Curso Completo
APIs y Servicios REST

Christian Ramirez
@christian_ramireezz

@latecnologiaavanza

API

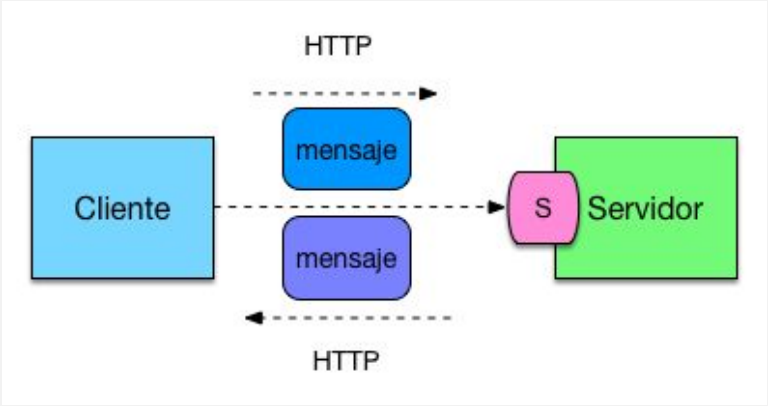
Definición

- **API** significa **Application Programming Interface** (Interfaz de Programación de Aplicaciones)
- Una API es un conjunto de reglas que permiten que dos aplicaciones se comuniquen entre sí
- Expone funcionalidades o datos para que otros sistemas puedan utilizarlos, sin necesidad de conocer su implementación interna

REST

Definición

- REST (Representational State Transfer) es un estilo arquitectónico propuesto por Roy Fielding para diseñar servicios web escalables y simples



Características

- **Recursos**: Todo se trata como un recurso (ej. usuarios, productos)
- **Identificación por URI**: Cada recurso tiene una dirección única
- **Operaciones estándar (HTTP)**: Se usan métodos como GET, POST, PUT, DELETE
- **Sin estado** (*stateless*): Cada solicitud debe contener toda la información
- **Formato estándar**: Normalmente JSON o XML, aunque hoy predomina JSON

JSON

Definición

- **JSON (JavaScript Object Notation)** es un formato ligero de texto para **almacenar e intercambiar datos**
- Es fácil de leer para humanos y fácil de procesar por máquinas
- Se basa en pares **clave : valor**

Ejemplo de JSON

```json
{
  "id": 1,
  "nombre": "Ana",
  "correo": "ana@example.com"
}
```

Métodos HTTP

Definición

- Los **métodos HTTP** son acciones que se usan en las solicitudes para decirle al servidor **qué operación queremos hacer** sobre un recurso (por ejemplo, un usuario o un producto)

- HTTP (*HyperText Transfer Protocol*) es un **protocolo de comunicación** usado por la web para enviar y recibir datos

Métodos HTTP

Método	Descripción	Ejemplo
GET	Obtener información	`GET /usuarios`
POST	Crear un nuevo recurso	`POST /usuarios`
PUT	Actualizar un recurso completo	`PUT /usuarios/1`
PATCH	Actualizar parcialmente un recurso (solo un campo)	`PATCH /usuarios/1`
DELETE	Eliminar un recurso	`DELETE /usuarios/1`

Método idempotente

- Un **método idempotente** es aquel que, **aunque se ejecute varias veces**, **produce el mismo resultado** en el servidor después de la primera vez

- `PUT /usuario/123`: si envías los mismos datos varias veces, el estado final del recurso será siempre igual

- `POST /usuario`: cada vez que lo envías, crea un nuevo usuario → **resultado distinto cada vez**

Códigos de Estado HTTP

Los códigos de estado indican el **resultado de una solicitud** HTTP. Son devueltos por el servidor y ayudan al cliente (como Postman, navegador o frontend) a entender qué ocurrió

Código	Significado	Cuándo usarlo
200	OK	Respuesta exitosa (GET, PUT, DELETE)
201	Created	Recurso creado (POST)
204	No Content	Operación exitosa sin contenido (DELETE)
400	Bad Request	El cliente envió datos incorrectos

URI, URL y URN

URI – Uniform Resource Identifier

Una **URI** es una **cadena de caracteres** que identifica un recurso de forma **única**, puede hacerlo mediante:

- **su ubicación** (como una URL), o
- **su nombre** (como un URN), o
- ambas cosas

URI – Uniform Resource Identifier

- Su propósito es identificar de forma única **cualquier recurso accesible o no** en la web: documentos, servicios, personas, etc
- URI es el concepto más general
- `https://api.miapp.com/usuarios/5` (es también una URL)
- urn:isbn:9780141036144 (es un URN)

URL – Uniform Resource Locator

Una **URL** es un tipo de URI que **especifica la ubicación de un recurso** en la red y **cómo acceder a él**, incluyendo:

- El **protocolo** (ej: `http`, `https`, `ftp`)
- El **dominio o IP**
- El **puerto** (opcional)
- La **ruta del recurso**
- Y a veces, **parámetros** y **fragmentos**

URL – Uniform Resource Locator

- Su propósito es **localizar** y acceder a un recurso. Cuando usas una API REST, trabajas principalmente con URLs

- Todas las URLs son URIs, pero no todas las URIs son URLs

- `https://api.miapp.com:8080/usuarios/5?activo=true#informacion`

URL – Uniform Resource Locator

https://api.miapp.com:8080/usuarios/5?activo=true#informacion

- Protocolo: `https`
- Dominio: `api.miapp.com`
- Puerto: `8080`
- `Ruta: /usuarios/5`
- `Parámetro: activo=true`
- `Fragmento: #informacion`

URN – Uniform Resource Name

- Un **URN** es una URI que **nombra** un recurso **sin decir dónde está ubicado ni cómo acceder a él**.
- Es más conceptual y **persistente** (no cambia con el tiempo)
- Su propósito es identificar un recurso por su **nombre dentro de un espacio de nombres estandarizado**
- urn:isbn:978-3-16-148410-0
- Este URN representa un libro por su número ISBN

Arquitectura Capas

@latecnologiaavanza

Separación de Responsabilidades

- La separación de responsabilidades es un **principio de diseño de software** que dice: "Cada módulo o capa del sistema debe encargarse de **una única responsabilidad**."
- Permite que el código sea más **organizado** y **mantenible**
- Cada parte del sistema sea más **reutilizable**
- Se facilite el **testeo** y el trabajo en equipo

Model - View - Controller (MVC)

Se utiliza mucho en aplicaciones **web tradicionales con interfaces gráficas** (como aplicaciones con HTML y JSP, o frameworks como Angular)

Componente	**Responsabilidad principal**
Model	Representa los datos (las entidades y lógica de negocio)
View	Interfaz de usuario (lo que ve el usuario)
Controller	Recibe la entrada del usuario y coordina acciones

Arquitectura en 3 capas

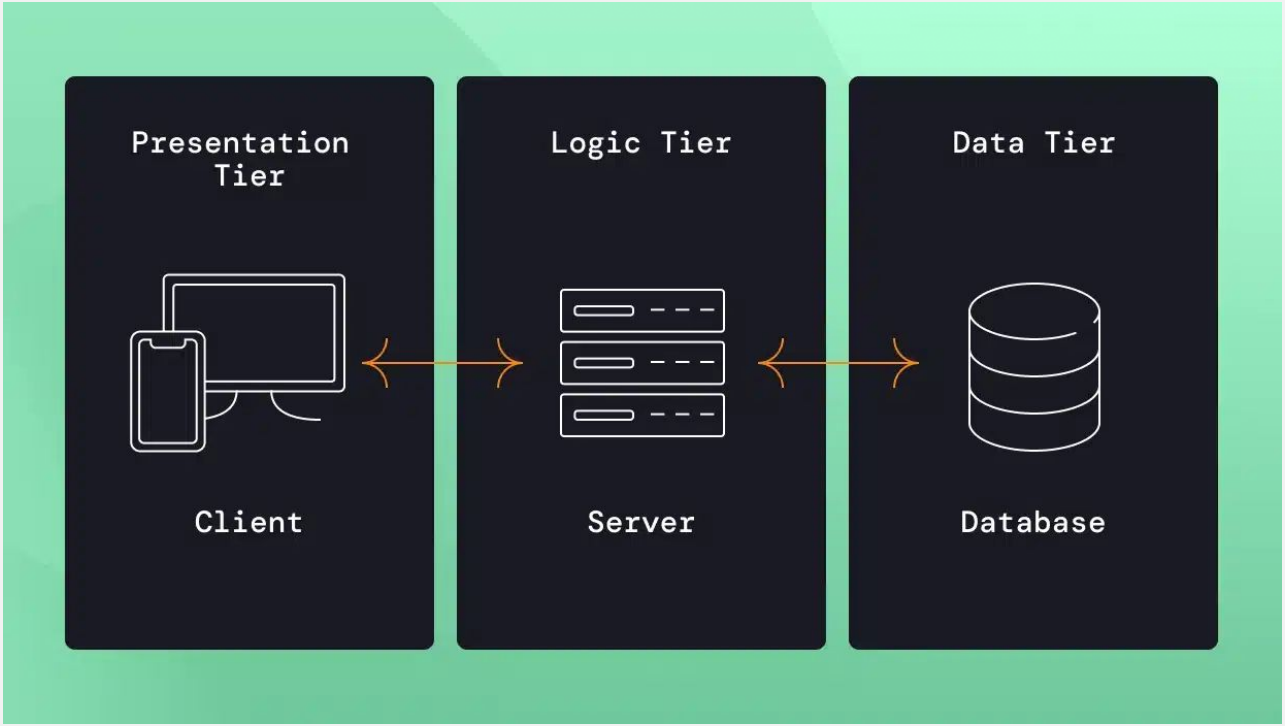
Es una forma de estructurar aplicaciones dividiendo el código en 3 capas bien diferenciadas:

Capa	**¿Qué hace?**
Controller	Recibe las solicitudes HTTP (como /usuarios) y delega a la capa de servicio
Service	Contiene la lógica del negocio (procesa, transforma, valida datos, etc.)
Repository	Se comunica directamente con la base de datos (usando JPA, Hibernate, etc.)

MVC y Arquitectura en 3 Capas

- En proyectos Spring Boot con APIs REST, **no usamos directamente View**, así que **no aplicamos el patrón MVC completo**
- Sin embargo, la arquitectura en capas **nace del mismo principio** de separación que MVC
- En lugar de View, tenemos respuestas en **formato JSON**
- Por lo tanto, usamos una **variación más adecuada**:→ Controller - Service - Repository (Modelo de 3 capas + DTOs)

Arquitectura en 3 capas



Estructura Proyecto

 @latecnologiaavanza

Estructura básica

```
src/main/java/com/miapp/
├── controller      # Controladores REST (entrada de datos)
├── service         # Lógica de negocio
│   └── impl        # Implementaciones de servicios
├── repository      # Interfaces JPA para acceder a la base de datos
├── dto             # Objetos de transferencia de datos
├── entity          # Entidades JPA (clases que representan tablas)
└── exception       # Manejo de errores personalizados
```

DTOs
ModelMapper


@latecnologiaavanza

DTO

- Un **DTO (Data Transfer Object)** es un **objeto simple usado para transportar datos** entre distintas capas de una aplicación, como entre el **cliente y el backend**, o entre la **capa de servicio y la capa de presentación**.

- Generalmente **no contiene lógica de negocio**, solo atributos (campos) y, a veces, validaciones

- Se usa principalmente para **entrada** (request) y **salida** (response) de datos en controladores REST o APIs

¿Por qué no exponer directamente la entidad?

- **Seguridad**: podrías mostrar campos sensibles (como contraseñas, tokens, roles internos, etc.)
- **Acoplamiento**: el cliente queda atado a la estructura interna de tu base de datos
- **Flexibilidad**: si cambias la entidad, puedes romper la API pública que consumen los clientes
- **Validación personalizada**: en los DTO puedes agregar anotaciones como `@NotNull`, `@Size`, `@Email`, etc., para validar lo que viene en la petición
- **Formato personalizado**: puedes cambiar nombres de campos, ignorar relaciones, o mostrar datos combinados como `nombreCompleto`

ModelMapper

ModelMapper es una **librería Java** que permite **mapear automáticamente objetos** entre sí. Es útil para convertir de:

- Entidad → DTO (por ejemplo, al enviar datos al frontend)
- DTO → Entidad (por ejemplo, al recibir datos del frontend)

Nuestra comunidad

@latecnologiaavanza

@latecnologiaavanza

latecnologiavanza

La Tecnología Avanza