Contenido

INTRODUCCION

Los sistemas distribuidos son un modelo de computación en el que varios equipos de hardware y software, ubicados en distintos lugares, colaboran para proporcionar un servicio conjunto. Estos sistemas están diseñados para funcionar de manera coordinada, ofreciendo una vista unificada al usuario, a pesar de que sus componentes pueden estar dispersos geográficamente y administrados de manera independiente.

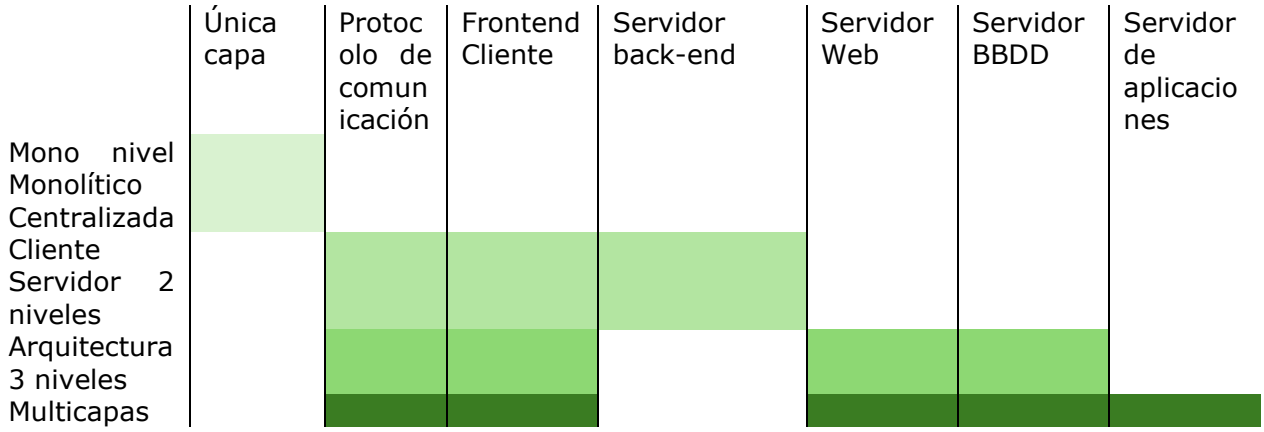
Una de las principales características de los sistemas distribuidos es la **transparencia**. Esto significa que los usuarios y las aplicaciones perciben el sistema como una única entidad cohesiva, sin necesidad de conocer la ubicación de los recursos o la distribución de las tareas. Otro aspecto fundamental es la **escalabilidad**, que permite aumentar o reducir la capacidad del sistema añadiendo o eliminando recursos sin afectar su funcionamiento global.

Los sistemas distribuidos también se destacan por su **tolerancia a fallos**. La redundancia y la replicación de componentes permiten que, ante fallos en algunas partes del sistema, otras puedan asumir sus funciones, manteniendo el servicio en funcionamiento.

Estos sistemas son la base de muchas aplicaciones modernas, como los servicios en la nube, redes sociales, plataformas de streaming y sistemas de comercio electrónico. A medida que la demanda de servicios confiables y escalables aumenta, los sistemas distribuidos se han vuelto una pieza fundamental en la infraestructura tecnológica actual

CAPAS DEL SISTEMA DISTRIBUIDO:

- **Mononivel (Monolítico):** Sistema centralizado.
- **Cliente-Servidor de 2 Niveles:** Separación entre cliente y servidor.
- **Arquitectura de 3 Niveles:** Presentación, lógica de negocio y acceso a datos distribuidos.
- **Multicapa:** Arquitectura distribuida en más de tres capas.



ARQUITECTURA ORIENTADA A SERVICIOS (SOA):

Es un paradigma que organiza y utiliza capacidades distribuidas bajo el control de diferentes propietarios y dominios. Sus principios incluyen:

1. Contrato de servicio.
2. Bajo acoplamiento.
3. Abstracción.
4. Reusabilidad.
5. Autonomía.
6. Sin estado.
7. Capacidad de descubrimiento.
8. Composiciones orquestadas.

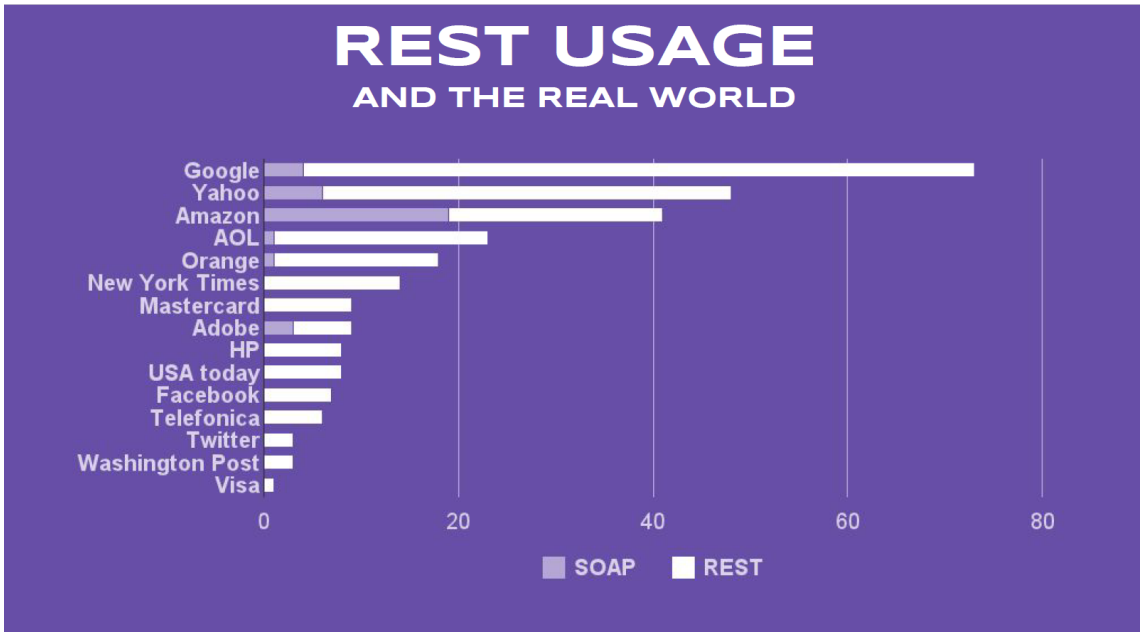
Dentro de este enfoque, los servicios web y la arquitectura REST (Representational State Transfer) son dos maneras comunes de implementar SOA. Los servicios web suelen usar protocolos estándar como SOAP (Simple Object Access Protocol), junto con lenguajes de descripción como WSDL (Web Services Description Language), para definir y ejecutar operaciones complejas, muchas veces en entornos empresariales que requieren transacciones seguras y confiables. Por otro lado, la arquitectura REST se enfoca en la simplicidad y la escalabilidad, utilizando el protocolo HTTP para interactuar con recursos identificados por URIs y favoreciendo la transferencia de datos en formatos como JSON o XML. Mientras que los servicios web proporcionan una estructura más rígida y formal, REST se destaca por su flexibilidad y menor sobrecarga, lo que facilita la integración y la comunicación entre sistemas heterogéneos. Ambos enfoques son complementarios dentro de la arquitectura SOA, permitiendo la creación de sistemas distribuidos robustos y escalables. TIPOS DE COLABORACIÓN ENTRE SERVICIOS:

RESTFULL

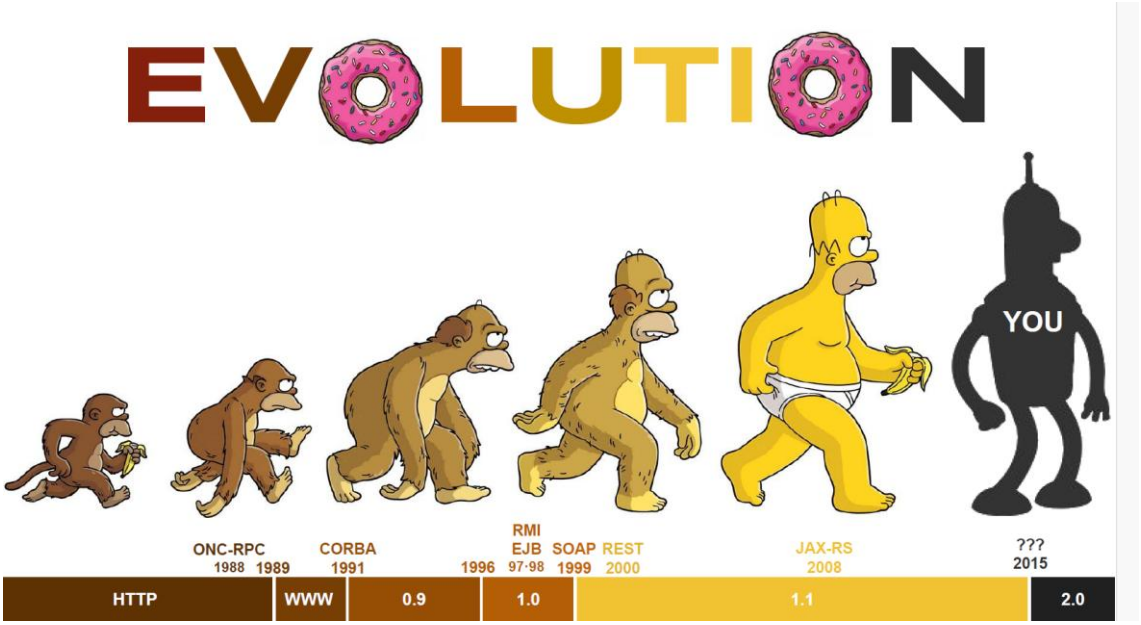
La arquitectura REST (Representational State Transfer) es un estilo arquitectónico para el diseño de sistemas distribuidos, especialmente utilizado en la creación de servicios web. Desarrollada por Roy Fielding en su tesis doctoral en el año 2000, REST define un conjunto de principios que establecen cómo los sistemas en red, como el web, deben comunicarse e intercambiar datos de manera eficiente y escalable.

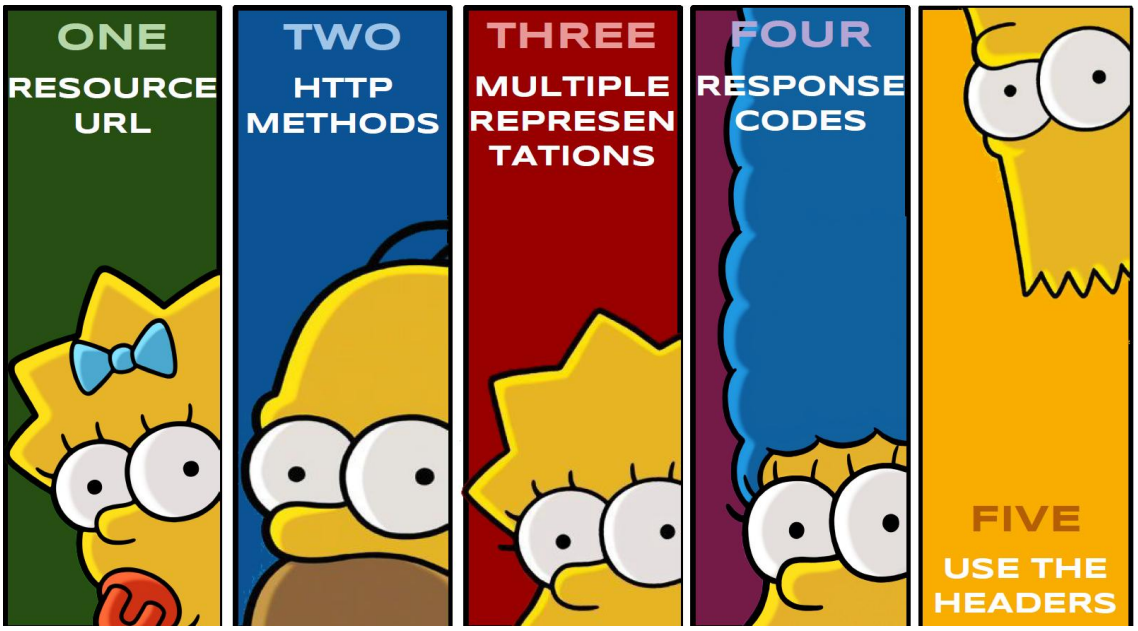
REST se basa en la idea de recursos, que son identificados de forma única a través de URIs (Uniform Resource Identifiers) y manipulados mediante operaciones estándar del protocolo HTTP, como GET, POST, PUT y DELETE. Cada recurso puede tener múltiples representaciones, como JSON, XML o HTML, que permiten la interacción flexible con diferentes tipos de clientes. Uno de los principios fundamentales de REST es su naturaleza sin estado, lo que significa que cada solicitud del cliente al servidor debe contener toda la información necesaria para ser procesada, sin depender de ninguna información almacenada en el servidor entre solicitudes.

Un sistema RESTful es aquel que implementa la arquitectura REST de manera coherente, adhiriéndose a sus principios y restricciones. Esto permite la creación de sistemas altamente escalables y fácilmente mantenibles, ya que la interfaz uniforme y la ausencia de estado facilitan la administración de servicios distribuidos y la integración con otras aplicaciones. Los sistemas RESTful son especialmente populares en el desarrollo de APIs (Application Programming Interfaces), ya que su diseño basado en recursos y su uso de métodos HTTP estándar permiten una comunicación clara, eficiente y bien estructurada entre diferentes sistemas.



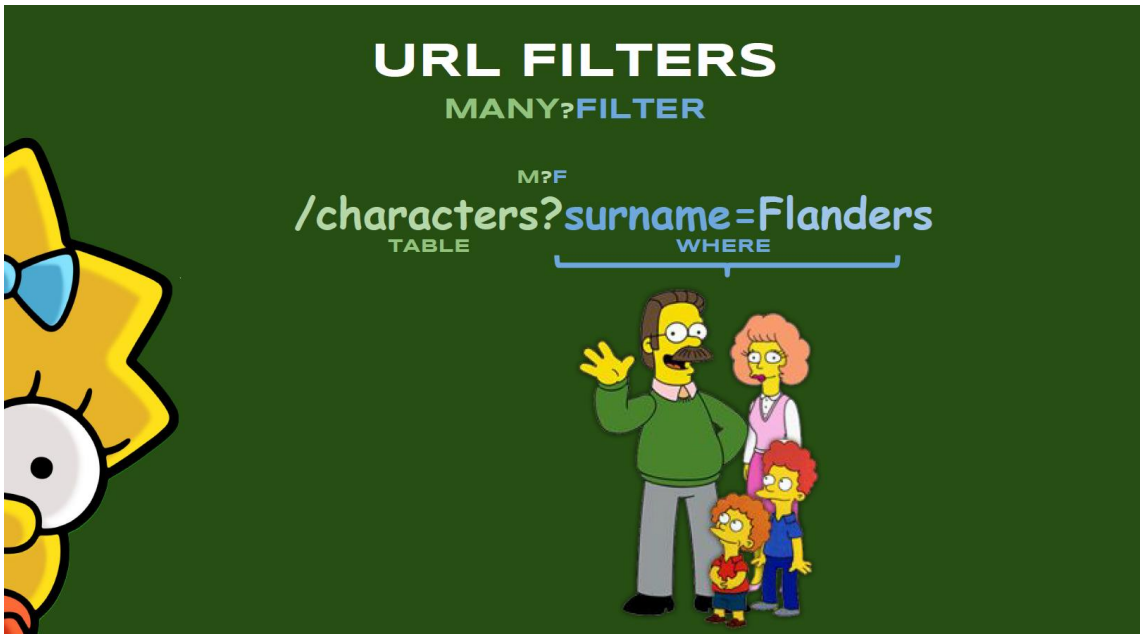
RESTful: Servicios web ligeros que utilizan HTTP para interacciones y se basan en operaciones CRUD sobre recursos definidos por URIs. Para mayor información consultar el enlace https://en.wikipedia.org/wiki/REST





RECURSOS DEFINIDOS POR URI

En las aplicaciones RESTful, los recursos se identifican y acceden mediante URIs (Uniform Resource Identifiers). Un recurso es cualquier entidad que puede ser manipulada o representada, como un objeto, un archivo, un servicio o cualquier tipo de dato. Cada recurso tiene una URI única que actúa como su dirección, lo que permite que los clientes puedan acceder y manipularlo de manera directa y específica. Por ejemplo, en una API de gestión de usuarios, un recurso podría ser un usuario específico identificado con la URI /users/123. Esta URI representa a ese usuario en particular, y cualquier operación realizada sobre ella, como una solicitud GET para recuperar su información o una solicitud DELETE para eliminarlo, se refiere exclusivamente a ese recurso.



METODOS HTTP

Un sistema RESTful utiliza los métodos HTTP estándar para interactuar con los recursos identificados por URIs, permitiendo realizar operaciones específicas sobre ellos. Los cuatro métodos principales son: **GET**, utilizado para recuperar información de un recurso sin modificarlo; **POST**, empleado para crear nuevos recursos o realizar operaciones que involucren la manipulación de datos en el servidor; **PUT**, que se utiliza para actualizar o reemplazar un recurso existente; y **DELETE**, que permite eliminar un recurso identificado por la URI. Al adherirse a estos métodos estándar, los sistemas RESTful proporcionan una interfaz uniforme y predecible para la interacción con los recursos, aprovechando el protocolo HTTP de manera eficiente y natural.

Además, el uso de estos métodos HTTP garantiza que las operaciones sobre los recursos sean claras y semánticamente apropiadas, facilitando la comunicación y la comprensión entre el cliente y el servidor. Por ejemplo, el uso de GET para obtener información garantiza que la operación no tiene efectos secundarios, mientras que PUT y DELETE son idempotentes, es decir, realizar la misma operación varias veces producirá el mismo resultado. Esto hace que las APIs RESTful sean robustas, fáciles de consumir y ampliamente adoptadas para el desarrollo de servicios web modernos.

HTTP METHODS

Method	Description	Type
GET	Obtener recurso o colección	safe & idempotent
POST	Crear sin ID (Location) Actualizar uno/más recursos	non-idempotent
PUT	Crear con ID Actualizar completamente	idempotent
DELETE	Eliminar recurso o colección	idempotent
PATCH (*)	Actualizar parcialmente	idempotent

PATCH (*) is proposed, but at moment is not included into HTTP spec

POST METHOD

Usa **POST** cuando no encaje ningún otro método:

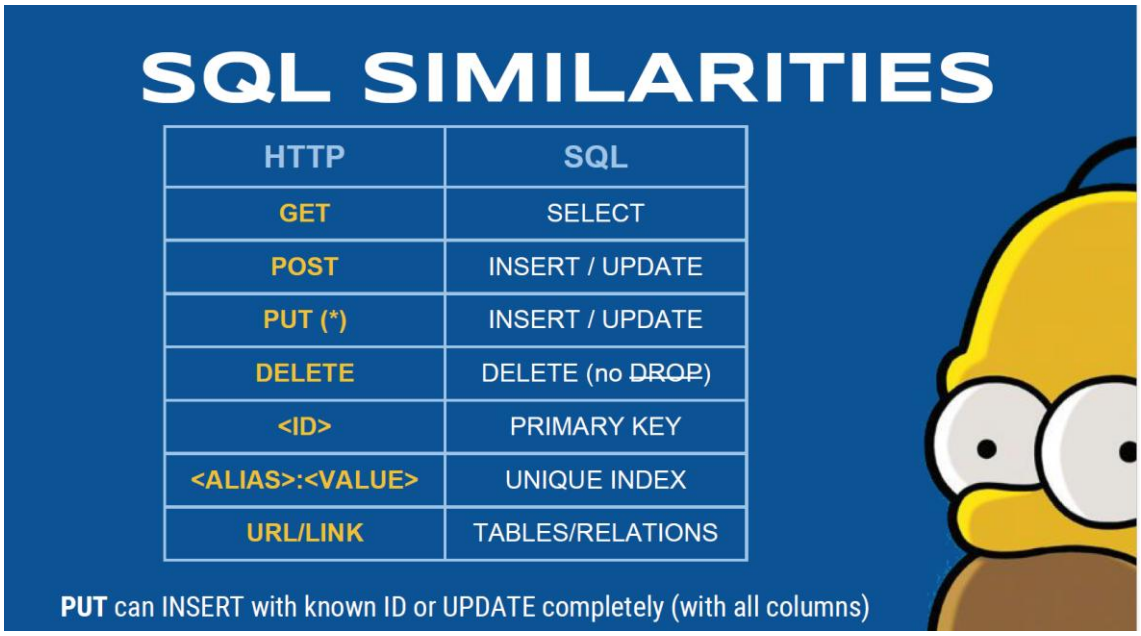
GET: obtener/recuperar recursos
PUT: actualizar/reemplazar totalmente un recurso
DELETE: eliminar/borrar un recurso
PATCH: actualizar parcialmente un recurso (*)

Casi todas actualizaciones serán parciales, salvo en ficheros/docs:

PATCH queda desaconsejado hasta que se incluya en HTTP
PUT queda "relegado" a crear recursos con URL/ID conocido

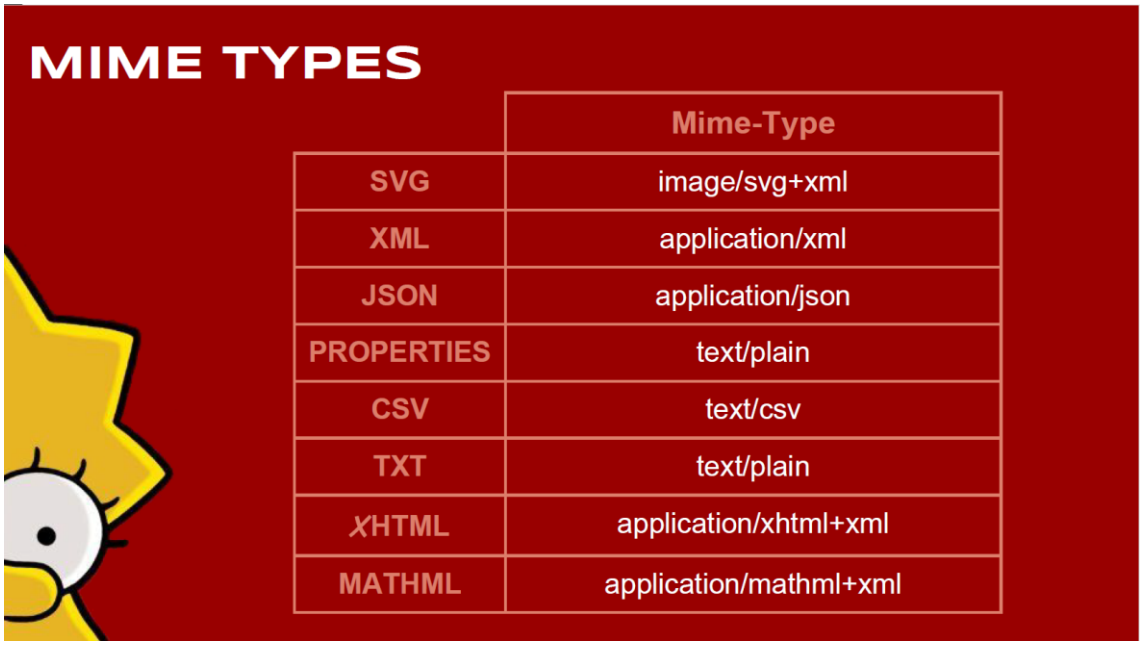
El 99% de las acciones que actualicen tus recursos son POST

POST can't retrieve (GET) or remove (DELETE) resources



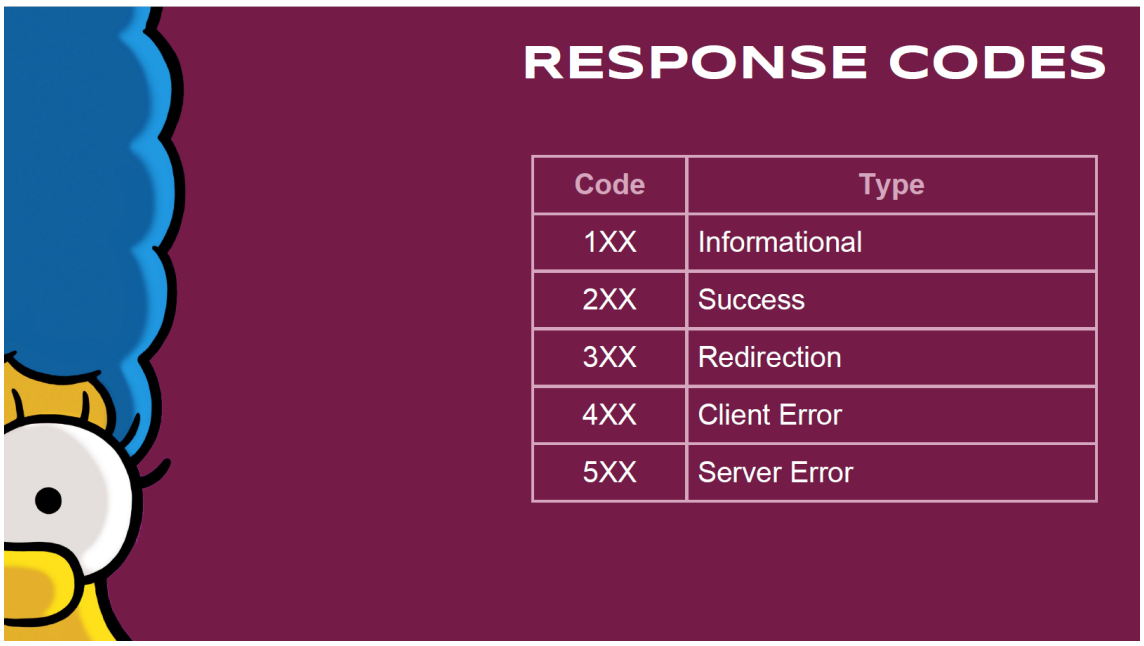
MULTIPLES REPRESENTACIONES

uno de sus principios fundamentales es la aceptación de múltiples representaciones de un recurso. Esto significa que un mismo recurso puede ser representado en diferentes formatos, como JSON, XML, HTML o texto plano, permitiendo a los clientes elegir la forma que mejor se adapte a sus necesidades. Esta flexibilidad no solo facilita la interoperabilidad entre diferentes sistemas, sino que también mejora la experiencia del usuario, ya que se puede acceder a la información de manera más conveniente y eficiente, dependiendo del contexto y la aplicación utilizada.



Para consultar la sintaxis de JSON : https://www.json.org/json-en.html

CODIGOS DE RESPUESTAS



1×× Informational

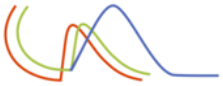
100 Continue

101 Switching Protocols

102 Processing

2×× Success

200 OK



201 Created

202 Accepted

203 Non-authoritative Information

204 No Content

205 Reset Content

206 Partial Content

207 Multi-Status

208 Already Reported

226 IM Used

3×× Redirection

300 Multiple Choices

301 Moved Permanently

302 Found

303 See Other

304 Not Modified

305 Use Proxy

307 Temporary Redirect

308 Permanent Redirect

4×× Client Error

400 Bad Request

401 Unauthorized

402 Payment Required

403 Forbidden

404 Not Found

405 Method Not Allowed

406 Not Acceptable

407 Proxy Authentication Required

408 Request Timeout

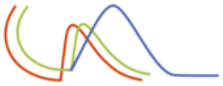
409 Conflict

410 Gone

411 Length Required

412 Precondition Failed

413 Payload Too Large



414 Request-URI Too Long

415 Unsupported Media Type

416 Requested Range Not Satisfiable

417 Expectation Failed

418 I'm a teapot

421 Misdirected Request

422 Unprocessable Entity

423 Locked

424 Failed Dependency

426 Upgrade Required

428 Precondition Required

429 Too Many Requests

431 Request Header Fields Too Large

444 Connection Closed Without Response

451 Unavailable For Legal Reasons

499 Client Closed Request

5×× Server Error

500 Internal Server Error

501 Not Implemented

502 Bad Gateway

503 Service Unavailable

504 Gateway Timeout

505 HTTP Version Not Supported

506 Variant Also Negotiates

507 Insufficient Storage

508 Loop Detected

510 Not Extended

511 Network Authentication Required

599 Network Connect Timeout Error

USAR ENCABEZADOS

Una de las características clave de REST es el uso de cabeceros HTTP, que permiten transmitir información adicional sobre la solicitud o la respuesta entre el cliente y el servidor. Estos cabeceros pueden incluir metadatos como el tipo de contenido (por

ejemplo, Content-Type), información sobre la autenticación (como Authorization), o detalles sobre la caché (por ejemplo, Cache-Control). Al emplear cabeceros, los desarrolladores pueden gestionar de manera más precisa el comportamiento de las interacciones, optimizando el rendimiento y garantizando la correcta interpretación de los datos. Esto facilita la comunicación clara y eficiente entre distintas aplicaciones, contribuyendo a una arquitectura más robusta y escalable.

Para mayor información puedes consultar https://spring.io/guides/tutorials/rest

VALIDACIONES

La validación de datos es un aspecto crucial en el desarrollo de aplicaciones web, ya que garantiza que los datos proporcionados por los usuarios o sistemas externos cumplan con ciertos requisitos antes de ser procesados. En el contexto de Spring Framework, se utilizan las anotaciones **@Valid** y **@Validated** para simplificar este proceso. Estas anotaciones permiten aplicar restricciones de validación a nivel de clase y de método, asegurando que los datos recibidos respeten reglas como formato, longitud, y valores permitidos. **@Valid** se usa comúnmente en parámetros de métodos y clases de entidades para activar validaciones estándar, mientras que **@Validated** habilita validaciones más complejas, como grupos de validación. Ambas integraciones son posibles gracias a tecnologías como **Hibernate Validator**, la implementación de referencia de Bean Validation (JSR 303 y JSR 380).

Para mayor información puedes consultar

https://docs.jboss.org/hibernate/validator/9.0/reference/en-US/html_single/

DTO

Los DTO (Data Transfer Objects) son objetos que se utilizan en aplicaciones RESTful para transferir datos entre el cliente y el servidor de manera eficiente. Su principal propósito es encapsular y estructurar la información que se envía o recibe a través de la API, minimizando el número de llamadas necesarias y optimizando el rendimiento de la comunicación. Al usar DTO, se pueden enviar solo los campos relevantes, reduciendo así el tamaño de las cargas útiles y mejorando la seguridad al ocultar datos innecesarios. Además, permiten desacoplar la lógica de la aplicación del modelo de dominio, facilitando la evolución del sistema y la adaptación a diferentes necesidades de presentación o consumo de datos.

Para mayor información puedes consultar http://modelmapper.org/user-manual/property-mapping/

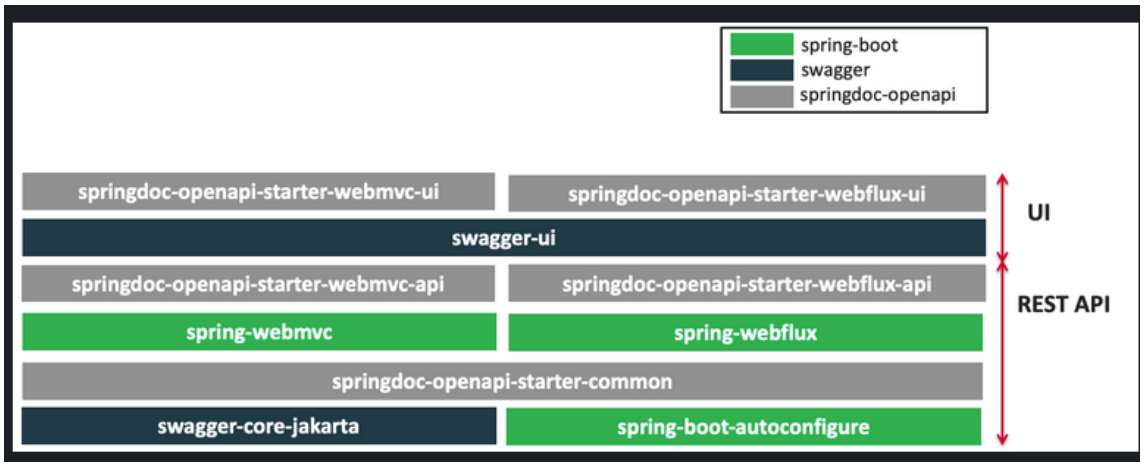
DOCUMENTACION CON SPRINGDOC-OPENAPI MODULES:

DEPENDENCIA

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

<version>2.5.0</version>

</dependency>



Springdoc es una biblioteca de código abierto para la generación automática de documentación de API REST para aplicaciones basadas en Spring Framework en Java. Sirve para simplificar y agilizar el proceso de documentación de APIs, extrayendo información directamente de los controladores y las anotaciones de Spring. Esto significa que los desarrolladores no tienen que escribir documentación manualmente, lo que reduce la posibilidad de errores y mantiene la documentación siempre actualizada con el código.

Springdoc puede generar documentación en varios formatos, como OpenAPI (anteriormente conocido como Swagger), que es un estándar ampliamente aceptado para describir y documentar APIs RESTful. Esta documentación puede ser utilizada por desarrolladores, equipos de pruebas, y otros interesados para comprender fácilmente cómo utilizar la API y qué esperar de ella. En resumen, Springdoc simplifica la tarea de documentar APIs en aplicaciones Spring, mejorando la eficiencia del desarrollo y la calidad de la documentación.

Requerimiento mínimo java 17

Para consultar la documentación creada usar los siguientes enlaces

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

http://localhost:8080/swagger-ui/index.html#/

ENLACES DE INTERÉS

INGLES

https://martinfowler.com/articles/richardsonMaturityModel.html

ESPAÑOL

https://medium.com/@davisaac8/modelo-de-madurez-de-richardson-637f77e06dba

https://martinfowler.com/articles/richardsonMaturityModel.html

https://springdoc.org/#properties

https://swagger.io/docs/