Material descargable

egg

API REST con Spring

¿Qué es una API?

Una **API (*Interfaz de Programación de Aplicaciones*)** es un conjunto de funciones y reglas que permiten que distintos programas se comuniquen entre sí a través de solicitudes y respuestas estructuradas.

Las API funcionan como puntos de acceso que facilitan la integración entre aplicaciones, servicios o plataformas, permitiendo el intercambio de datos y la ejecución de funciones de manera controlada y segura. Gracias a esto, los desarrolladores pueden incorporar funcionalidades de otras aplicaciones sin necesidad de programarlas desde cero.

Existen diferentes tipos de API según su accesibilidad:

- **Públicas:** Están disponibles para cualquier desarrollador y pueden utilizarse libremente o bajo ciertas restricciones.
- **Privadas:** Solo pueden ser utilizadas por aplicaciones o usuarios autorizados dentro de una organización o ecosistema específico.

¿Qué es REST?

REST (Transferencia de Estado Representacional) es un estilo de arquitectura de software que se basa en principios clave para facilitar la comunicación entre sistemas a través de Internet. Algunos de estos conceptos clave incluyen:

- **Recursos**: Todo en una API REST es un recurso, identificado de manera única mediante una URL.
- **Verbos HTTP**: Las operaciones CRUD se mapean a los métodos HTTP estándar (GET, POST, PUT, DELETE).
- **Sin estado**: Cada solicitud del cliente al servidor debe contener toda la información necesaria para procesarla, lo que simplifica la escalabilidad.
- **Transferencia de Representación**: Los recursos se transfieren entre el cliente y el servidor en formatos como JSON o XML.
- ***Interfaz Uniforme***: La interfaz de la API REST debe seguir convenciones estandarizadas para las URLs y los métodos HTTP.

Beneficios del uso de API REST

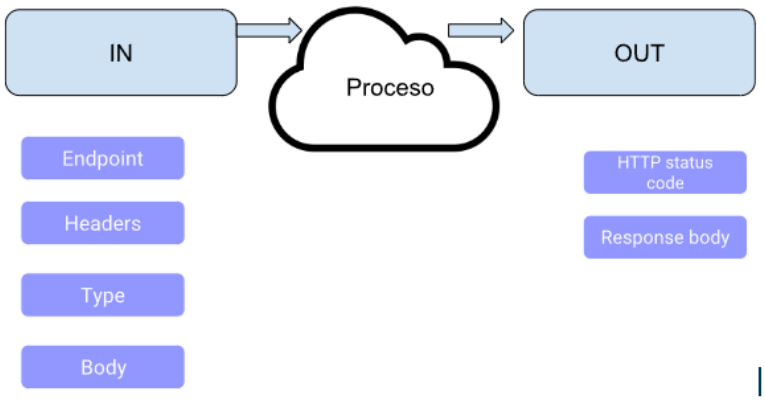
- **Simplicidad**: El protocolo HTTP y los estándares facilitan su comprensión y adopción.
- **Escalabilidad**: La separación entre cliente y servidor permite escalar cada componente de manera independiente.
- **Desacoplamiento**: Cliente y servidor pueden evolucionar independientemente mientras se mantenga la interfaz definida.
- **Facilidad de integración:** Utiliza formatos de datos estándar y protocolos comunes para facilitar la integración.
- **Visibilidad**: La API es auto-descriptiva y fácilmente explorable, mejorando la visibilidad del sistema.
- **Rendimiento**: Aprovecha las características de escalabilidad y cacheabilidad de HTTP para ofrecer buen rendimiento.

Niveles de Madurez en API REST

El Modelo de Madurez de Richardson clasifica los niveles de una API REST según su adherencia a los principios de REST. Estos niveles van desde 0 (el más bajo) hasta 3 (el más alto), donde una API de nivel 3 es considerada "RESTful".

¿Te gustaría explorar más acerca de los niveles de Madurez? Puedes acceder al **siguiente enlace** para obtener información adicional.

1. ¿Cómo funciona una API?



El funcionamiento de una API sigue una serie de pasos que permiten la comunicación entre aplicaciones. A continuación, se detallan los elementos clave involucrados en este proceso:

Punto de entrada:

- Es la dirección a la que se envían las solicitudes para interactuar con la API.
- Generalmente, es una **URL** específica o una ruta de acceso que define el recurso al que se quiere acceder.

Cabeceras (*Headers*):

- Son metadatos incluidos en la solicitud o respuesta HTTP.
- Contienen información como el tipo de contenido (*Content-Type*), autenticación, compresión, entre otros.

Cuerpo de la solicitud (*Body*):

- Incluye los datos que se envían a la API.
- Puede estar en distintos formatos, como **JSON**, **XML** o **texto plano**, según la implementación de la API.

Procesamiento:

- La API recibe la solicitud y realiza las acciones necesarias, como acceder a bases de datos, ejecutar cálculos o interactuar con otros servicios.

Código de estado:

- Es un número de tres dígitos en la respuesta HTTP que indica el resultado de la solicitud.
- Algunos códigos comunes son:
    - **200 OK:** La solicitud se procesó con éxito.
    - **400 Bad Request:** Hubo un error en la solicitud.
    - **401 Unauthorized:** Falta autenticación o no es válida.
    - **404 Not Found:** El recurso solicitado no existe.
    - **500 Internal Server Error:** Error en el servidor.

Respuesta:

- Contiene los datos devueltos por la API, que pueden ser la información solicitada, una confirmación de la acción realizada o un mensaje de error en caso de problemas.

2. Endpoints, queries y params

Un **endpoint** es un punto de acceso único dentro de una API, al que se puede acceder a través de una **URL** (*Uniform Resource Locator*). Es la dirección específica donde se envían solicitudes y desde donde se reciben respuestas. Cada **endpoint** está diseñado para ejecutar una función específica, como recuperar, crear, actualizar o eliminar recursos.

Para personalizar las solicitudes y obtener respuestas más específicas, se pueden incluir **parámetros** en la URL. Estos varían según el tipo de solicitud HTTP (*GET, POST, PUT, DELETE, etc.*) y su propósito. Existen dos tipos principales de parámetros:

- **Parámetros de Ruta (*Path Parameters*):**
  - Se utilizan para identificar un recurso específico dentro de la API.
  - Se incluyen directamente en la URL, generalmente como parte de la estructura del **endpoint**.
  - Ejemplo**: /usuarios/123** → Aquí, 123 es un parámetro de ruta que representa el ID de un usuario específico.
- **Parámetros de Consulta (*Query Parameters*):**
  - Permiten filtrar, ordenar o personalizar la respuesta del servidor.
  - Se añaden al final de la URL, después de un signo de interrogación (?). Si hay más de un parámetro, se separan con el símbolo &.
  - Ejemplo**: /usuarios?edad=25&pais=Argentina** → En este caso, edad=25 y pais=Argentina son parámetros de consulta que filtran la lista de usuarios según la edad y el país.

3. Tipos de peticiones

En el contexto de las API, las solicitudes HTTP permiten interactuar con los recursos del servidor. A continuación, se presentan las peticiones más comunes, alineadas con las operaciones del modelo **CRUD** (*Create, Read, Update, Delete*):

GET ***Lectura de Datos (Read)***	Se utiliza para solicitar datos de un recurso específico. Es una operación de **solo lectura**, por lo que **no modifica** los datos en el servidor.	• **Segura**: No altera el estado del servidor. • **Idempotente**: Puede repetirse sin generar cambios en los datos. • No debe incluir un cuerpo de solicitud.

POST ***Creación de Recursos (Create)***	Se utiliza para **enviar datos** al servidor y crear un nuevo recurso. Comúnmente se emplea para **formularios** o **datos estructurados**.	• **No segura**: Modifica el estado del servidor. • **No idempotente**: Múltiples solicitudes pueden generar múltiples recursos. • **Debe incluir** un cuerpo de solicitud con los datos a enviar.
PUT ***Actualización Total de Recursos (Update)***	Se utiliza para **actualizar completamente** un recurso existente.	• **No segura**: Modifica el estado del servidor. • **Idempotente**: Repetir la misma solicitud no genera cambios adicionales. • **Debe incluir** un cuerpo de solicitud con todos los datos del recurso, incluso si no cambian.
DELETE ***Eliminación de Recursos (Delete)***	Se utiliza para **eliminar** un recurso del servidor.	• **No segura**: Modifica el estado del servidor. • **Idempotente**: Si el recurso ya fue eliminado, repetir la solicitud no genera errores. • **No debe incluir** un cuerpo de solicitud.

🚀 **¿Quieres profundizar más?** Existen otras peticiones HTTP con funcionalidades adicionales. Te invitamos a investigar sobre:

- **PATCH** → Para actualizaciones parciales.
- **HEAD** → Para obtener solo los encabezados de una respuesta.
- **OPTIONS** → Para conocer los métodos soportados por un endpoint.

4. Códigos de respuesta HTTP

Los códigos de estado HTTP son respuestas estándar que los servidores web envían a los clientes para indicar el resultado de una solicitud. Se dividen en cinco categorías principales, cada una con un propósito específico. Conocerlas facilita la interpretación de las respuestas del servidor y permite diagnosticar posibles errores de manera más eficiente.

Las principales clases de códigos HTTP son:

- **1xx - Informativos**: Indican que la solicitud fue recibida y el servidor continúa procesándola.

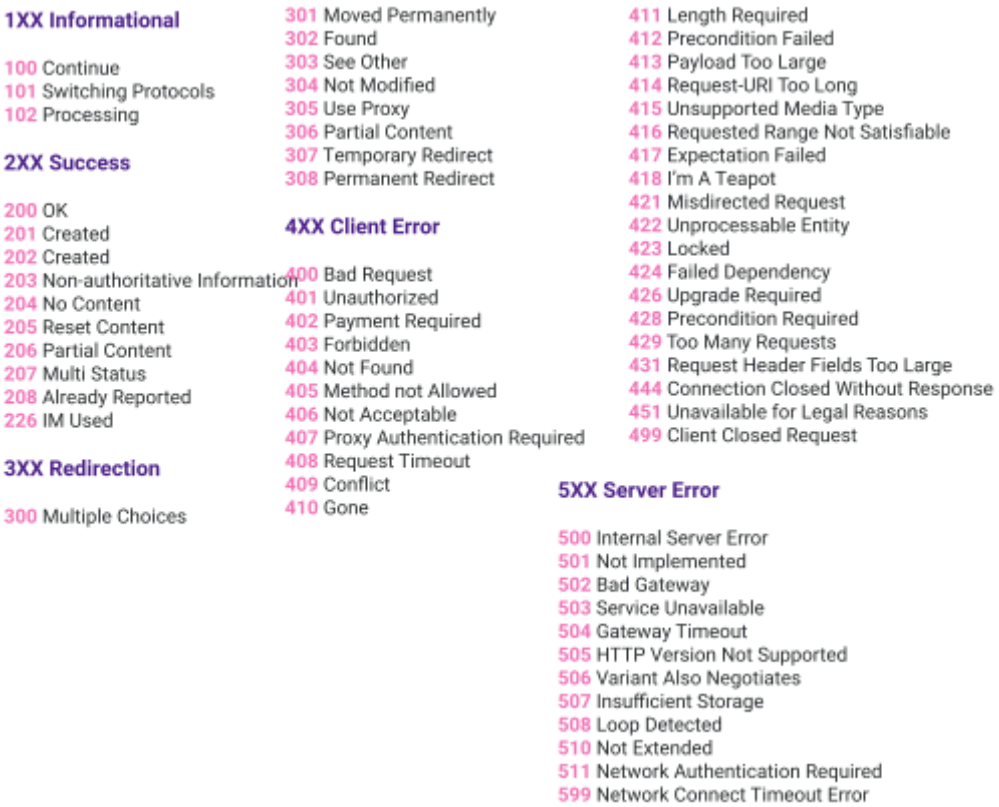
- **2xx - Éxito:** La solicitud fue recibida, comprendida y procesada correctamente.

- **3xx - Redirección :** Indican que la solicitud debe dirigirse a un nuevo recurso o ubicación.

- **4xx - Errores del Cliente:** Representan problemas en la solicitud realizada por el cliente, como un recurso inexistente o falta de autenticación.

- **5xx - Errores del Servidor:** Indican fallos en el servidor que impiden completar la solicitud correctamente.

A continuación, encontrarás un listado detallado con los códigos más comunes en cada categoría.



¿Deseas aprender más sobre los códigos HTTP? Puedes acceder al **siguiente enlace** para obtener información adicional.

Implementación de una API REST con Spring

En este módulo nos centraremos en construir aplicaciones de nivel 2 del Modelo de Madurez de Richardson, ya que es lo más común en el mercado. Aquí están los pasos básicos que seguiremos:

- **Modelo de Datos:** Definiremos las entidades que usaremos en nuestra aplicación.
- **Acceso a Datos:** Utilizaremos Spring Data JPA para crear la capa de acceso a datos.
- **Capa de Servicios:** Implementaremos las reglas de negocio en la capa de servicios.
- **Controladores REST:** Utilizaremos la notación @RestController de Spring para crear controladores que manejen solicitudes RESTful.
- **Verbos HTTP**: Implementaremos los métodos HTTP estándar (GET, POST, PUT, DELETE) para realizar operaciones CRUD en nuestros recursos.

Las capas se mantienen igual a como venías trabajando. La diferencia estará en los controladores que ya no retornarán una página HTML, sino que deberán recibir y proveer información en formato JSON.

Uso de `@RestController` en Spring

La anotación **@RestController** en Spring simplifica la creación de controladores que manejan solicitudes RESTful al combinar la funcionalidad de @Controller y @ResponseBody. Veamos en detalle cómo funciona:

- @Controller: Esta anotación marca una clase como controlador en Spring MVC. Indica que la clase define uno o más métodos para manejar solicitudes HTTP.
- @ResponseBody: Esta anotación indica que el valor devuelto por un método de controlador debe ser serializado directamente al cuerpo de la respuesta HTTP, en lugar de ser interpretado como el nombre de una vista para resolver.

Al utilizar la anotación @RestController en la declaración de la clase, cada método dentro de dicha clase se considera un controlador de solicitud. Esto significa que

cada método está preparado para manejar solicitudes HTTP entrantes. Además, cualquier valor devuelto por estos métodos se serializa automáticamente al formato adecuado, como JSON o XML, mediante un convertidor de mensajes integrado en Spring.

Manejo de Solicitudes HTTP en Spring Boot

En Spring Boot, los métodos de un controlador se utilizan para manejar solicitudes HTTP. Cada método en el controlador puede estar anotado con una anotación que indica el tipo de solicitud HTTP que maneja (por ejemplo, `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`).

Ejemplo de Método:

```
@PostMapping("crear")
    public ResponseEntity<Object> crearAutor(String nombre) {
        try {
            autorServicio.crearAutor(nombre);
            return new ResponseEntity<>(HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
```

Explicación de Componentes:

1. **Anotaciones de Mapeo:**
    - `@PostMapping`: Esta anotación indica que el método manejará las solicitudes HTTP POST. En este caso, la URL a la que responde es "crear".
2. **Firma del Método:**
    - `public ResponseEntity<Object> crearAutor(@RequestParam String nombre)`: El método `crearAutor` acepta un parámetro `nombre` (de tipo String) enviado en la solicitud HTTP. La anotación `@RequestParam` indica que el parámetro se pasa como un parámetro de consulta (query parameter) en la URL o en el cuerpo de la solicitud.
3. **Cuerpo del Método:**
    - `autorServicio.crearAutor(nombre);`: Aquí se llama a un servicio para crear un nuevo autor. Esta es la lógica de negocio que se ejecuta cuando se recibe la solicitud.
4. **Respuesta HTTP:**
    - `return new ResponseEntity<>(HttpStatus.OK);`: Si la operación es exitosa, se devuelve una respuesta con un código de

estado HTTP 200 (OK). El `ResponseEntity` permite establecer no solo el cuerpo de la respuesta (que puede ser `null` en este caso) sino también el código de estado.

- `return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);`: Si ocurre una excepción, se devuelve una respuesta con un código de estado HTTP 500 (Internal Server Error).

Tipos de Métodos HTTP:

- **`@GetMapping`**: Maneja solicitudes HTTP GET. Se usa para obtener datos del servidor.
- **`@PostMapping`**: Maneja solicitudes HTTP POST. Se usa para enviar datos al servidor para crear un nuevo recurso.
- **`@PutMapping`**: Maneja solicitudes HTTP PUT. Se usa para actualizar un recurso existente en el servidor.
- **`@DeleteMapping`**: Maneja solicitudes HTTP DELETE. Se usa para eliminar un recurso del servidor.

Otras Anotaciones Útiles:

- **`@RequestMapping`**: Puede usarse para manejar solicitudes HTTP de cualquier tipo (GET, POST, etc.) y permite especificar una URL base para el método del controlador.
- **`@PatchMapping`**: Maneja solicitudes HTTP PATCH. Se usa para realizar actualizaciones parciales en un recurso.