text": "MTB por carretera"}
```

. A green box with the text 'Es status code es 201' has an arrow pointing to the '201 Created' status. The 'Save Response' button is visible in the top right of the response section.

Cientes de APIs REST

ejem1

- **API REST Posts**

- Reemplazo de posts

- Method: PUT
- URL: `http://127.0.0.1:8080/posts/o`
- Headers: Content-Type: `application/json`
- Body:

```
{ "id": 0, "user": "Pepe", "title": "Vendo moto", "text": "Regalada" }
```

- Result:

```
{ "id": 0, "user": "Pepe", "title": "Vendo moto", "text": "Regalada" }
```

- Status code: `200 (OK)`

Cientes de APIs REST

ejem1

- **API REST Posts**

- Borrado de posts
 - Method: DELETE
 - URL: `http://127.0.0.1:8080/posts/1`
 - Result:

```
{ "id": 0, "user": "Pepe", "title": "Vendo moto", "text": "Regalada"}
```

- Status code: 200 (OK)

<https://stackoverflow.com/questions/6439416/deleting-a-resource-using-http-delete>

APIs REST con Spring

- Introducción
- Clientes de APIs REST
- **APIs REST con Spring**
- Conversión entre objetos y JSON
- Cliente REST en el servidor
- Documentación de APIs REST
- Imágenes

APIs REST con Spring

- Para implementar una **API REST** con **Spring** se puede usar:
 - **JAX-RS**
 - ▢ Estandar Java Enterprise Edition
 - ▢ *Java API for RESTful Web Services*
 - ▢ <https://jersey.java.net/>
 - **Spring MVC**
 - ▢ Framework Spring (no estandar)
 - ▢ Mismo sistema usado para generar páginas web

APIs REST con Spring

- **Spring MVC** es una parte de Spring para la construcción de aplicaciones web
- Sigue la arquitectura **MVC** (*Model View Controller*)
- **Permite estructurar la aplicación en:**
 - **Model:** Modelos de datos (objetos Java)
 - **View:**
 - **Web:** Plantilla que genera la página HTML.
 - **REST:** Conversión del modelo a JSON (automático)
 - **Controller:** Controlador que atiende las peticiones HTTP que llegan del navegador

APIs REST con Spring

ejem1

- **Ciclo de vida completa de un recurso**
 - **Endpoints REST**
 - Creación: POST
 - Consulta un recurso: GET
 - Consulta de varios recursos: GET
 - Borrado: DELETE
 - Actualización: PUT
 - **Gestión de recursos**
 - PostService: Mapa en memoria indexado por id

APIs REST con Spring

ejem1

- **Aplicación principal**

- La aplicación se ejecuta como una **app Java normal**
- Botón derecho proyecto > Run as... > Java Application...

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

• pom.xml

Proyecto padre
para aplicaciones
SpringBoot

Dependencias
necesarias para
implementar
aplicaciones web
**Spring MVC y
SpringBoot**

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>es.codeurjc</groupId>
  <artifactId>rest_ejer1</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.0-RC2</version>
    <relativePath/>
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>17</java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
  </dependencies>

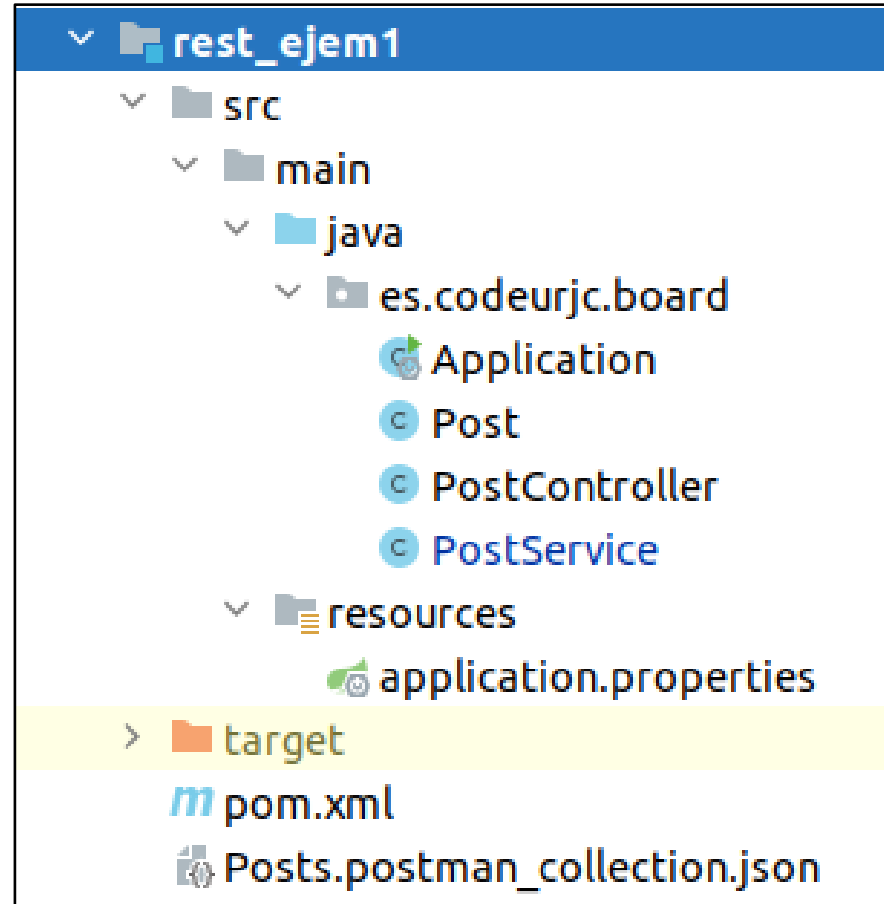
  ...

</project>
```

APIs REST con Spring

ejem1

- Estructura de la aplicación



APIs REST con Spring

ejem1

- **Devolver un recurso concreto (GET)**
 - Con **@GetMapping** se indica que el método atiende peticiones **GET**
 - Se devuelve un array con todos los recursos
 - Si no hay recursos, el array se devuelve vacío

APIs REST con Spring

ejem1

- Devolver todos los recursos (GET)

```
@RestController
public class PostController {

    @Autowired
    private PostService posts;

    @GetMapping("/posts/")
    public Collection<Post> getPosts() {
        return posts.findAll();
    }
    ...
}
```

APIs REST con Spring

ejem1

- **Devolver un recurso concreto (GET)**
 - Con **@GetMapping** se indica que el método atiende peticiones **GET**
 - El id del recurso se condifica en la URL y se accede a él usando un **@PathVariable**
 - Si el recurso existe se devuelve, y si no, se devuelve **404 NOT FOUND**. Por eso el método devuelve un **ResponseEntity**

APIs REST con Spring

ejem1

- Devolver un recurso concreto (GET)

```
@GetMapping("/posts/{id}")
public ResponseEntity<Post> getPost(@PathVariable long id) {

    Post post = posts.findById(id);

    if (post != null) {
        return ResponseEntity.ok(post);
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

El id está en la URL
y se accede con
@PathVariable

Se devuelve el
objeto o **NOT
FOUND**

APIs REST con Spring

ejem1

- **Nuevo recurso (POST)**
 - Con **@PostMapping** se indica que el método atiende peticiones **POST**
 - El cuerpo de la petición se obtiene con un parámetro anotado con **@RequestBody**
 - Se **devuelve el nuevo objeto** al cliente (con un id)
 - Se devuelve la **HEADER Location** con la URL del objeto recién creado

<https://www.rfc-editor.org/rfc/rfc9110.html#name-location>

APIs REST con Spring

ejem1

- Nuevo recurso (POST)

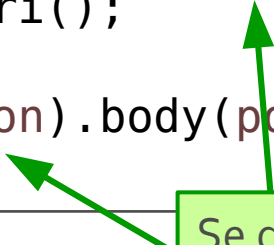
```
@PostMapping("/posts/")
public ResponseEntity<Post> createPost(@RequestBody Post post) {

    posts.save(post);

    URI location = fromCurrentRequest().path("/{id}")
        .buildAndExpand(post.getId()).toUri();

    return ResponseEntity.created(location).body(post);
}
```

Se define la
cabecera Location



APIs REST con Spring

ejem1

- **Borrar un recurso (DELETE)**
 - Con **@DeleteMapping** se indica que el método atiende peticiones **DELETE**
 - El id del recurso se codifica en la URL y se accede a él usando un **@PathVariable**
 - Si el recurso existe se borra y opcionalmente se devuelve
 - Si no existe, se devuelve **404 NOT FOUND**. Por eso el método devuelve un **ResponseEntity**

APIs REST con Spring

ejem1

- **Borrar un recurso (DELETE)**

```
@DeleteMapping("/posts/{id}")
public ResponseEntity<Post> deletePost(@PathVariable long id) {

    Post post = posts.findById(id);

    if (post != null) {
        posts.deleteById(id);
        return ResponseEntity.ok(post);
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

APIs REST con Spring

ejem1

- **Reemplazar un recurso (PUT)**
 - Con **@PutMapping** se indica que el método atiende peticiones **PUT**
 - El id del recurso se condifica en la URL y se accede a él usando un **@PathVariable**
 - El nuevo anuncio se envía en el body y se accede con **@RequestBody**
 - Si el recurso existe se actualiza y se devuelve de nuevo
 - Si no existe, se devuelve **404 NOT FOUND**. Por eso el método devuelve un **ResponseEntity**

APIs REST con Spring

ejem1

- Reemplazar un recurso (PUT)


```
@PutMapping("/posts/{id}")
public ResponseEntity<Post> replacePost(@PathVariable long id,
    @RequestBody Post newPost) {

    Post post = posts.findById(id);

    if (post != null) {
        newPost.setId(id);
        posts.save(newPost);

        return ResponseEntity.ok(post);
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

En caso de
discrepancia
prevalece el id de
la URL



APIs REST con Spring

ejem2

- Factorizar URL mapping en el controller
 - Cuando **todas las URLs** de un controlador empiezan de forma **similar**, se puede poner la anotación **@RequestMapping** a nivel de **clase** con la parte común
 - Cada **método** sólo tiene que incluir la **parte propia** (si existe)

APIs REST con Spring

ejem2

- Factorizar URL mapping en el controller



```
@RestController
@RequestMapping("/posts")
public class PostController {

    @GetMapping("/")
    public Collection<Post> getPosts() {...}

    @GetMapping("/{id}")
    public ResponseEntity<Post> getPost(@PathVariable long id) {...}

    @PostMapping("/")
    public ResponseEntity<Post> createPost(@RequestBody Post post) {...}

    @PutMapping("/{id}")
    public ResponseEntity<Post> replacePost(@PathVariable long id, @RequestBody Post newPost) {...}

    @DeleteMapping("/{id}")
    public ResponseEntity<Post> deletePost(@PathVariable long id) {...}
}
```

APIs REST con Spring

- Acceso a los parámetros de la URL
 - Filtros, definición de propiedades, términos de búsqueda, etc..
 - `http://portal.com/posts?city=Madrid`

```
@GetMapping("/posts")
public List<Post> getPosts(@RequestParam String city) {
    ...
}
```


Ejercicio 1

- Implementa una **API REST** en el servidor para gestionar **Items**
- Los items se gestionarán en **memoria** (como en el ejemplo de los posts)

Ejercicio 1

- **API REST Items**

- Consulta de items
 - Method: GET
 - URL: `http://127.0.0.1:8080/items/`
 - Result:

```
[
  { "id": 1, "description": "Leche", "checked": false },
  { "id": 2, "description": "Pan", "checked": true }
]
```

- Status code: 200 (OK)

Ejercicio 1

- **API REST Items**
 - Consulta de un item concreto
 - Method: GET
 - URL: `http://127.0.0.1:8080/items/1`
 - Result:

```
{ "id": 1, "description": "Leche", "checked": false }
```

- Status code: 200 (OK)

Ejercicio 1

- **API REST Items**

- Creación de un item

- Method: POST
- URL: `http://127.0.0.1:8080/items/`
- Headers: Content-Type: `application/json`
- Body:

```
{ "description" : "Galletas", "checked": true }
```

- Result:

```
{ "id": 2, "description" : "Galletas", "checked": true }
```

- Status code: `201 (Created)`
- Header: Location=`http://127.0.0.1:8080/items/2`

Ejercicio 1

- **API REST Items**

- Reemplazo de un item

- Method: PUT
- URL: `http://127.0.0.1:8080/items/1`
- Headers: Content-Type: `application/json`
- Body:

```
{ "id": 1, "description" : "Leche", "checked": true }
```

- Result:

```
{ "id": 1, "description" : "Leche", "checked": true }
```

- Status code: `200 (OK)`

Ejercicio 1

- **API REST Items**

- Borrado de un item
 - Method: DELETE
 - URL: `http://127.0.0.1:8080/items/1`
 - Result:

```
{ "id": 1, "description" : "Leche", "checked": true }
```

- Status code: 200 (OK)

APIs REST con Spring

- Introducción
- Clientes de APIs REST
- APIs REST con Spring
- **Conversión entre objetos y JSON**
- Cliente REST en el servidor
- Documentación de APIs REST
- Imágenes

Conversión entre objetos y JSON

- Cuando se implementa una API REST es deseable controlar cómo se **convierten los objetos a JSON** (y viceversa)
- Spring utiliza la librería **Jackson** en modo **data binding** para hacer esta tarea
- Existen **diferentes formas** de controlar la conversión, pero la más sencilla es usando **anotaciones**

<https://github.com/FasterXML/jackson>

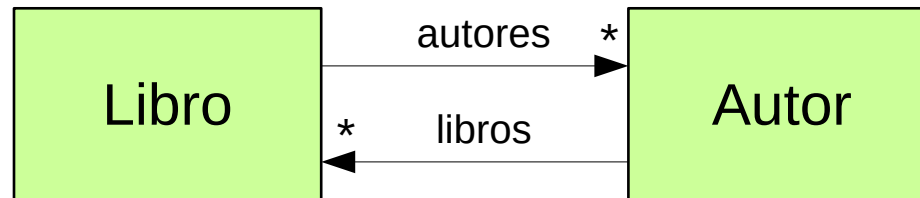
<https://www.baeldung.com/jackson>

Conversión entre objetos y JSON

- **NO es una buena práctica** usar los objetos del modelo directamente en la API REST
- Es recomendable usar el patrón **DTO** (*Data Transfer Object*)
 - Hay veces que con una clase es suficiente **PostDTO**
 - Otras veces es necesario crear clases específicas por cada endpoint (**PostCreateRequest**, **PostReplaceRequest**, **PostResource...**)
- Estas estrategias se estudiarán más adelante

Conversión entre objetos y JSON

- Modelo para gestionar libros y autores



```
public class Libro {  
  
    private long id;  
    private String titulo;  
    private int precio;  
  
    private List<Autor> autores;  
}
```

```
public class Autor {  
  
    private long id;  
    private String nombre;  
    private String nacionalidad;  
  
    private List<Libro> libros;  
}
```

Conversión entre objetos y JSON

- ¿Qué ocurre si tenemos esta API REST?

```
@RestController
public class LibrosAutoresController {

    private List<Libro> libros = ...

    @GetMapping("/libros")
    public List<Libro> getLibros() {
        return libros;
    }
}
```

Conversión entre objetos y JSON

- ¿Qué ocurre si tenemos esta API REST?

```
@RestController
public class LibrosAutoresController {

    private List<Libro> libros = ...

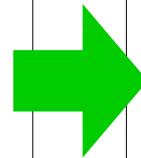
    @GetMapping("/libros")
    public List<Libro> getLibros() {
        return libros;
    }
}
```

Al intentar convertir los objetos a JSON, se produce un **ERROR por StackOverflow** (o similar).
Como un libro tiene un autor y un autor tiene también un libro existe una referencia circular.

Conversión entre objetos y JSON

- Ignorando atributos circulares
 - Se pueden ignorar del JSON los atributos de la clases que generan la referencia circular

```
public class Libro {  
  
    private long id;  
    private String titulo;  
    private int precio;  
  
    @JsonIgnore  
    private List<Autor> autores;  
}
```



```
[  
  {  
    "id":0,  
    "titulo":"Bambi",  
    "precio":3  
  },  
  {  
    "id":1,  
    "titulo":"Batman",  
    "precio":4  
  }  
]
```

- **Datos diferentes por URL**
 - El problema es que esta solución impide obtener la **lista de autores** asociada a un **libro**
 - Lo ideal sería tener **más o menos información** en función de si estamos accediendo a la **lista** de libros o a un libro **concreto**
 - **Lista de libros:** Sin autores
 - **Libro concreto:** Información del libro e información de sus autores (pero sin todos sus libros)

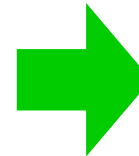
Conversión entre objetos y JSON

- **Datos diferentes por URL**
 - Creamos un nuevo **interfaz** Java
 - Anotamos algunos atributos con **@JsonView** pasando ese **interfaz** como parámetro
 - Anotamos el método de **@RestController** igual que los atributos (**@JsonView** con el **interfaz** como parámetro)
 - Los objetos que devuelva el método tendrán únicamente los **atributos con ese interfaz**

Conversión entre objetos y JSON

```
public class Autor {  
  
    interface Basico {}  
  
    @JsonView(Basico.class)  
    private long id = -1;  
  
    @JsonView(Basico.class)  
    private String nombre;  
  
    @JsonView(Basico.class)  
    private String nacionalidad;  
  
    private List<Libro> libros;  
}
```

```
@JsonView(Autor.Basico.class)  
@GetMapping("/autores")  
public List<Autor> getAutores() {  
    return autores;  
}
```



```
[  
  {  
    "id":0,  
    "titulo":"Bambi",  
    "precio":3  
  },  
  {  
    "id":1,  
    "titulo":"Batman",  
    "precio":4  
  }  
]
```


Conversión entre objetos y JSON

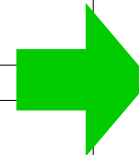
- **Datos diferentes por URL**
 - Si queremos que en un **método** de la **API REST** se devuelvan atributos anotados con diferentes interfaces hay que **crear un nuevo interfaz**
 - Ese **nuevo interfaz** tiene que **heredar** de los interfaces usados por los atributos
 - Usamos ese interfaz en el **@JsonView** del método del **@RestController**

Conversión entre objetos y JSON

ejem6

```
public class Autor {  
  
    interface Basico {}  
    interface Libros {}  
  
    @JsonView(Basico.class)  
    private long id = -1;  
    ...  
    @JsonView(Libros.class)  
    private List<Libro> libros;  
}
```

```
interface AutorDetalle  
    extends Autor.Basico, Autor.Libros,  
            Libro.Basico {}  
  
@JsonView(AutorDetalle.class)  
@GetMapping("/autores/{id}")  
public Autor getAutor(@PathVariable int id){  
    return autores.get(id);  
}
```



```
{  
  "id":1,  
  "nombre":"Gerard",  
  "nacionalidad":"Frances",  
  "libros":[  
    {  
      "id":1,  
      "titulo":"Batman",  
      "precio":4  
    },  
    {  
      "id":2,  
      "titulo":"Spiderman",  
      "precio":2  
    }  
  ]  
}
```

APIs REST con Spring

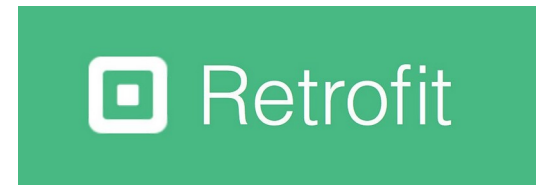
- Introducción
- Clientes de APIs REST
- APIs REST con Spring
- Conversión entre objetos y JSON
- **Cliente REST en el servidor**
- Documentación de APIs REST
- Imágenes

Cientes REST en el servidor

- Ya hemos visto cómo hacer peticiones a un servicio REST desde el **navegador** con **jQuery**
- Un **servidor web** también se puede convertir en **cliente de APIs REST** de otros servicios:
 - Ejemplo: Redes sociales, información meteorológica, libros, fotos, vídeos...

Cientes REST en el servidor

- **Cientes REST Java**
 - Java HttpClient
 - Jersey Client
 - Apache HttpClient
 - **RestTemplate de Spring**
 - Retrofit
 - Feign



Cientes REST en el servidor

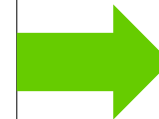
- **RestTemplate de Spring**
 - Para hacer peticiones REST en **Spring** se usa un objeto de la clase **RestTemplate**
 - Se indica la **URL** y la **clase** de los objetos que devolverá la consulta

Cientes REST en el servidor

ejem7

- **RestTemplate de Spring**
 - Para procesar la respuesta se indica la clase o registro que estructura los datos

```
{
  "kind": "books#volumes",
  "totalItems": 579,
  "items": [
    {
      "kind": "books#volume",
      "volumeInfo": {
        "title": "Java a Tope: J2me...",
        "publisher": "Sergio Gálvez Rojas",
        ...
      },
      {...},
      {...}
    ]
    ...
  }
```



```
record BooksResponse(
  List<Book> items
){}

record Book(
  VolumeInfo volumeInfo
){}

record VolumeInfo(
  String title
){}
```

Cientes REST en el servidor

ejem7

- RestTemplate de Spring

```
RestTemplate restTemplate = new RestTemplate();

String url="https://www.googleapis.com/.../volumes?q=intitle:"+title;

BooksResponse data =
    restTemplate.getForObject(url, BooksResponse.class);

List<String> bookTitles = new ArrayList<String>();

for (Book book : data.items()) {
    bookTitles.add(book.volumeInfo().title());
}

return bookTitles;
```


Cientes REST en el servidor

ejem8

- **RestTemplate: Cliente REST Spring**
 - Es posible acceder a los datos directamente **sin definir clases**
 - Para ello se usa la librería **Jackson** de procesamiento de **JSON** para Java
 - **Jackson** también se usa para convertir (mapear) el **JSON** a objetos (vistos del ejemplo anterior)

<https://github.com/FasterXML/jackson>

<https://www.baeldung.com/jackson>

Cientes REST en el servidor

ejem8

- **RestTemplate: Cliente REST Spring**

```
RestTemplate restTemplate = new RestTemplate();

String url="https://www.googleapis.com/.../volumes?q=intitle:"+title;

ObjectNode data = restTemplate.getForObject(url, ObjectNode.class);

List<String> bookTitles = new ArrayList<String>();

ArrayNode items = (ArrayNode) data.get("items");

for (int i = 0; i < items.size(); i++) {
    JsonNode item = items.get(i);
    String bookTitle = item.get("volumeInfo").get("title").asText();
    bookTitles.add(bookTitle);
}
```

Cientes REST en el servidor

ejem8

- RestTemplate: Cliente REST Spring

```
RestTemplate restTemplate = new RestTemplate();

String url="https://www.googleapis.com/.../volumes?q=intitle:"+title;

ObjectNode data = restTemplate.getForObject(url, ObjectNode.class);

List<String> bookTitles = new ArrayList<String>();

ArrayNode items = (ArrayNode) data.get("items");

for (int i = 0; i < items.size(); i++) {
    JsonNode item = items.get(i);
    String bookTitle = item.get("volumeInfo").get("title").asText();
    bookTitles.add(bookTitle);
}
```

Cientes REST en el servidor

- **Spring Feign**

- Definición **declarativa** de una API REST que va a ser consumida
 - Se define un interfaz Java con un método por cada endpoint REST (con anotaciones)
 - Spring genera una implementación que realiza las consultas usando un cliente REST
 - Un componente puede inyectar el interfaz y al invocar los métodos se ejecutan las consultas
 - Se inyecta como una dependencia más

Cientes REST en el servidor

ejem9

- Consumo de la API REST de libros

```
@FeignClient(name = "books", url = "https://www.googleapis.com/")
public interface BooksService {

    public record BooksResponse(List<Book> items) {
    }

    public record Book(VolumeInfo volumeInfo) {
    }

    public record VolumeInfo(String title) {
    }

    @GetMapping("books/v1/volumes")
    BooksResponse getBooks(@RequestParam String q);
}
```

Anotamos la interfaz con la url del servidor rest

Podemos definir la estructura de la respuesta como registros en el propio servicio

Definimos los métodos como en un controlador

Cientes REST en el servidor

ejem9

- Consumo de la API REST de libros

```
@RestController
public class BooksController {

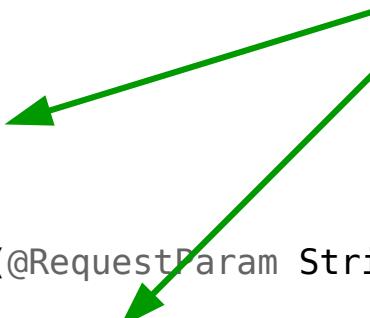
    @Autowired BooksService service;

    @GetMapping("/booktitles")
    public List<String> getBookTitles(@RequestParam String title) {

        BooksResponse data = service.getBooks("intitle:" + title);

        List<String> bookTitles = new ArrayList<String>();
        for (Book book : data.items()) {
            bookTitles.add(book.volumeInfo().title());
        }
        return bookTitles;
    }
}
```

Injectamos la interfaz y la usamos para llamar al servidor rest



Cientes REST en el servidor

- Consumo de la API REST de anuncios

```
@SpringBootApplication
@EnableFeignClients
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Anotamos la aplicación para que Spring genere automáticamente el cliente feign

Cientes REST en el servidor

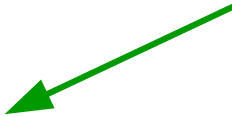
ejem9

- Dependencias

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>2022.0.0-RC1</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

La dependencia está en Spring Cloud, no en Spring Boot



APIs REST con Spring

- Introducción
- Clientes de APIs REST
- APIs REST con Spring
- Conversión entre objetos y JSON
- Cliente REST en el servidor
- **Documentación de APIs REST**
- Imágenes

Documentación de APIs REST

- El mecanismo más usado para documentar APIs REST es la especificación **OpenAPI**
- Es una evolución de **Swagger Specification**
- Tiene un ecosistema de herramientas: **generación de código**, webs interactivas, etc...

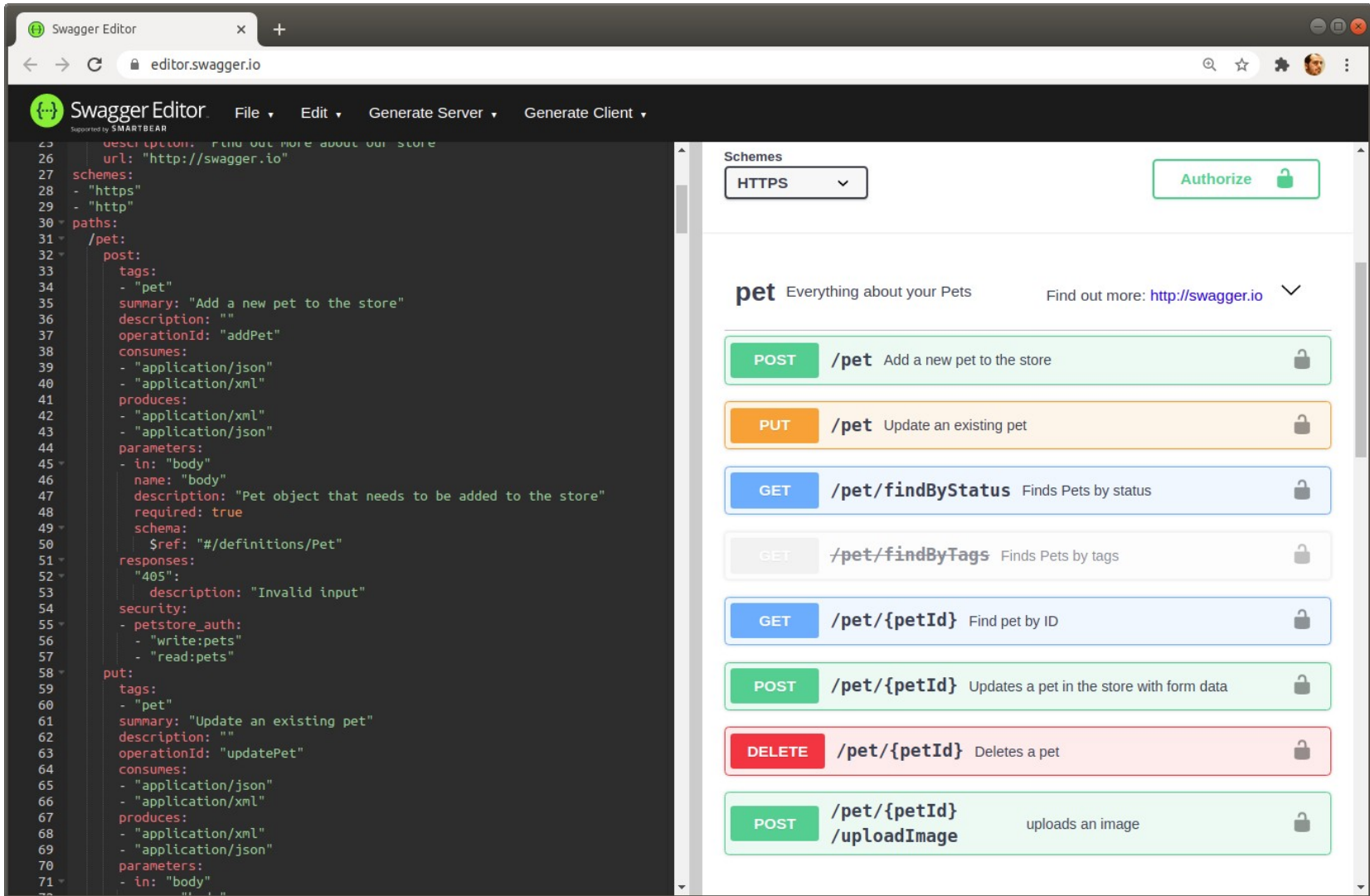


<https://www.openapis.org/>

Documentación de APIs REST

- **Especificación OpenAPI**
 - Se puede especificar en YAML o en JSON
 - Describe cada operación (*endpoint*)
 - Formato de URL
 - Formato requerido en el cuerpo de la petición
 - Formato de los resultados
 - Se puede editar usando el **Swagger Editor**, un editor interactivo con “previsualización en tiempo real”

<https://editor.swagger.io/>



The screenshot shows the Swagger Editor interface in a web browser. The left pane displays the OpenAPI JSON definition for a pet store API. The right pane shows the rendered API documentation, including a list of endpoints with their HTTP methods, paths, and descriptions.

Swagger Editor JSON Definition (Left Pane):

```

25 description: Find out more about our store
26 url: "http://swagger.io"
27 schemes:
28 - "https"
29 - "http"
30 paths:
31 /pet:
32 post:
33 tags:
34 - "pet"
35 summary: "Add a new pet to the store"
36 description: ""
37 operationId: "addPet"
38 consumes:
39 - "application/json"
40 - "application/xml"
41 produces:
42 - "application/xml"
43 - "application/json"
44 parameters:
45 - in: "body"
46 name: "body"
47 description: "Pet object that needs to be added to the store"
48 required: true
49 schema:
50 $ref: "#/definitions/Pet"
51 responses:
52 "405":
53 description: "Invalid input"
54 security:
55 - petstore_auth:
56 - "write:pets"
57 - "read:pets"
58 put:
59 tags:
60 - "pet"
61 summary: "Update an existing pet"
62 description: ""
63 operationId: "updatePet"
64 consumes:
65 - "application/json"
66 - "application/xml"
67 produces:
68 - "application/xml"
69 - "application/json"
70 parameters:
71 - in: "body"
  
```

Rendered API Documentation (Right Pane):

Schemes: HTTPS (dropdown) **Authorize** (button)

pet Everything about your Pets Find out more: <http://swagger.io>

- POST** /pet Add a new pet to the store
- PUT** /pet Update an existing pet
- GET** /pet/findByStatus Finds Pets by status
- GET** /pet/findByTags Finds Pets by tags
- GET** /pet/{petId} Find pet by ID
- POST** /pet/{petId} Updates a pet in the store with form data
- DELETE** /pet/{petId} Deletes a pet
- POST** /pet/{petId}/uploadImage uploads an image

Documentación de APIs REST

- **Formato OpenAPI 3**

```
openapi: 3.0.0
info:
  version: 1.0.0
  title: Sample API
  description: A sample API to illustrate OpenAPI concepts
paths:
  /list:
    get:
      description: Returns a list of stuff
      responses:
        '200':
          description: Successful response
```

<https://app.swaggerhub.com/help/tutorials/openapi-3-tutorial>

Documentación de APIs REST

- **Formato OpenAPI 3**
 - Los ficheros se dividen en 3 partes
 - **Meta Information**
 - **Path Items (endpoints)**
 - Parameters, Request bodies, Responses
 - **Reusable Components**
 - Schemas (data models), Parameters, Responses, Other components

Documentación de APIs REST

- **Formato OpenAPI 3**
- Meta Information

```

openapi: 3.0.0
info:
  version: 1.0.0
  title: Simple Artist API
  description: A simple API to illustrate OpenAPI concepts

servers:
  - url: https://example.io/v1

# Basic authentication
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
security:
  - BasicAuth: []

paths: {}
  
```

Documentación de APIs REST

- Formato OpenAPI 3
- Path Items

```

openapi: 3.0.0
info:
  version: 1.0.0
  title: Simple API
  description: A simple API to illustrate OpenAPI concepts

servers:
  - url: https://example.io/v1

components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
security:
  - BasicAuth: []

# ----- Added lines -----
paths:
  /artists:
    get:
      description: Returns a list of artists
# ---- /Added lines -----

```



```
openapi: 3.0.0
info:
  ...
paths:
  /artists:
    get:
      description: Returns a list of artists
      # ----- Added lines -----
      responses:
        '200':
          description: Successfully returned a list of artists
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
                  required:
                    - username
                  properties:
                    artist_name:
                      type: string
                    artist_genre:
                      type: string
                    albums_recorded:
                      type: integer
                    username:
                      type: string
        '400':
          description: Invalid request
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string
      # ----- /Added lines -----
```

```
openapi: 3.0.0
info:
...
paths:
  /artists:
    get:
      description: Returns a list of artists
      # ----- Added lines -----
      parameters:
        - name: limit
          in: query
          description: Limits the number of items on a page
          schema:
            type: integer
        - name: offset
          in: query
          description: Specifies the page number of the artists to be displayed
          schema:
            type: integer
      # ---- /Added lines -----
      responses:
        '200':
          description: Successfully returned a list of artists
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
                  required:
                    - username
                  properties:
                    artist_name:
                      type: string
                    artist_genre:
                      type: string
                    albums_recorded:
                      type: integer
                    username:
                      type: string
...
...
```

Documentación de APIs REST

- **Formato OpenAPI 3**
 - Reusable Components
 - Schemas (data models)
 - Parameters
 - Request bodies
 - Responses
 - Response headers
 - Examples
 - Links
 - Callbacks

Documentación de APIs REST

- **Enfoque *API First***
 - Es muy importante que una API sea entendible por sus clientes
 - Al igual que se diseñan interfaces gráficas de usuario con los propios usuarios antes de su implementación, también es conveniente diseñar las APIs con los futuros usuarios
 - OpenAPI se utiliza para diseñar APIs

<https://swagger.io/resources/articles/adopting-an-api-first-approach/>

Documentación de APIs REST

- **Beneficios del enfoque *API First***
 - Es mucho más sencillo de obtener feedback de los clientes
 - Es más fácil que sea coherente y cumpla reglas de estilo (al evitar detalles de implementación)
 - Equipos pueden trabajar en paralelo (front y back) una vez definida la API
 - Se pueden usar herramientas de generación de código

<https://swagger.io/resources/articles/adopting-an-api-first-approach/>

Documentación de APIs REST

- **Generación de OpenAPI partiendo de código**
 - SpringDoc es un proyecto que genera la especificación OpenAPI partiendo de una API REST en SpringBoot



OpenAPI 3
&
Spring Boot

<https://springdoc.org/>

<https://www.baeldung.com/spring-rest-openapi-documentation>

Documentación de APIs REST

ejem10

- **Generación de OpenAPI partiendo de código**
 - Si añadimos la dependencia Maven de SpringDoc

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.0.0-RC1</version>
</dependency>
```

- La especificación de la API se puede descargar de

<http://localhost:8080/v3/api-docs/>

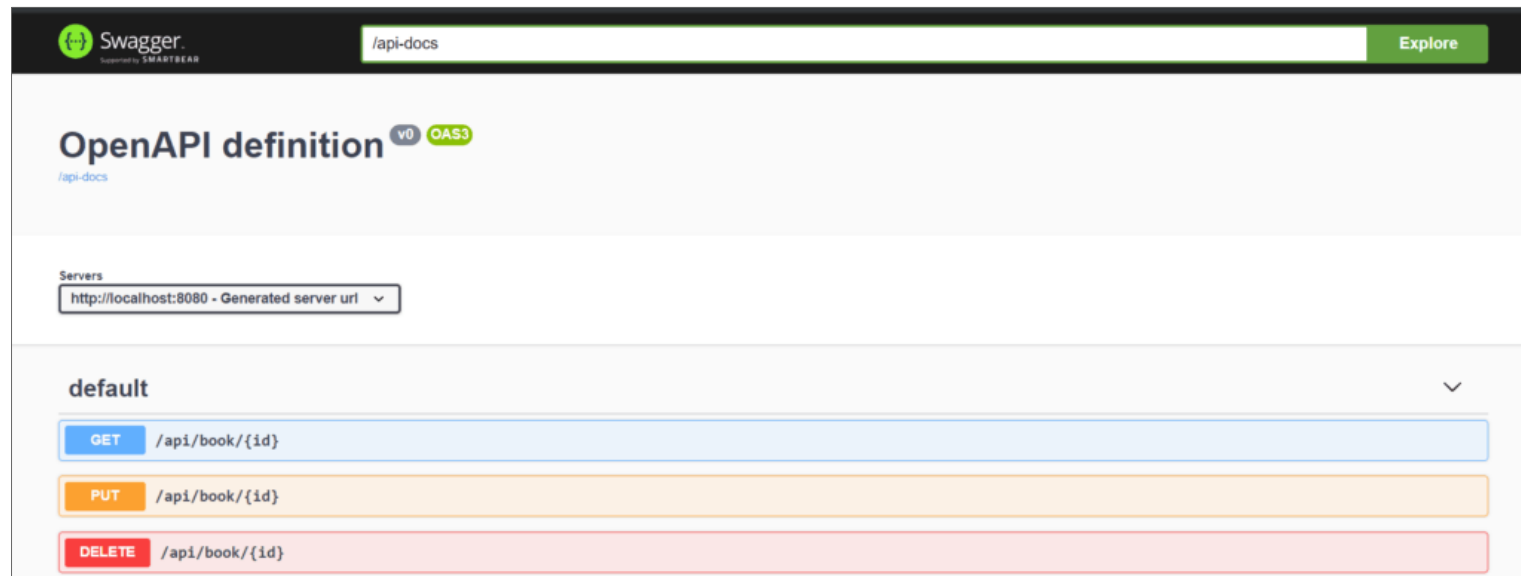
<http://localhost:8080/v3/api-docs.yaml>

Documentación de APIs REST

ejem10

- **Generación de OpenAPI partiendo de código**
 - Se publica un interfaz web interactivo que documenta la API y permite interactuar con ella

<http://localhost:8080/swagger-ui.html>



Documentación de APIs REST

ejem10

- **Generación de OpenAPI partiendo de código**
 - Se pueden añadir anotaciones específicas para documentar mejor la API
 - Usar la anotación **@ResponseStatus** en endpoints
 - Usar anotaciones **@Operation** y **@ApiResponses** proporcionadas por SpringDoc

```
@Operation(summary = "Get a book by its id")

@ApiResponses(value = {
    @ApiResponse(
        responseCode = "200",
        description = "Found the book",
        content = {@Content(
            mediaType = "application/json",
            schema = @Schema(implementation=Book.class)
        )}
    ),
    @ApiResponse(
        responseCode = "400",
        description = "Invalid id supplied",
        content = @Content
    ),
    @ApiResponse(
        responseCode = "404",
        description = "Book not found",
        content = @Content
    )
})

@GetMapping("/{id}")
public Book findById(
    @Parameter(description="id of book to be searched")
    @PathVariable long id) {

    return ...;
}
```



GET /api/book/{id} Get a book by its id	
Parameters	
Name	Description
id <small>* required</small> integer (path)	id of book to be searched
<input type="text" value="id - id of book to be searched"/>	
Responses	
Code	Description
200	Found the book
Media type <input type="text" value="application/json"/>	
Controls Accept header.	
Example Value Schema	
<pre>{ "id": 0, "title": "string", "author": "string" }</pre>	
400	Invalid id supplied
404	Book not found

APIs REST con Spring

- Introducción
- Clientes de APIs REST
- APIs REST con Spring
- Conversión entre objetos y JSON
- Cliente REST en el servidor
- Documentación de APIs REST
- **Imágenes**

Imágenes

- Existen **diversas estrategias** para gestionar imágenes en una API REST
- Vamos a considerar que cada entidad solo puede tener **una única imagen (opcional)**
- La imagen se podrán descargar en una URL que estará guardada como atributo del recurso

```
{
  "id": 1,
  "user": "Pepe",
  "title": "Vendo moto",
  "text": "Barata, barata",
  "image": "http://server/posts/1/image"
}
```

Imágenes

ejem11

- Upload image

```
@PostMapping("/{id}/image")
public ResponseEntity<Object> uploadImage(@PathVariable long id,
    @RequestParam MultipartFile imageFile) throws IOException {

    Post post = posts.findById(id);

    if (post != null) {

        URI location = fromCurrentRequest().build().toUri();

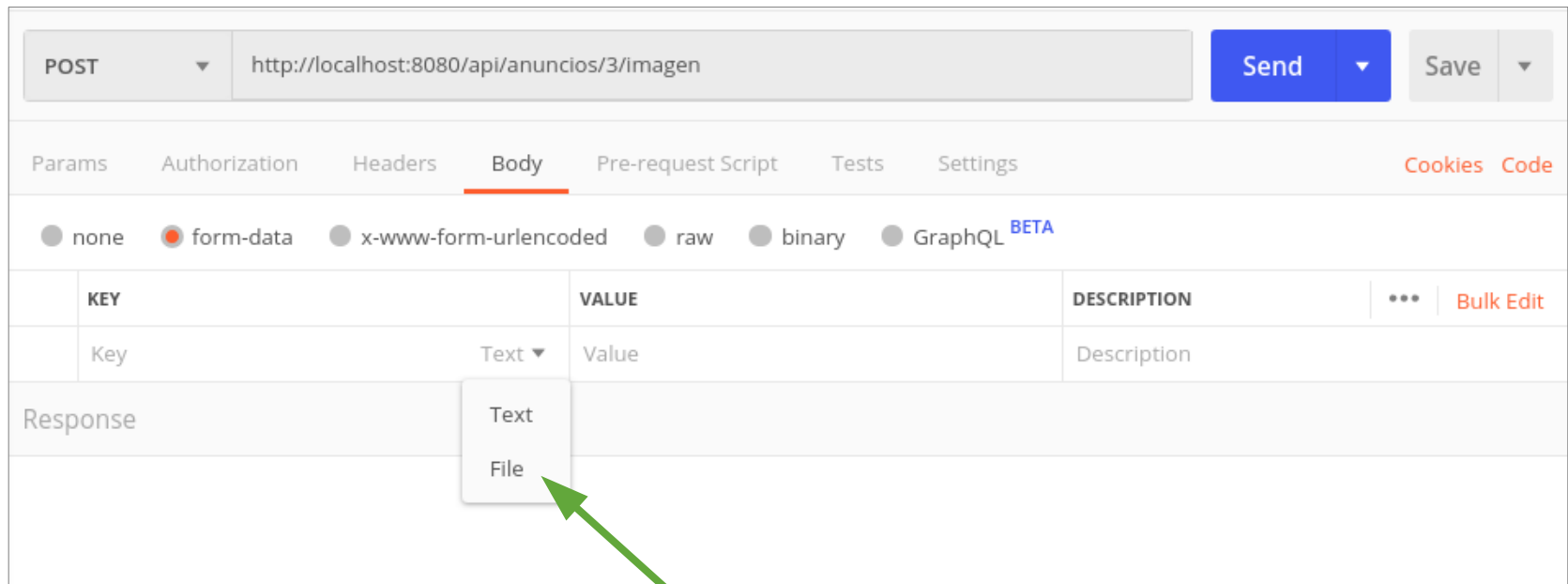
        post.setImage(location.toString());
        posts.save(post);

        imgService.saveImage(POSTS_FOLDER, post.getId(), imageFile);

        return ResponseEntity.created(location).build();

    } else {
        return ResponseEntity.notFound().build();
    }
}
```

- Subir un fichero con Postman



- Subir un fichero con Postman

The screenshot shows the Postman interface for a POST request to `http://localhost:8080/api/anuncios/3/imagen`. The 'Body' tab is selected, and the 'form-data' radio button is chosen. A table lists the form data with one entry: 'imageFile' with a 'File' type and a 'Select Files' button. Two green arrows point to the 'imageFile' key and the 'Select Files' button. The 'Response' section is empty.

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> imageFile	Select Files	
Key	Value	Description

Response

Imágenes

- Download image

```
@GetMapping("/{id}/image")
public ResponseEntity<Object> downloadImage(@PathVariable long id)
    throws MalformedURLException {

    return this.imgService.createResponseFromImage(POSTS_FOLDER, id);
}
```


Imágenes

- Delete image

```
@DeleteMapping("/{id}/image")
public ResponseEntity<Object> deleteImage(@PathVariable long id)
    throws IOException {

    Post post = posts.findById(id);

    if(post != null) {

        post.setImage(null);
        posts.save(post);

        this.imgService.deleteImage(POSTS_FOLDER, id);

        return ResponseEntity.noContent().build();

    } else {
        return ResponseEntity.notFound().build();
    }
}
```