

Curso de Spring  
Framework

# APIs REST con Spring

**Micael Gallego**

micael.gallego@gmail.com

@micael\_gallego

**Francisco Gortázar**

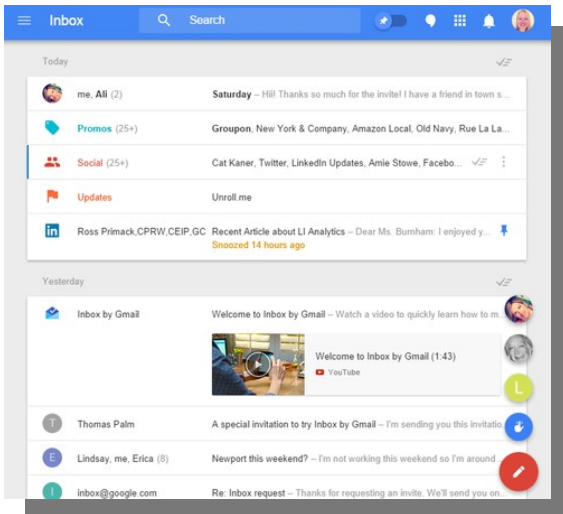
patxi.gortazar@gmail.com

@fgortazar

- **Introducción**
- Formato JSON
- Funcionamiento de un servicio REST
- Clientes de servicios REST
- APIs REST con Spring
- Conversión entre objetos y JSON
- Cliente REST en el servidor

- En una aplicación web, el cliente (**navegador**) se comunica con el servidor (**servidor web**) usando el protocolo **http**
- En una aplicación web sin AJAX, 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 utilizan para intercambiar **información** entre el navegador y el servidor (pero no HTML)

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

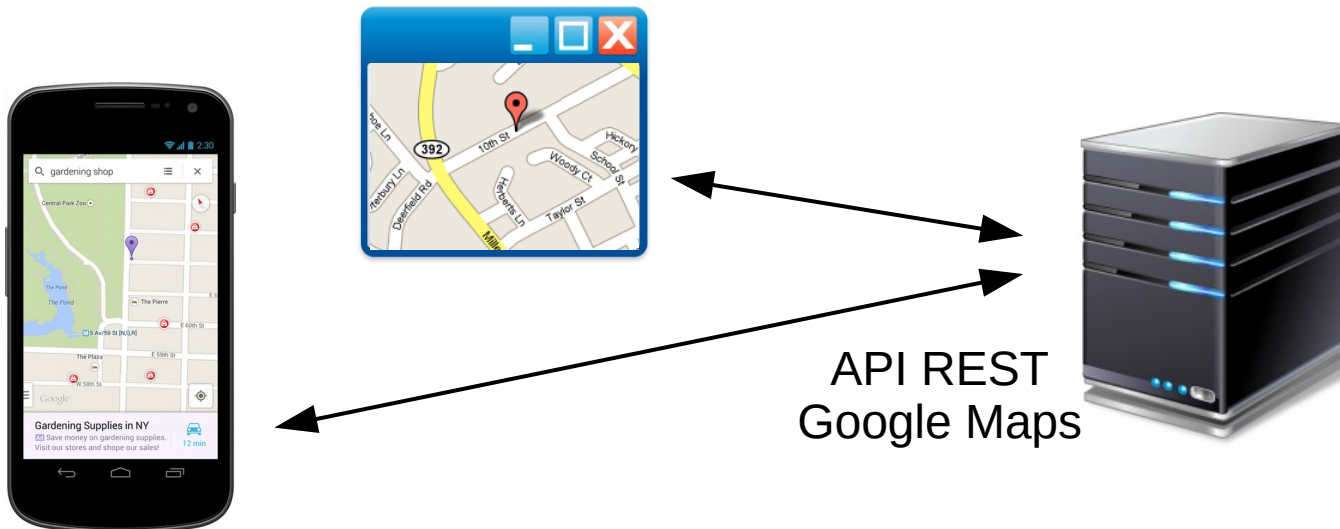


`http://www.miweb.com/users/34`

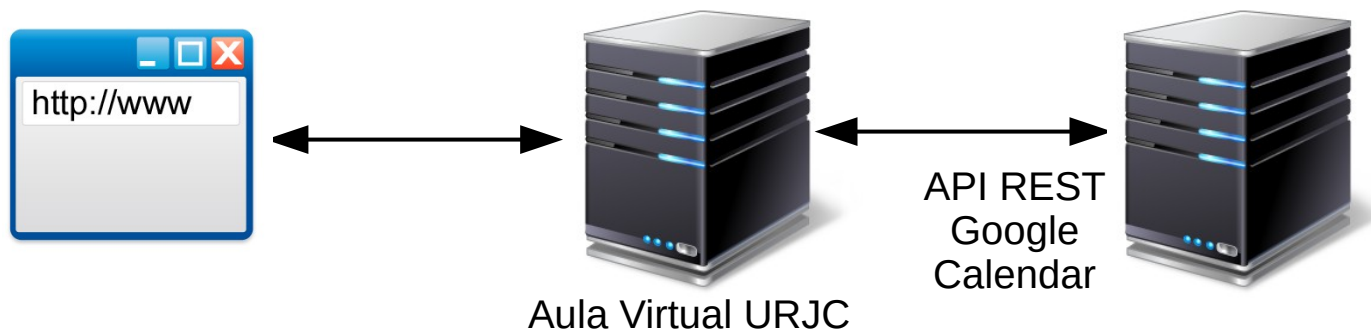
```
{  
  "name": "Pepe",  
  "surname": "López",  
  "age": 45,  
  "email": "pepe@miweb.com",  
  "friends": [ "María", "Juan" ]  
}
```



- 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



- 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

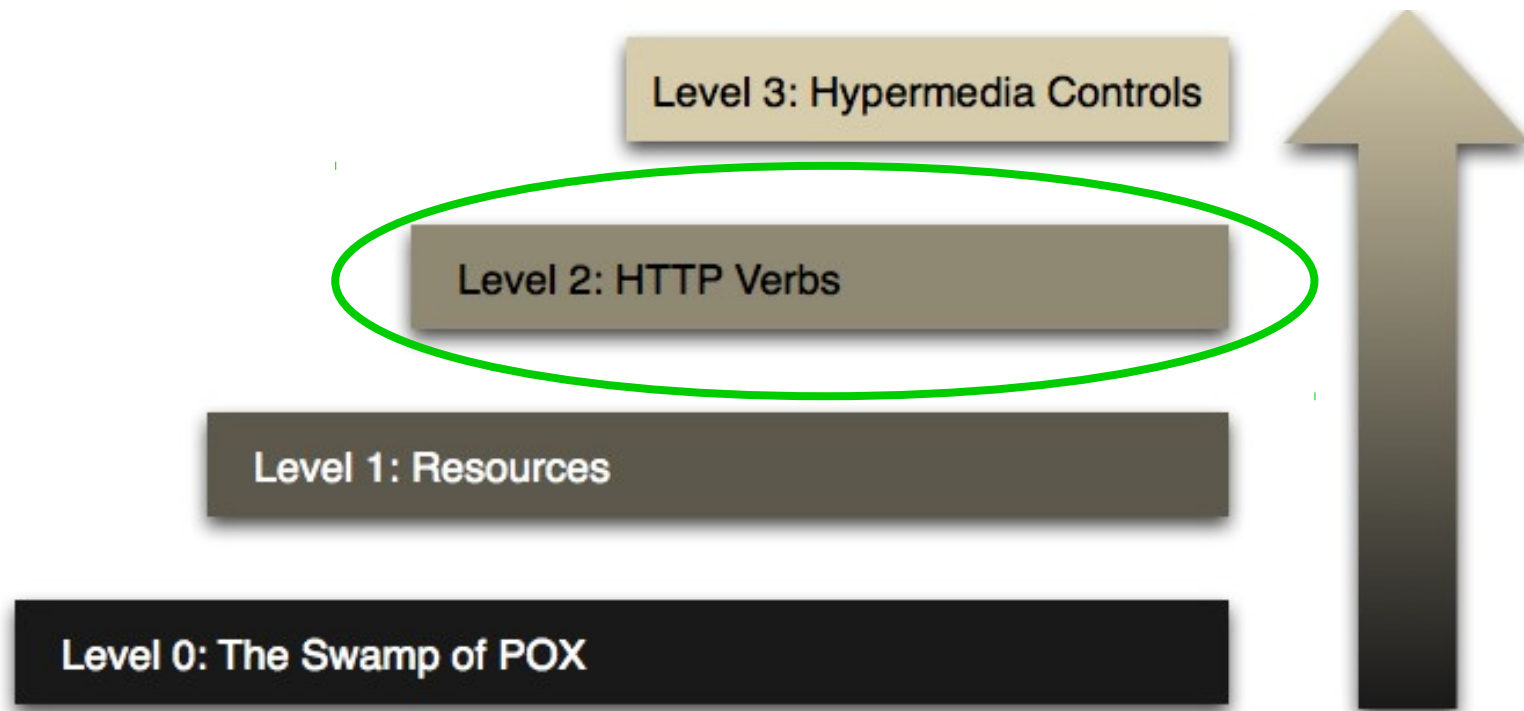


- REST es acrónimo de ***REpresentational State Transfer***, Transferencia de Estado Representacional.
- El término se acuñó en el año 2000, en la tesis doctoral de **Roy Fielding**, uno de los principales autores de la especificación del protocolo **HTTP**
- Existen otros tipos de **servicios web** como **SOAP** basados en **XML** y mucho **más complejos**, pero no se usan tanto como los servicios web REST

# INTRODUCCIÓN

## Servicios web tipo REST

- Niveles de cumplimiento de los principios REST



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



- Un **servicio REST** ofrece operaciones **CRUD** (**creación, lectura, actualización y borrado**) sobre recursos (items de información) del servidor web
- Se aprovecha de todos los aspectos del **protocolo http**: URL, métodos, códigos de estado, cabeceras...
- La información se intercambia en formato **JSON** (o XML)

- Introducción
- **Formato JSON**
- Funcionamiento de un servicio REST
- Clientes de servicios REST
- APIs REST con Spring
- Conversión entre objetos y JSON
- Cliente REST en el servidor

- Acrónimo de *JavaScript Object Notation*
- Es un subconjunto de la notación literal de objetos de **JavaScript**
- Se procesa de forma muy rápida en **JavaScript**



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

## Información estructurada con JSON

```
{ "menu": {  
  "id": 1,  
  "value": "File",  
  "enabled": true,  
  "popup": {  
    "menuitem": [  
      { "value": "New", "onclick": "CreateNewDoc()" },  
      { "value": "Open", "onclick": "OpenDoc()" },  
      { "value": "Close", "onclick": "CloseDoc()" }  
    ]  
  }  
}}
```

- **JSON** se utiliza para la codificación de la información en la mayoría de los servicios REST(aunque también se puede usar **XML**).
- También se usa para **estructurar** cualquier tipo de información:
  - **Ficheros de configuración**
  - Datos en disco
  - Bases de datos NoSQL (Mongo)

- Introducción
- Formato JSON
- **Funcionamiento de un servicio REST**
- Clientes de servicios REST
- APIs REST con Spring
- Conversión entre objetos y JSON
- Cliente REST en el servidor

- El enfoque más habitual en los servicios REST es el nivel 2:
  - Los recursos se identifican en la **URI**  
<http://server/anuncio/vendo-moto-23-10-2014>
  - Las **operaciones** que se quieren realizar con ese recurso son los **métodos del protocolo HTTP**
  - La información se devuelve codificada en **JSON**
  - Se usan los códigos de **estado http** para notificar errores (p.e. 404 Not found)

- Los recursos se identifican en la URI
  - Parte de la URL es fija y otra parte apunta al recurso concreto
  - Ejemplos:
    - <http://server/anuncios/vendo-moto-23-10-2014>
    - <http://server/users/bob>
    - <http://server/users/bob/anuncio/comparto-piso>
    - <http://server/users/bob/anuncio/44>

<http://blog.2partsmagic.com/restful-uri-design/>



- **Las operaciones se codifican como métodos http**
  - **GET:** Devuelve el recurso, generalmente codificado en JSON. No envían información en el cuerpo de la petición.
  - **DELETE:** Borra el recurso. No envían información en el cuerpo de la petición.
  - **POST:** Añade un nuevo recurso. Envía el recurso en el cuerpo de la petición.
  - **PUT:** Modifica el recurso. Habitualmente se envía el recurso obtenido con GET pero modificando los campos que se consideren (existen optimizaciones)

- La información se devuelve codificada en JSON

- Petición:

- URL:** <http://server/bob/bookmarks/6>

- Método:** GET

- Respuesta:

- mime-type:** application/json

- Body:**

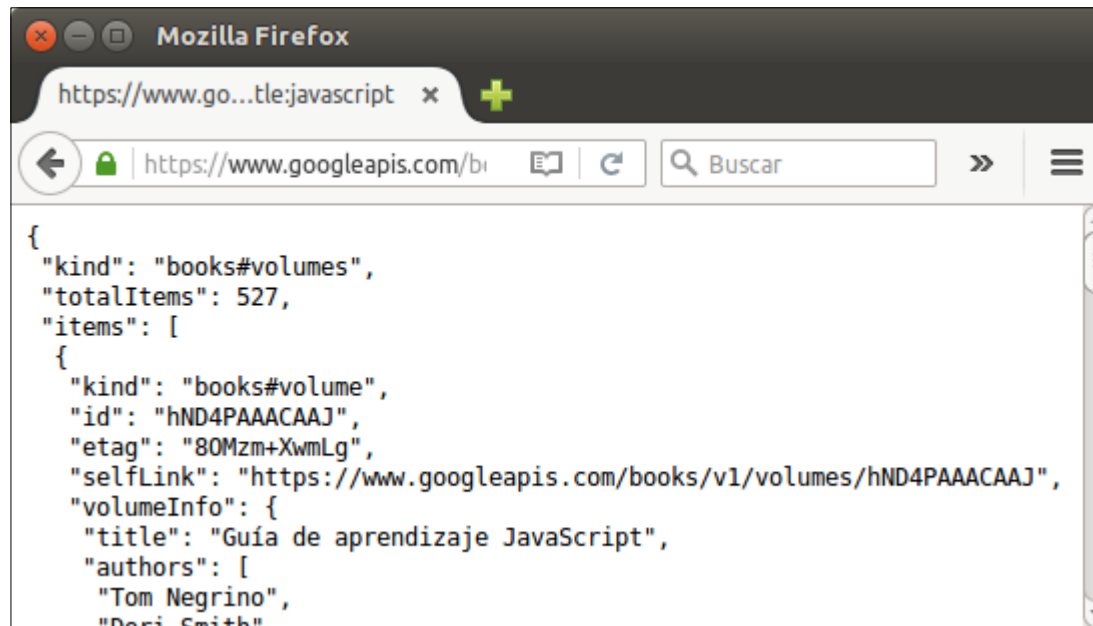
```
{  
  id: 6,  
  uri: "http://bookmark.com/2/bob",  
  description: "A description"  
}
```

- Se usan los códigos de estado http para notificar errores:
  - **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.

- Introducción
- Formato JSON
- Funcionamiento de un servicio REST
- **Cientes de servicios REST**
- APIs REST con Spring
- Conversión entre objetos y JSON
- Cliente REST en el servidor

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

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

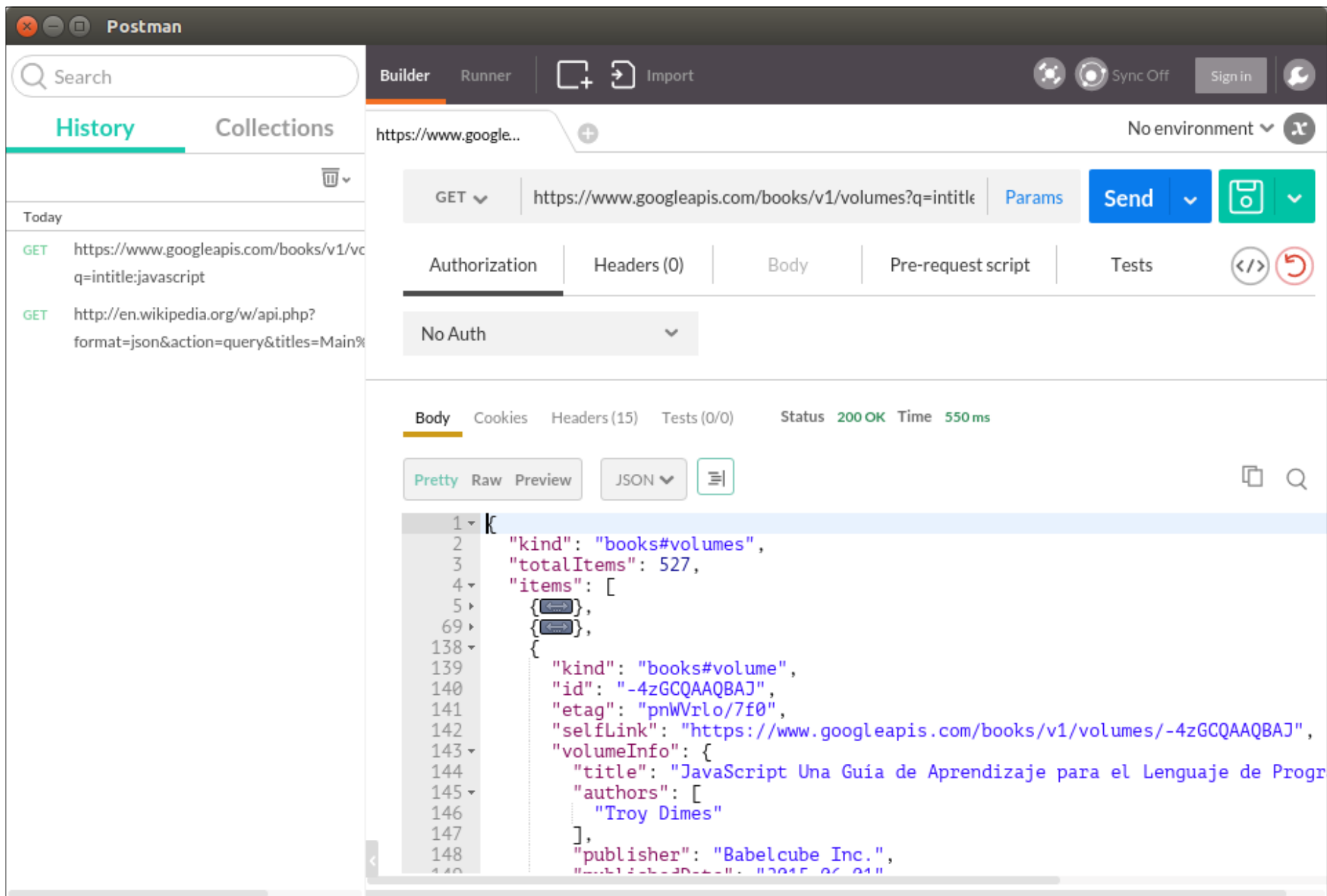
# APIS REST

## Cientes de servicios REST

- Postman - REST Client para Google Chrome

The image shows the Postman website and a screenshot of the Postman application interface. The website header includes the Postman logo, the URL [www.getpostman.com](http://www.getpostman.com), a star rating of 5 stars (5790), a link to [Herramientas para desarrolladores](#), and a user count of 1.643.907 usuarios. A green button labeled "INICIAR APLICACIÓN" is visible. Below the header are tabs for "DESCRIPCIÓN GENERAL", "OPINIONES", "AYUDA", and "RELACIONADOS". The main content area features the text "Working with APIs? We've got you covered." and a screenshot of the Postman application. The application interface shows a "History" sidebar with items like "JSONBiot Core API" and "Postman Echo". The main panel displays a "Request Headers" tab with a GET request to "https://echo.getpostman.com/headers". The status bar at the bottom indicates "Status: 200 OK Time: 1312ms". To the right of the application screenshot, there are two feature highlights: "Funciona sin conexión" (Works offline) and "Compatible con tu dispositivo" (Compatible with your device). Below these, a text block states: "Supercharge your API workflow with Postman! Build, test, and document your APIs faster. More than a million developers already do....". Another text block says: "Supercharge your API workflow with Postman! Build, test, and document your APIs faster. More than a million developers already do....". A final text block mentions: "The idea for Postman arose while the founders were working together, and were frustrated with the existing tools for testing APIs."





- 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>

- **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)

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

script.js

```
$(document).ready(function() {  
  
    $.ajax({  
        url: "https://www.googleapis.com/books/v1/volumes?q=intitle:java"  
    }).done(function(data) {  
        console.log(data);  
    });  
  
});
```

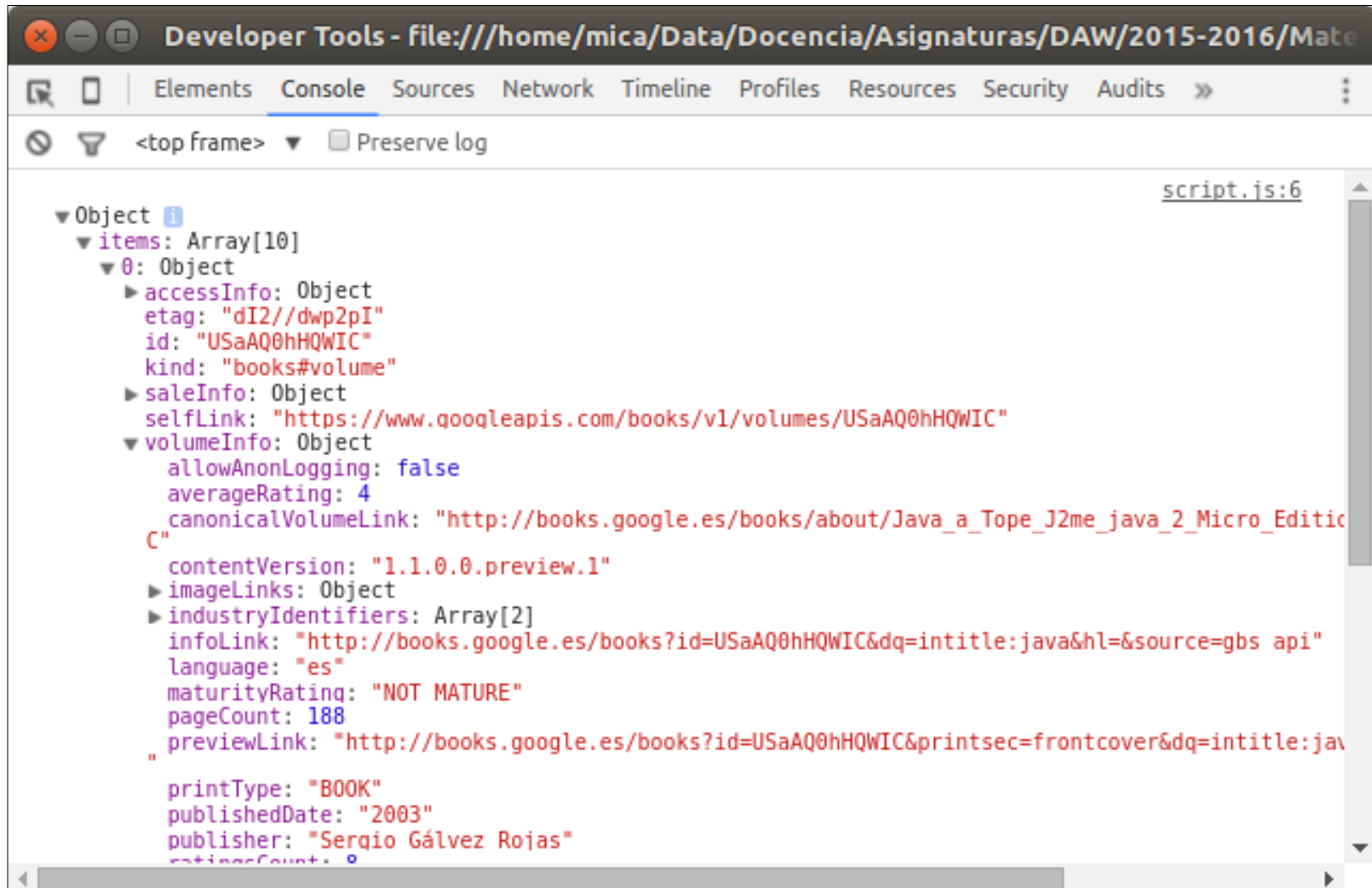
<http://api.jquery.com/jquery.ajax/>

- Cliente JavaScript: jQuery

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://code.jquery.com/jquery-2.2.0.min.js">
    </script>
    <script src="script.js"></script>
  </head>
  <body>
  </body>
</html>
```

# APIS REST

## Clientes de servicios REST



The screenshot shows the 'Console' tab of a web browser's Developer Tools. The address bar indicates the file path: `file:///home/mica/Data/Docencia/Asignaturas/DAW/2015-2016/Mate`. The console displays a log entry for `script.js:6`, showing a JavaScript object with the following structure:

```
Object
  items: Array[10]
    0: Object
      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_Editic"
        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
```

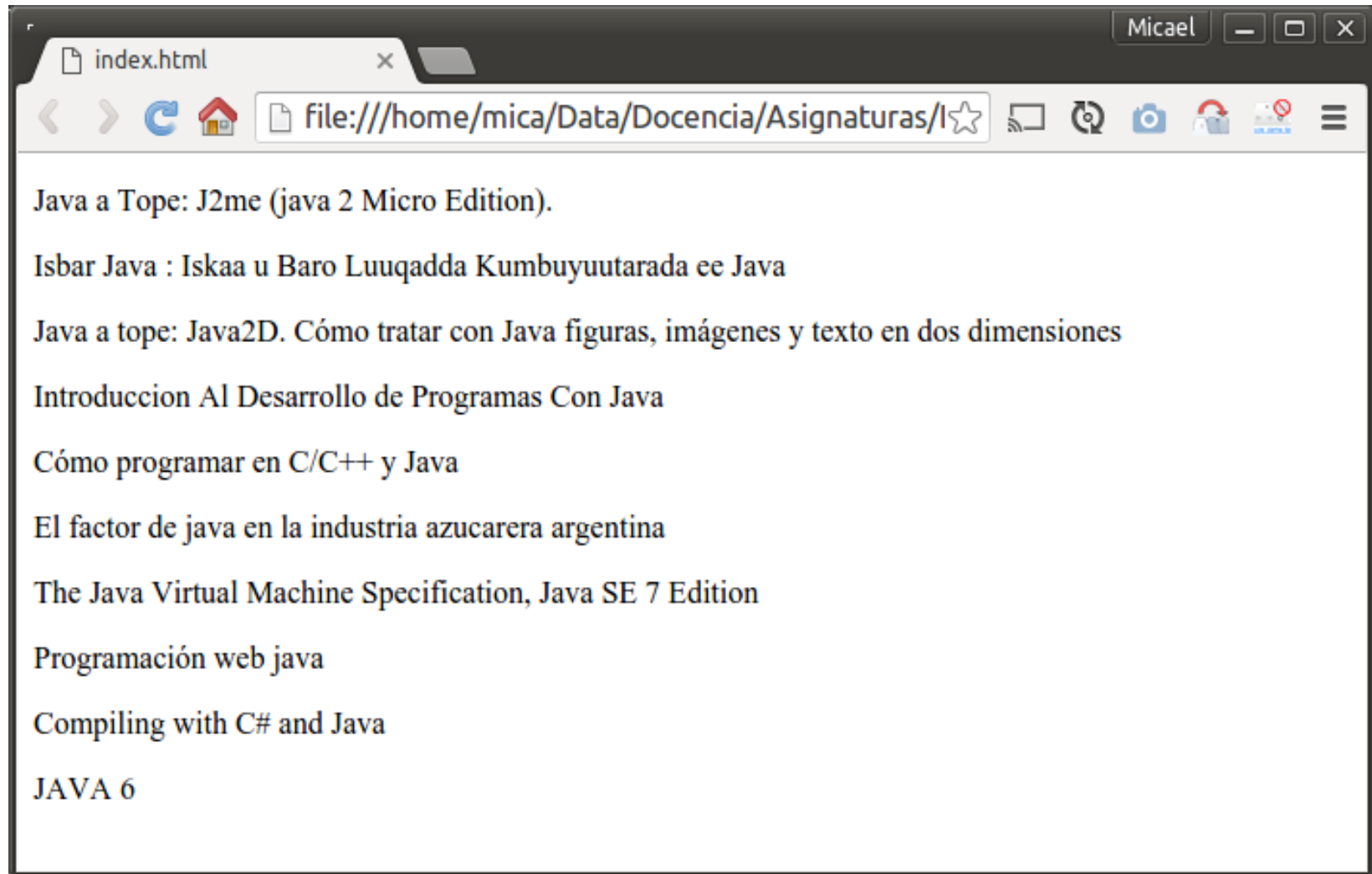
- Cliente JavaScript: jQuery
  - Muestra en la página los títulos de los libros

script.js

```
$(document).ready(function() {  
    $.ajax({  
        url: "https://www.googleapis.com/books/v1/volumes?q=intitle:java"  
    }).done(function(data) {  
        for(var i=0; i<data.items.length; i++){  
            $("body").append(  
                "<p>" + data.items[i].volumeInfo.title + "</p>";  
            }  
        });  
    });  
});
```

# APIS REST

## Cientes de servicios REST





- Introducción
- Formato JSON
- Funcionamiento de un servicio REST
- Clientes de servicios REST
- **APIs REST con Spring**
- Conversión entre objetos y JSON
- Cliente REST en el servidor

- Controlador

Se anota la clase con  
**@RestController**

```
@RestController
public class AnuncioController {
    @GetMapping("/anuncio")
    public Anuncio anuncios() {
        return new Anuncio("Pepe", "Vendo moto", "...");
    }
}
```

Se devuelve el **objeto completo**  
Spring se encarga de **convertir** el  
objeto a **JSON**

- **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);
    }
}
```

# Spring REST

- 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.urjc.code.daw</groupId>
  <artifactId>ejem1</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.2.RELEASE</version>
    <relativePath/>
  </parent>

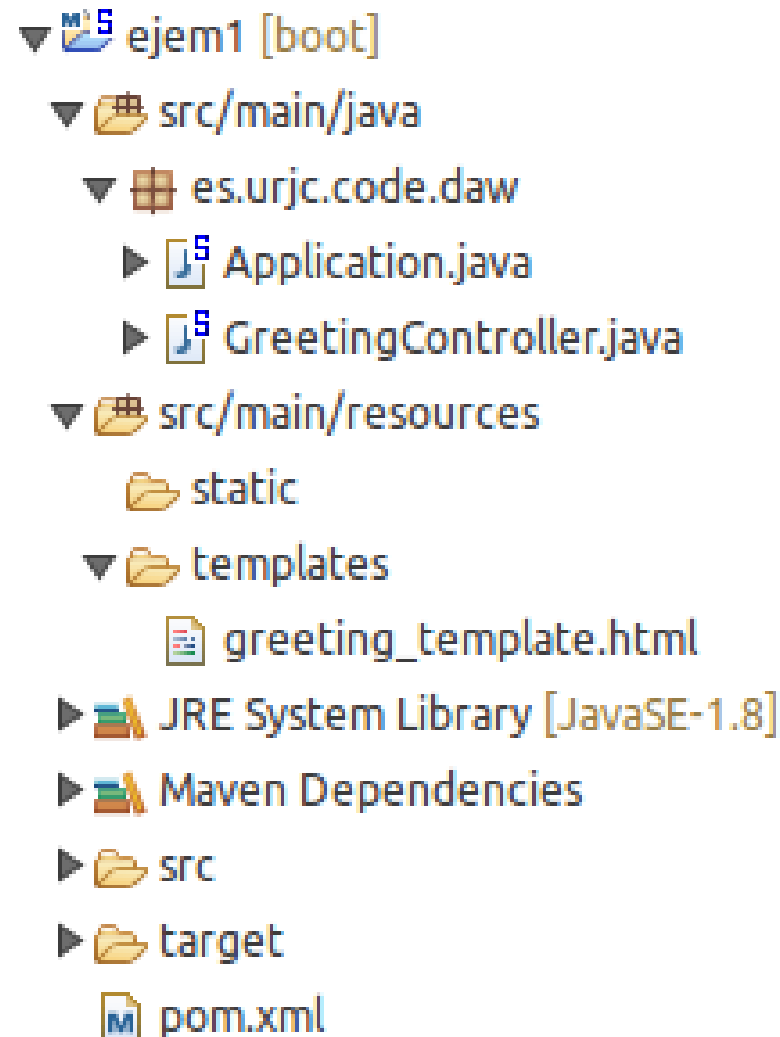
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>1.8</java.version>
  </properties>

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

</project>
```

# APIs REST con Spring

- Estructura de la aplicación

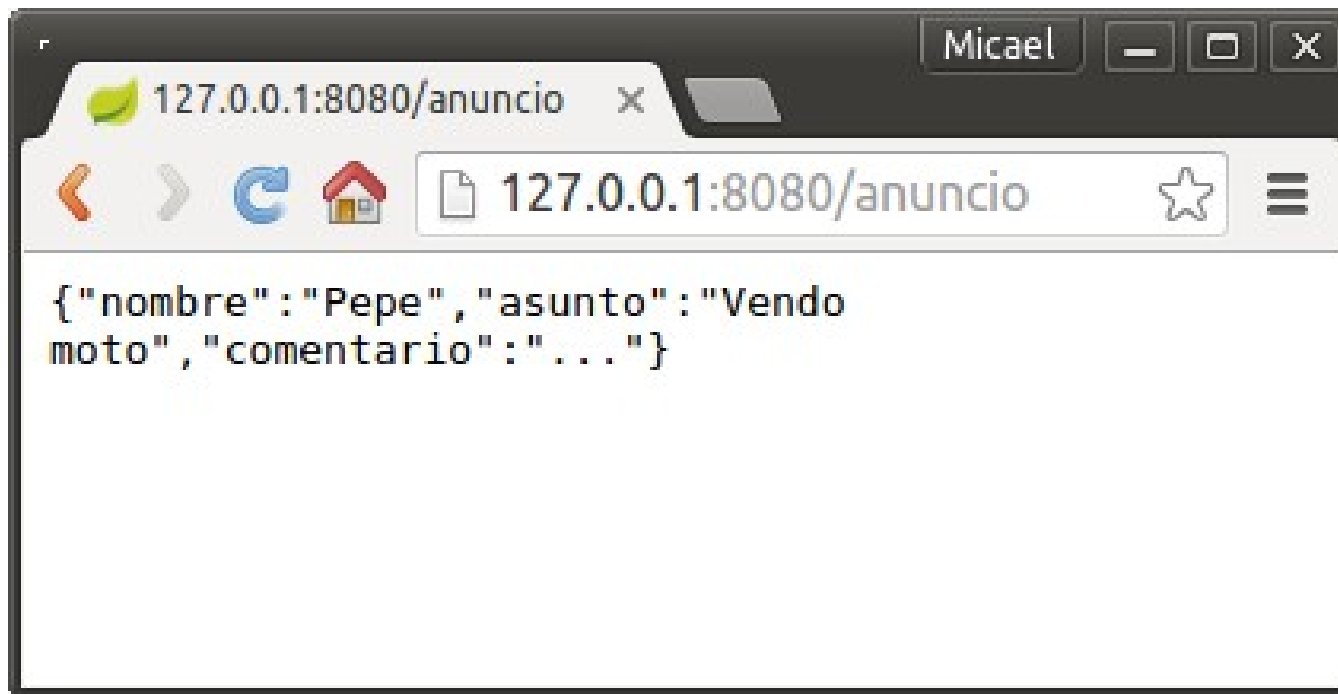


ejem1

# APIs REST con Spring

ejem1

- Petición desde el navegador



- **Procesamiento de URLs con parámetros**
  - Desde el código Java se puede acceder a los valores incluidos como parámetros en la URL
    - <http://micasa.com/ofertas?poblacion=Madrid>

```
@GetMapping("/ofertas")  
public List<Oferta> ofertas(@RequestParam String poblacion) {  
    ...  
}
```

- Parámetros como *paths* en la URL
  - La información también se puede incluir como parte de la propia URL, en vez de como parámetros
  - Se accede a ellos con la anotación `@PathVariable`
    - `http://micasa.com/ofertas/Madrid`

```
@GetMapping("/ofertas/{id}")  
public List<Oferta> ofertas(@PathVariable String poblacion) {  
    ...  
}
```



- **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**
  - La anotación **@ResponseStatus(HttpStatus.CREATED)** indica que se devuelva el estado **201** al cliente si todo va bien
  - Se **devuelve el nuevo objeto** al cliente (con un id)

- Nuevo recurso (POST)

```
@RestController
public class AnunciosController {

    //Atributos y otros métodos...

    @PostMapping("/anuncios")
    @ResponseStatus(HttpStatus.CREATED)
    public Anuncio nuevoAnuncio(@RequestBody Anuncio anuncio) {

        //Se guarda el nuevo anuncio en memoria o BBDD.
        //Se guarda en el anuncio un id único y se devuelve
        return anuncio;
    }
}
```




Diagram illustrating the mapping of the `@ResponseStatus(HttpStatus.CREATED)` annotation to the HTTP status `201 CREATED`. A green box labeled "Estado 201 CREATED" has a green arrow pointing to the `HttpStatus.CREATED` value in the code.

- **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**

- Devolver un recurso concreto (GET)

```
@RestController
public class AnunciosController {

    //Atributos y otros métodos...

    @GetMapping("/anuncios/{id}")
    public ResponseEntity<Anuncio> getAnuncio(@PathVariable long id) {

        //Si está el anuncio con id...
        return new ResponseEntity<>(anuncio, HttpStatus.OK);

        //Si no existe...
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

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

Se devuelve el objeto o NOT FOUND

- **Borrar un recurso (DELETE)**
  - Con **@DeleteMapping** se indica que el método atiende peticiones **DELETE**
  - El id del recurso se condifica 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**

- **Borrar un recurso (DELETE)**

```
@RestController
public class AnunciosController {

    //Atributos y otros métodos...

    @DeleteMapping("/anuncios/{id}")
    public ResponseEntity<Anuncio> borraAnuncio(@PathVariable long id){

        //Si está el anuncio con id, se borra y se devuelve...
        return new ResponseEntity<>(anuncio, HttpStatus.OK);

        //Si no existe...
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

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

Se devuelve el  
objeto o **NOT FOUND**

- **Actualizar 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**

- Actualizar un recurso (PUT)

```
@RestController
public class Anunciador {

    //Atributos y otros métodos...

    @PutMapping("/anuncios/{id}")
    public ResponseEntity<Anuncio> actulizaAnuncio(
        @PathVariable long id, @RequestBody Anuncio anuncioActualizado) {

        //Si está el anuncio con id, se actualiza y se devuelve...
        return new ResponseEntity<>(anuncioActualizado, HttpStatus.OK);

        //Si no existe...
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

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

El anuncio actualizado se envía en el body **@RequestBody**

Se devuelve el objeto o NOT FOUND



- **Ejemplo API REST Anuncios**

- La clase **AnunciosController** gestiona una lista de anuncios en memoria
- Se asigna un **id a cada anuncio** con un contador atómico (para evitar condiciones de carrera)
- El mapa es ***thread-safe***, es decir, puede ser usado por varios hilos a la vez sin que se produzcan condiciones de carrera

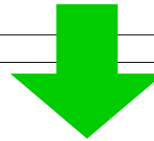
- **Factorizar URL mapping en el controller**
  - Cuando **todas las URLs** de un controlador empiezan de forma **similar**, se puede poner la anotación **@RequestMapping** a nivel de **clase** con la parte común
  - Cada **método** sólo tiene que incluir la **parte propia** (si existe)

- Factorizar URL mapping en el controller

```
@RestController
public class AnunciosController {

    @GetMapping("/anuncios/{id}")
    public ResponseEntity<Anuncio> getAnuncio(@PathVariable long id){...}

    @GetMapping("/anuncios/")
    public List<Anuncio> anuncios(){...}
}
```



```
@RestController
@RequestMapping("/anuncios")
public class AnunciosController {

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

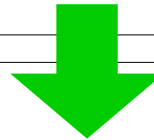
    @GetMapping("/")
    public List<Anuncio> anuncios(){...}
}
```

- Factorizar URL mapping en el controller

```
@RestController
public class AnunciosController {

    @GetMapping("/anuncios/{id}")
    public ResponseEntity<Anuncio> getAnuncio(@PathVariable long id){...}

    @GetMapping("/anuncios/")
    public List<Anuncio> anuncios(){...}
}
```



```
@RestController
@RequestMapping("/anuncios")
public class AnunciosController {

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

    @GetMapping("/")
    public List<Anuncio> anuncios(){...}
}
```

# Ejercicio 1

- Implementa una **API REST** en el servidor para gestionar **Items**
- Los items se gestionarán en **memoria** (como en el ejemplo de los anuncios)
- La API es similar a la que se ha usado en los ejemplos de **apps web SPA**

- **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)

- **API REST Items**

- Consulta de item concreto

- Method: GET
    - URL: `http://127.0.0.1:8080/items/1`
    - Result:

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

- Status code: 200 (OK)

- **API REST Items**

- Modificación de items

- 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)



- **API REST Items**

- Modificación de items

- Method: DELETE
    - URL: `http://127.0.0.1:8080/items/1`
    - Result:

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

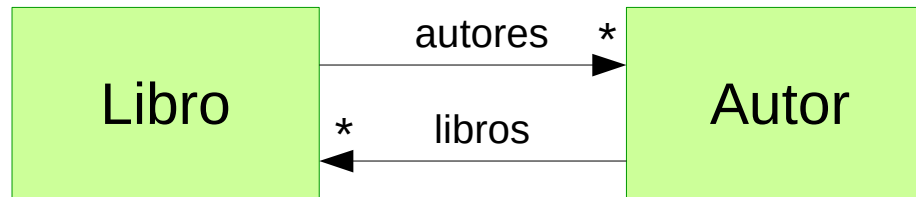
- Status code: 200 (OK)

- Introducción
- Formato JSON
- Funcionamiento de un servicio REST
- Clientes de servicios REST
- APIs REST con Spring
- **Conversión entre objetos y JSON**
- Cliente REST en el servidor

- 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**

# 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;  
}
```

- ¿Qué ocurre si tenemos esta API REST?

```
@RestController
public class LibrosAutoresController {

    private List<Libro> libros = ...

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

- ¿Qué ocurre si tenemos esta API REST?

```
@RestController
public class LibrosAutoresController {

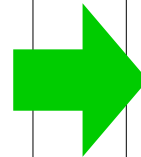
    private List<Libro> libros = ...

    @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.

- 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)



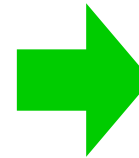
- **Datos diferentes por URL**
  - Creamos un nuevo **interfaz** Java
  - Anotamos algunos atributos con **@JsonView** pasando ese **interfaz** como parámetro
  - Anotamos el método de **@RestController** igual que los atributos (**@JsonView** con el **interfaz** como parámetro)
  - Los objetos que devuelva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

# Conversión entre objetos y JSON

ejem6

```
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  
  }  
]
```

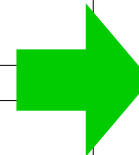
- **Datos diferentes por URL**
  - Si queremos que en un **método** de la **API REST** se devuelvan atributos anotados con diferentes interfaces hay que **crear un nuevo interfaz**
  - Ese **nuevo interfaz** tiene que **heredar** de los interfaces usados por los atributos
  - Usamos ese interfaz en el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

# Conversión entre objetos y JSON

ejem7

```
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  
    }  
  ]  
}
```

- Introducción
- Formato JSON
- Funcionamiento de un servicio REST
- Clientes de servicios REST
- Implementación de una APIs REST con Spring
- Conversión entre objetos y JSON
- **Cliente 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...

- **Cliente REST Java**

- No existe un cliente para peticiones REST en la librería estándar de Java, es necesario usar una **librería externa**:

- Jersey Client
- Apache HttpClient
- Retrofit
- RestTemplate Spring



- **RestTemplate: Cliente REST 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



- **RestTemplate: Cliente REST Spring**
  - Petición a la API de GoogleBooks para extraer los libros

```
RestTemplate restTemplate = new RestTemplate();

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

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

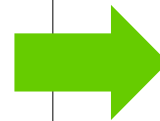
List<String> bookTitles = new ArrayList<String>();

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

return bookTitles;
```

- **RestTemplate: Cliente REST Spring**
  - Para procesar la respuesta se indica la clase que estructura los datos

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



Estas clases pueden tener métodos, atributos privados...

```
class BooksResponse {
    List<Book> items;
}

class Book {
    VolumeInfo volumeInfo;
}

class VolumeInfo {
    String title;
}
```

- **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)

<http://wiki.fasterxml.com/JacksonHome>

- **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);  
}
```

- 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);  
}
```

- **Spring Feign**

- Definición **declarativa** de una API REST que va a ser consumida
  - Spring genera automáticamente las consultas
  - No es necesario usar RestTemplate (aunque Spring lo usa por debajo)
  - Se inyecta como una dependencia más
- Eliminación de código repetitivo y tedioso
- Mismo concepto de los *Repository* de Spring Data

- Consumo de la API REST de anuncios

```
@GetMapping("/")
public String tablon(Model model) {
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<List<Anuncio>> response =
        restTemplate.exchange(
            "http://localhost:8080/anuncios/",
            HttpMethod.GET,
            null,
            new ParameterizedTypeReference<List<Anuncio>>() {});

    model.addAttribute("anuncios", response.getBody());

    return "tablon";
}
```

- Consumo de la API REST de anuncios

```
@FeignClient(value = "anuncios", url = "http://localhost:8080")  
public interface AnunciosService {  
  
    @GetMapping("/anuncios/")  
    List<Anuncio> getAnuncios();  
  
}
```

Anotamos la  
iterfaz con la url  
del servidor rest

Definimos los  
métodos como en un  
controlador, junto  
con el tipo devuelto

```
@Controller  
public class AnunciosController {  
    @Autowired AnunciosService service;  
  
    @GetMapping("/")  
    public String tablon(Model model) {  
        model.addAttribute("anuncios", service.getAnuncios());  
        return "tablon";  
    }  
}
```

Injectamos la  
interfaz y la usamos  
para llamar al  
servidor rest



- 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