Tema 3
Arquitectura REST

Arquitectura distribuida usando HTTP

PDS - Procesos de Desarrollo de Software 2025/26

Contenidos

1. Motivación.
2. Fundamentos de HTTP.
3. Arquitectura REST.
4. REST y Arquitecturas centradas en el dominio.
5. Diseño de APIS: Herramientas y buenas prácticas.

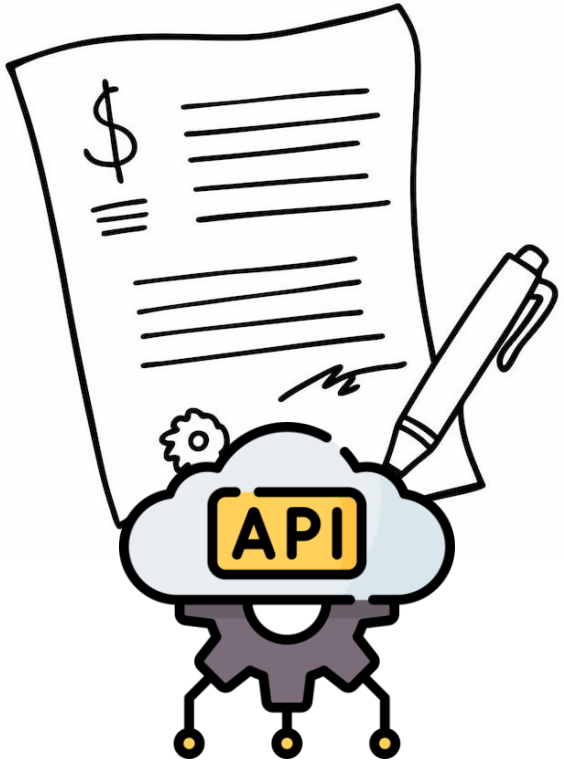
1. Motivación

¿Qué y para qué sirve una API REST?

1. Motivación – ¿Qué es una API?

Una API es el contrato público mediante el cual un sistema ofrece capacidades y datos a otros sistemas de forma controlada y estructurada.

1. **Es una puerta de entrada.**
   No accedes a la base de datos ni al código. Tú controlas el acceso.
2. **Es un contrato.**
   Define qué se puede hacer y cómo hacerlo.
3. **Ofrece capacidades (*no implementación*)**
   Publica lo que el sistema hace, no cómo lo hace.
4. **Permite desacoplar sistemas.**
   Mientras respeten el contrato, todo sigue funcionando.



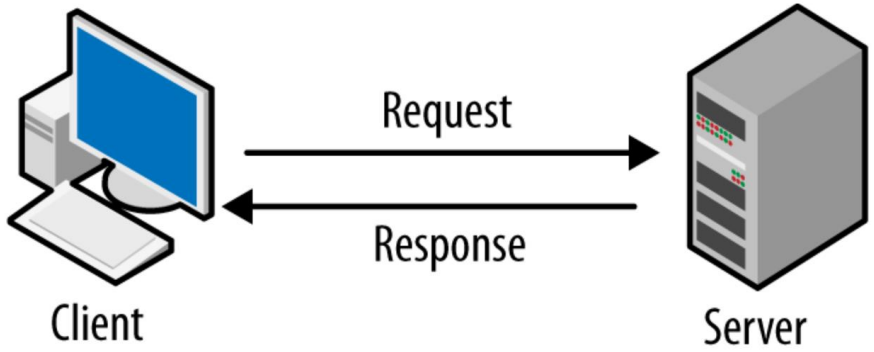
1. Motivación – Cliente-servidor: peticiones y respuestas

La arquitectura **cliente-servidor** es un modelo de comunicación donde un sistema solicita servicios y otro los proporciona.

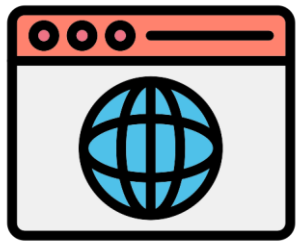
¿Cómo funciona?

Un sistema solicita servicios y otro sistema los proporciona.

1. **Request:** el cliente pide algo.
2. **Processing:** el servidor aplica reglas.
3. **Response:** devuelve resultado o error.



1. Motivación – Cliente-servidor: Ejemplo de conversación



```
GET / HTTP/1.1
Host: www.um.es
Accept: text/html
User-Agent: Mozilla/5.0
```

REQUEST
Un navegador web realiza una petición de una página web.

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Cache-Control: max-age=60
-----------------------------------------
<html>
  <head>
    <title>Universidad de Murcia</title>
  </head>
  <body>
    ...
  </body>
</html>
```

RESPONSE
La respuesta está formada por unas cabeceras con meta-datos y un cuerpo de respuesta.

1. Motivación – ¿Qué había antes?

Para implementar el modelo cliente–servidor en la web o en sistemas desacoplados se han utilizado distintas estrategias.

RPC (Remote Procedure Call)

1. Llamadas a funciones remotas
2. Enfoque basado en acciones
3. HTTP usado como canal de transporte

Servicios SOAP

1. Basados en XML
2. Contratos formales (WSDL)
3. Más complejos y pesados

1. Motivación – ¿Qué es una API REST?

Con SOAP y RPC la Web ya funcionaba como un sistema distribuido.

Muchos sistemas empezaron a usar HTTP solo como canal de transporte:

```
POST /crearUsuario
POST /actualizarDatos
POST /hacerOperacionX
```

Entonces aparece REST (Roy Fielding, 2000) con una idea clave.

Si la Web ya funciona bien…

¿Por qué no diseñar las APIs siguiendo sus principios?

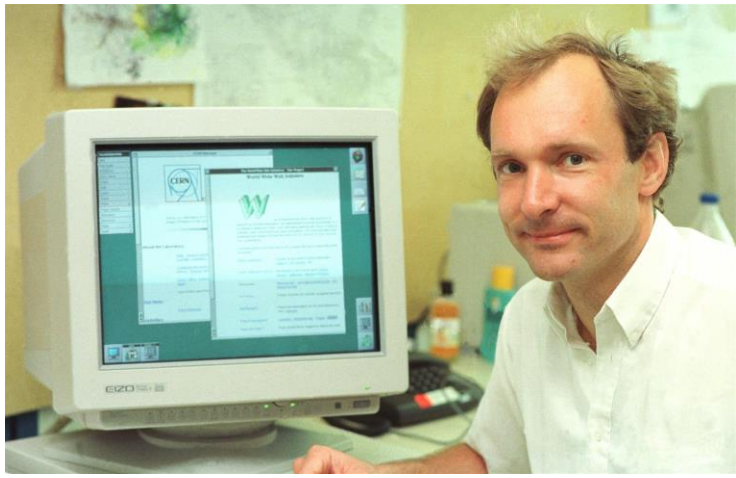
2. Fundamentos de HTTP

La arquitectura de la web

2. Fundamentos de HTTP – ¿Qué es?

HTTP (Hypertext Transfer Protocol) es el lenguaje de la Web.

- Permite que sistemas distribuidos se comuniquen.
- Define cómo se hacen peticiones y se reciben respuestas.
- Fue diseñado para ser simple y escalable.



Nace en 1.989 en el CERN por Tim Berners-Lee.

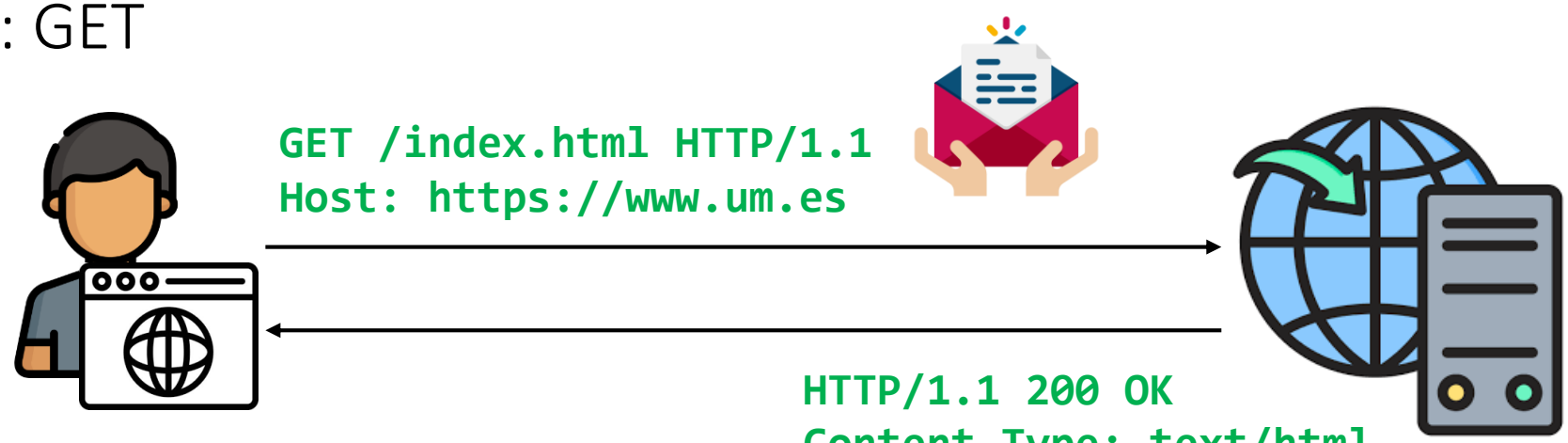
- Para compartir documentos científicos.
- Es la base de la WWW.

REST usa HTTP como idea para implementar APIs.

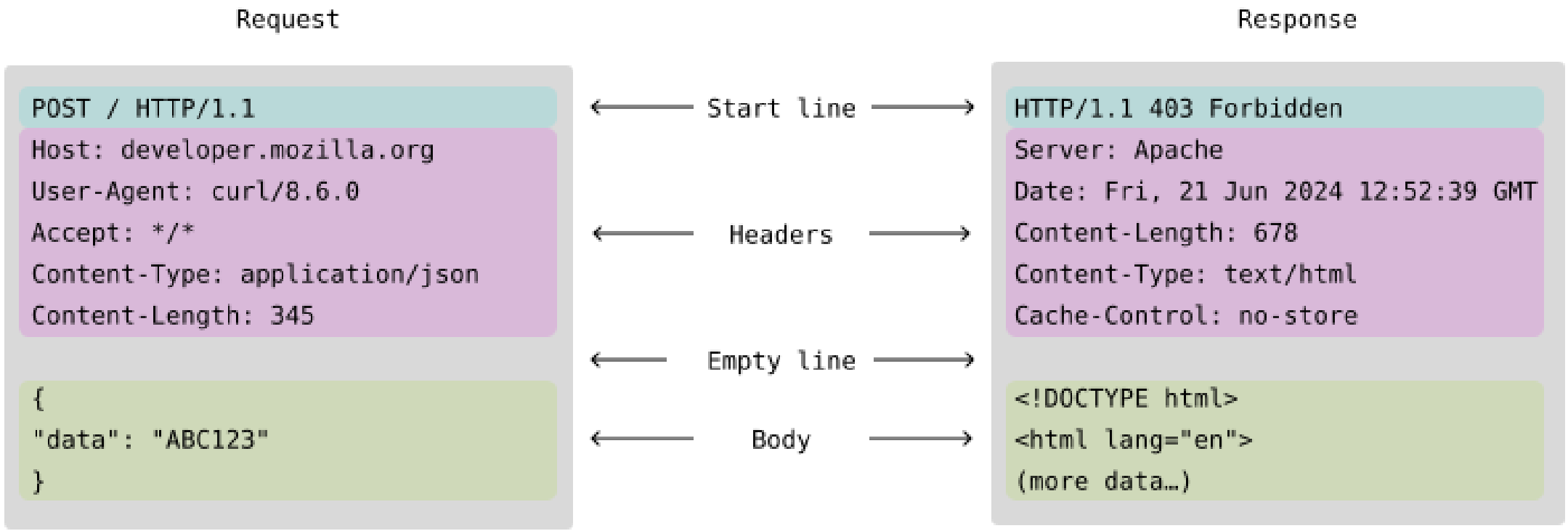
2. Fundamentos de HTTP – Estructura de mensajes

HTTP está basado en operaciones, llamadas verbos o métodos.

Ejemplo: GET



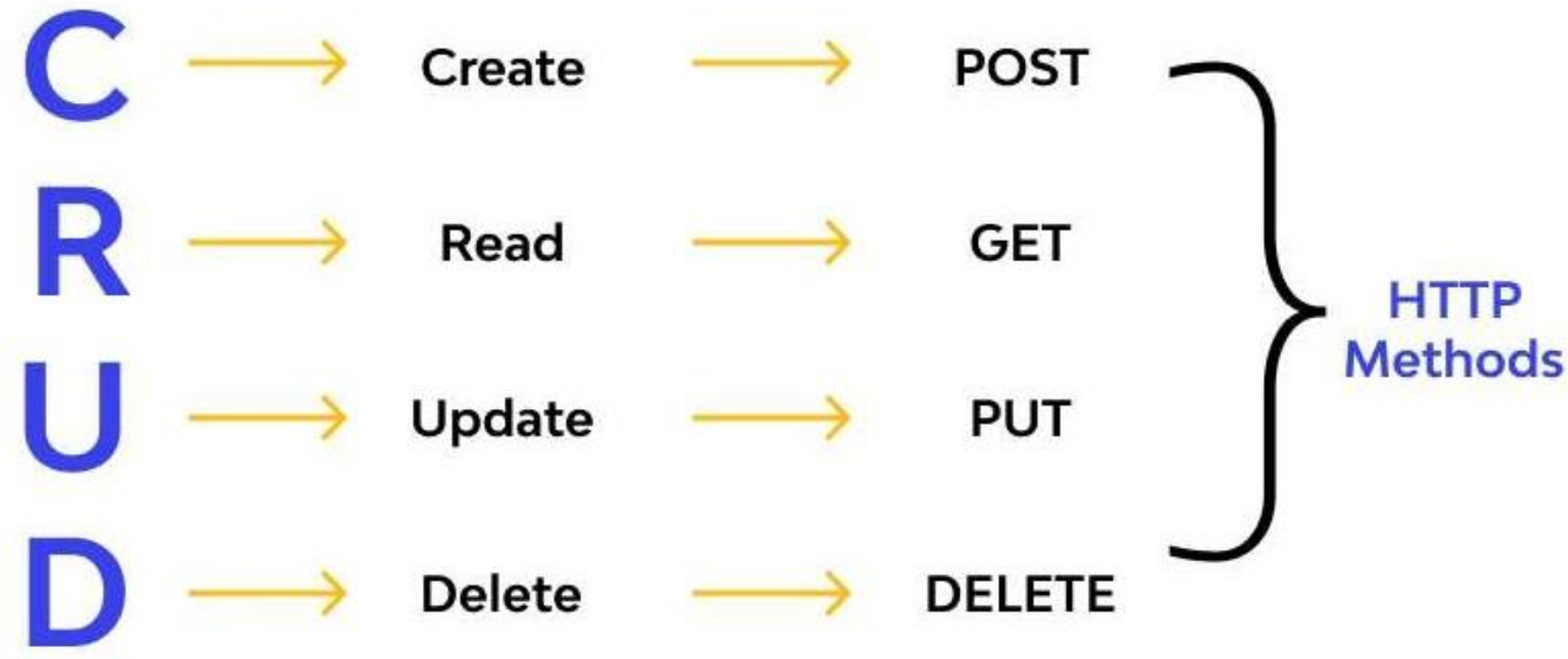
2. Fundamentos de HTTP – Formato mensajes



* https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Messages

2. Fundamentos de HTTP – Métodos

Los métodos HTTP permiten realizar las operaciones **CRUD**
(Create-Read-Update-Delete)



2. Fundamentos de HTTP – Semántica de los métodos

Método	Intención	Modifica estado	Idempotente
GET	Obtener recurso	✗	✓
POST	Procesar / crear	✓	✗
PUT	Reemplazar recurso completo	✓	✓
PATCH	Modificar parcialmente	✓	✗
DELETE	Eliminar recurso	✓	✓

2. Fundamentos de HTTP – Idempotencia

La idempotencia consiste en ejecutar la misma petición varias veces produce el mismo resultado final que ejecutarla una sola vez.

```
PUT /usuarios/1
{"nombre": "Ana"}
```

Da igual el número de veces que lo llames, el nombre siempre es “Ana”.

```
DELETE /usuarios/1
```

Da igual el número de veces que lo llames, el recurso seguirá eliminado.

```
POST /pedidos
{"producto": "portatil",
"cantidad": 1, "precio":
899.9 }
```

Si lo llamas 5 veces, creas 5 recursos distintos.

2. Fundamentos de HTTP – Otros métodos útiles

Otras operaciones adicionales reflectivas útiles.

HEAD

- Igual que GET, pero solo devuelve el header (sin el body)
- Útil para comprobar existencia (llamada ligera) o meta-datos.

OPTIONS

- Indica qué métodos están permitidos para un recurso.
- Muy usado en CORS y APIs modernas.

TRACE

- Se usa para diagnosticar qué cambios han hecho servidores intermedios (firewalls, proxies, etc).

2. Fundamentos de HTTP – Negocio de mensajes

HTTP no solo transporta datos sino permite negociar cómo se intercambian.

Lo que te mando.

```
Content-Type: application/json
```

Lo que espero recibir.

```
Accept: application/json
```

¿Quién soy? (Autenticación y autorización)

```
Authorization: Basic dXN1YXJpbzpwYXNzd29yZA== (BASE64)
```

* https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Messages

2. Fundamentos de HTTP – Peticiones

Petición HTTP - POST

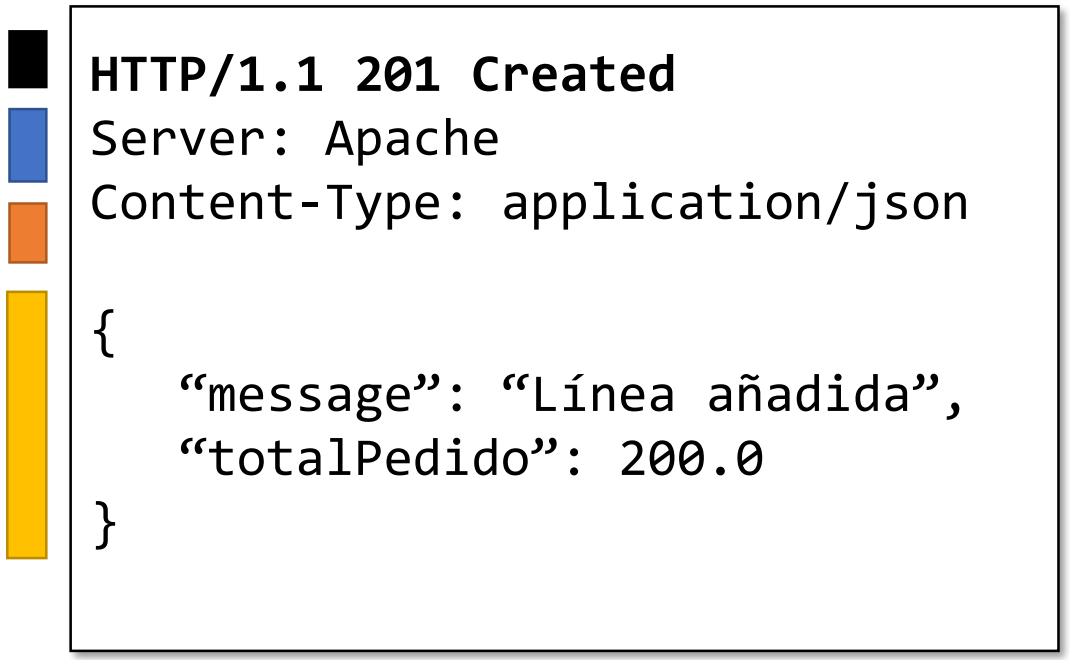
```
POST /pedido/37/linea HTTP/1.1
Host: mi-tienda-online.es
Accept: */*
Content-Type: application/json
Content-Length: 252

{
   “productoId”: 42
   “cantidad”: 2
}
```

- start-line:
  - Verbo
  - Path del recurso
  - Versión de HTTP
- Cabeceras
  - (Meta-datos sobre el mensaje)
  - Request headers
    - Información de la petición o sobre cómo debe tratarla el servidor
  - Representation headers
    - Si el mensaje tiene un cuerpo
    - Información sobre qué formato tiene
- Cuerpo
  - Solo PUT, POST, PATCH

2. Fundamentos de HTTP – Respuestas

Respuesta HTTP - POST



```
HTTP/1.1 201 Created
Server: Apache
Content-Type: application/json

{
   “message”: “Línea añadida”,
   “totalPedido”: 200.0
}
```

- start-line:
  - Versión de HTTP
  - **Código de respuesta**
- Cabeceras
  - (Meta-datos sobre el mensaje)
  - Request headers
    - Información sobre cómo ha sido tratada la petición
  - Representation headers
    - Información sobre qué formato tiene la respuesta
- Cuerpo
  - Datos de la respuesta

2. Fundamentos de HTTP – Códigos de respuesta

HTTP no solo transporta datos sino permite negociar cómo se intercambian.

2XX – Éxito

- 200 OK → petición correcta
- 201 Created → recurso creado
- 204 No Content → correcto, sin cuerpo

3XX – Redirección

- 304 Not Modified → usar versión en caché

https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status

2. Fundamentos de HTTP – Códigos de respuesta

HTTP no solo transporta datos sino permite negociar cómo se intercambian.

4XX – Error del cliente

- 400 Bad Request → petición mal formada
- 404 Not Found → recurso no existe
- 409 Conflict → conflicto de negocio

5XX – Error del servidor

- 500 Internal Server Error
- 501 Not Implemented

https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status

3. Arquitectura REST

3. Arquitectura REST – Representational State Transfer

Estilo arquitectónico para aplicaciones distribuidas inspirado en los principios de la web.

Ideado y propuesto por **Roy Fielding (2000)**

- Tesis doctoral: https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf

Es un estilo arquitectónico.

No es un protocolo, librería o framework, sino un **conjunto de restricciones arquitectónicas**.

En teoría… No depende de HTTP, JSON o XML

Pero en la práctica… Se implementa casi siempre sobre HTTP y JSON.

3. Arquitectura REST – ¿Cómo se organiza?

Los servicios REST se organizan como un sitio web.

Sin embargo, en lugar de retornar recursos web (HTML), se obtiene una representación estructurada de los recursos (XML, JSON).

Acceso uniforme a los recursos:

- Todo recurso tiene una URL que lo identifica.
- Los recursos son accedidos con peticiones HTTP (GET, POST, etc.)
- Los recursos se relacionan mediante enlaces.

3. Arquitectura REST – Recursos

Un servicio REST representa una colección de recursos.

Tipos de recursos

- Recurso individual (ej. Producto de una tienda online)
- Colecciones de recursos individuales de un mismo tipo.

Todo recurso (individual o colección) tiene una URL.

- El acceso a la URL permite obtener una representación estructurada del recurso.
- Si el recurso es una colección, se obtiene un resumen de cada recurso que contiene y un enlace al recurso individual.

3. Arquitectura REST – Recursos

Recursos

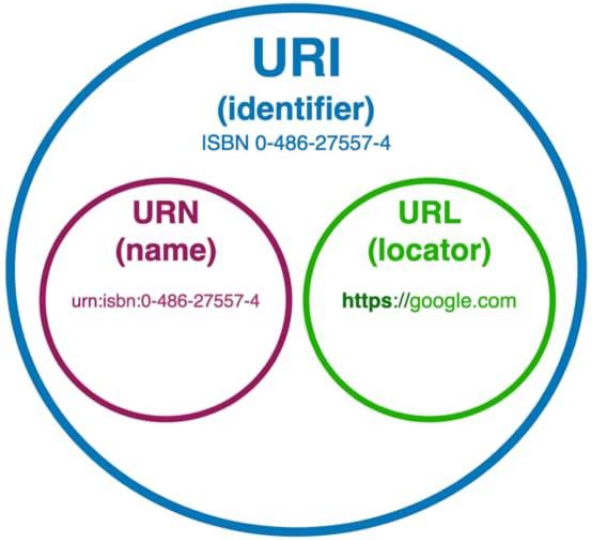
Todo se modela como **recursos identificables**.

- Cada recurso tiene una URI.

```
GET /usuarios
GET /usuarios/123
GET /usuarios/123/pedidos
```

- No hablamos de acciones sino de cosas.

```
GET /usuarios > GET /obtenerUsuarios
```



3. Arquitectura REST – Recursos

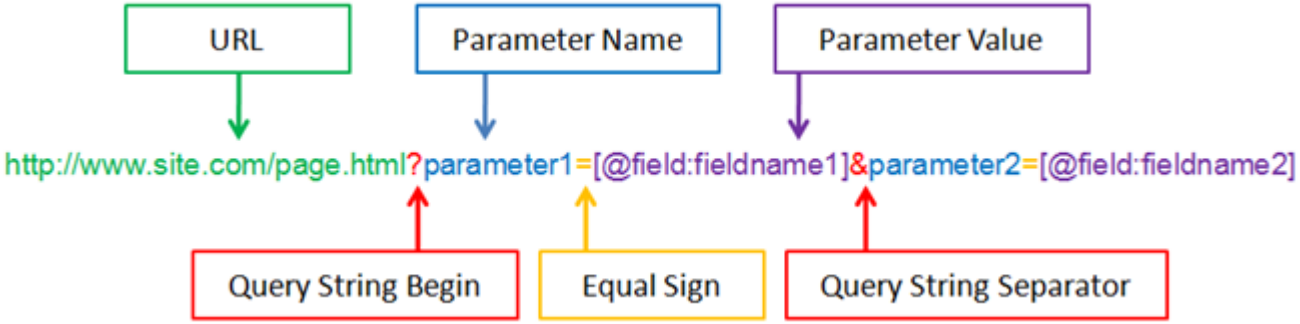
Recursos – Query strings

Permite modificar la consulta sin cambiar el recurso base.

- No crea un recurso nuevo.
- Describe una variante de la representación.
- Es parte de la URI (se puede cachear)

Usos

- Filtrar y ordenar
- Paginar



3. Arquitectura REST – Interfaz uniforme

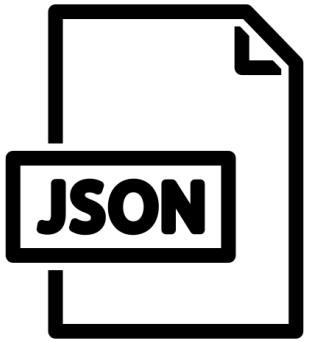
El estilo arquitectónico REST exige que la interfaz de acceso a los servicios sea uniforme.

- En general, se usa HTTP.
- El acceso es mediante operaciones HTTP: **GET**, **POST**, etc.
- Las operaciones HTTP tienen una semántica común:
  Ejemplo: la petición **GET** retorna la representación de un recurso y no causa efectos laterales.
- Respuesta estandarizada de las operaciones.
  Por ejemplo, códigos HTTP: 404 Not Found, 200 OK

3. Arquitectura REST – Representaciones

Representaciones

La información que se devuelve incluye datos, meta-datos y puede tener más de un formato.



- Representaciones.
  `Accept: application/json` o `Accept: application/xml`
- Se pueden devolver datos y meta-datos.

3. Arquitectura REST – Restricciones

Hay una serie de restricciones que definen el estilo arquitectural.

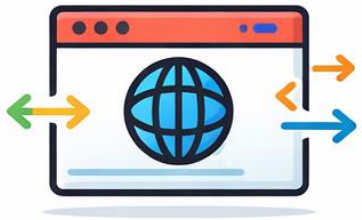
Deben cumplirse para que el servicio sea considerado RESTful.

1 Cliente–Servidor

2 Stateless

3 Cacheable

4 Interfaz uniforme



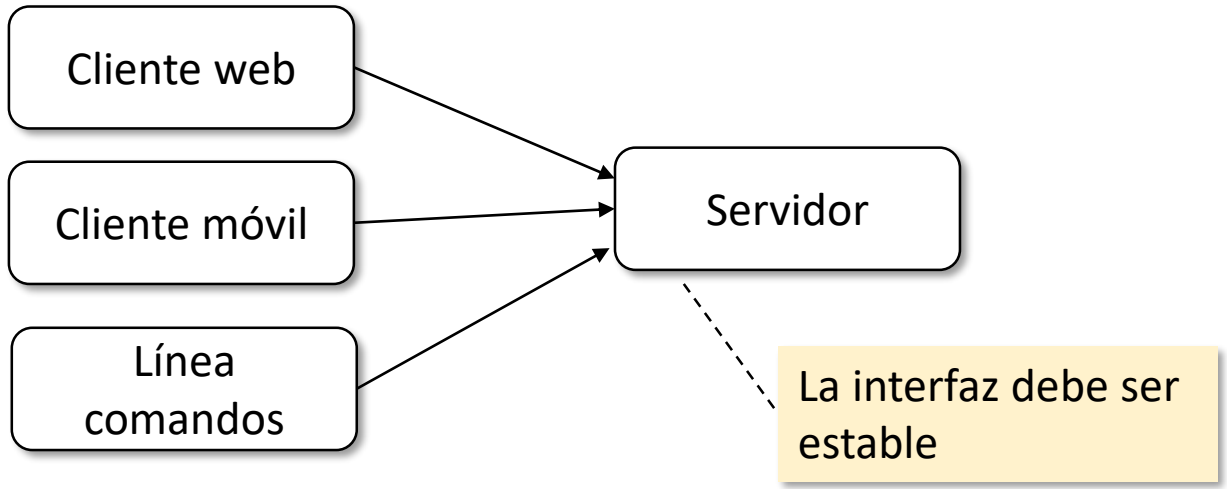
5 Sistema en capas

6 Code-on-Demand

3. Arquitectura REST – Restricciones- Cliente/servidor

Utilizar la arquitectura cliente/servidor favorece la separación de responsabilidades (separation of concerns)

- Servidor se encarga del backend.
  almacenamiento, reglas de negocio
- Cliente maneja el frontend.
  interfaz de usuario, experiencia de usuario.
- Servidores y clientes pueden evolucionar y escalar de manera independiente.



3. Arquitectura REST – Restricciones - Stateless

Cada petición debe contener toda la información para que el servidor pueda procesarla sin depender de la conversación anterior.

Cada petición debe contener toda la información para que el servidor pueda procesarla sin depender de la conversación anterior.

- El servidor no debe almacenar información sobre el estado de la conversación actual.
- **Razones:** visibilidad, fiabilidad y escalabilidad.

3. Arquitectura REST – Restricciones- Cacheable

El cliente debe poder cachear respuestas para evitar hacer nuevas llamadas para obtener recursos que no han cambiado.

- El uso de caché es importante tenerlo en mente a la hora de definir nuestra API REST.
- Para ello las respuestas deben ir marcadas con cacheable o no cacheables.
- El servidor es quién decide si las respuestas con cacheables.

3. Arquitectura REST – Restricciones – Interfaz uniforme

El sistema debe usar el mismo tipo de interfaz y de operaciones.

- Identificación de recursos por URI.
- Mensajes auto descriptivos.
- Navegabilidad (HATEOAS).
- Siempre usamos los mismos verbos HTTP.

```
GET /usuarios/123; PUT /usuarios/123 ; DELETE /usuarios/123; POST /usuarios
```

- La petición explica cómo debe procesarse:

```
Content-Type: application/json
```

3. Arquitectura REST – Restricciones – Organización en capas

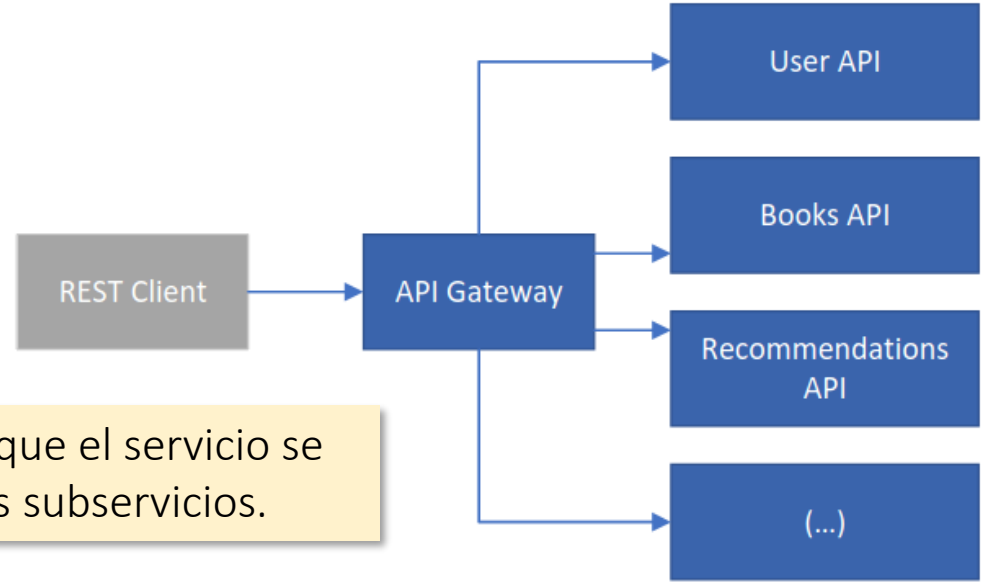
5 Sistema en capas

Un componente no puede ver lo que hay detrás de otros componentes a los que invoca.

El cliente no sabe si está comunicándose con el servidor real o con un proxy gracias a la interfaz uniforme.

Razones:

- Extensibilidad
- Escalabilidad



El cliente no sabe que el servicio se divide en multiples subservicios.

3. Arquitectura REST – Restricciones – Code-on-demand

El servidor puede enviar código ejecutable al cliente para ampliar su funcionalidad.

Ejemplo

- Servidor devuelve una página HTML que incluye JavaScript.
- Ese JavaScript se ejecuta en el navegador y…
  - valida formularios, renderiza gráficos, o hace nuevas peticiones.
- Es opcional (La única restricción opcional en REST).
- En APIs modernas casi nunca se usa.

Puede introducir riesgos de seguridad.

3. Arquitectura REST – Buenas prácticas – Versiones

Las APIs evolucionan.

- Nuevos campos
- Cambios de formato
- Eliminación de atributos
- Nuevas reglas

Versión por URL

```
GET /v1/usuarios
GET /v2/usuarios
```

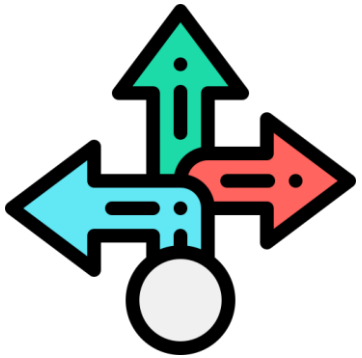
Versión por cabecera

```
Accept: application/vnd.miapi.v1+json
```

3. Arquitectura REST – Buenas prácticas – HATEOAS

HATEOAS (Hypermedia As The Engine Of Application State)

- Principio de diseño de APIs REST.
  Técnicamente, en la tesis de Fielding forma parte de la interfaz uniforme.
- El servidor incluye enlaces en las respuestas.
- Una API puede indicar cómo continuar la interacción.



```
> GET /usuarios?page=1
    {
      "data": [...],
      "links": {
        "self": "/usuarios?page=1",
        "next": "/usuarios?page=2",
        "prev": null
      }
    }
```

3. Arquitectura REST – Buenas prácticas – Errores

No basta por devolver mensajes de error (404, 500, …)

Buenas prácticas

- Usar códigos HTTP correctamente.
- No usar 500 para errores de negocio.
- Mantener formato consistente.
- No exponer detalles internos.

```
{
  "error": {
    "code": "USUARIO_NO_ENCONTRADO",
    "message": "El usuario no existe",
    "details": []
  }
}
```

3. Arquitectura REST – Buenas prácticas – Seguridad

- No exponer identificadores secuenciales.

✔ `/usuarios/9f8a2c7b-41e3-4c8f` ✘ `/usuarios/124`

- Usar HTTPs siempre.
  Protege credenciales, tokens y datos sensibles.
- No devolver más información de la necesaria.

```
{
  "id": 123,
  "nombre": "Ana",
  "passwordHash": "...",
  "rolInterno": "ADMIN“
}
```

4. REST y Arquitecturas centradas en el dominio.

4. REST y DDD – ¿Qué problema resuelven?

REST

Comunicación

- Expone funcionalidades a través de HTTP.
- Integración de sistemas.
- APIs públicas.
- Interoperabilidad.

DDD

Complejidad

- Modelar lógica de negocio compleja.
- Protege las reglas del dominio.
- Reduce ambigüedad conceptual.
- Evita modelos anémicos.

4. REST y DDD – Incompatibilidades

- REST es orientado a recursos (estado).
  REST quiere `POST /pedido`
- DDD es orientado a comportamiento (acciones y reglas).
  DDD quiere `pedido.crear ()`

Si tratamos de alinearlos…

Opción 1. Hacer el diseño DDD orientado a recursos.

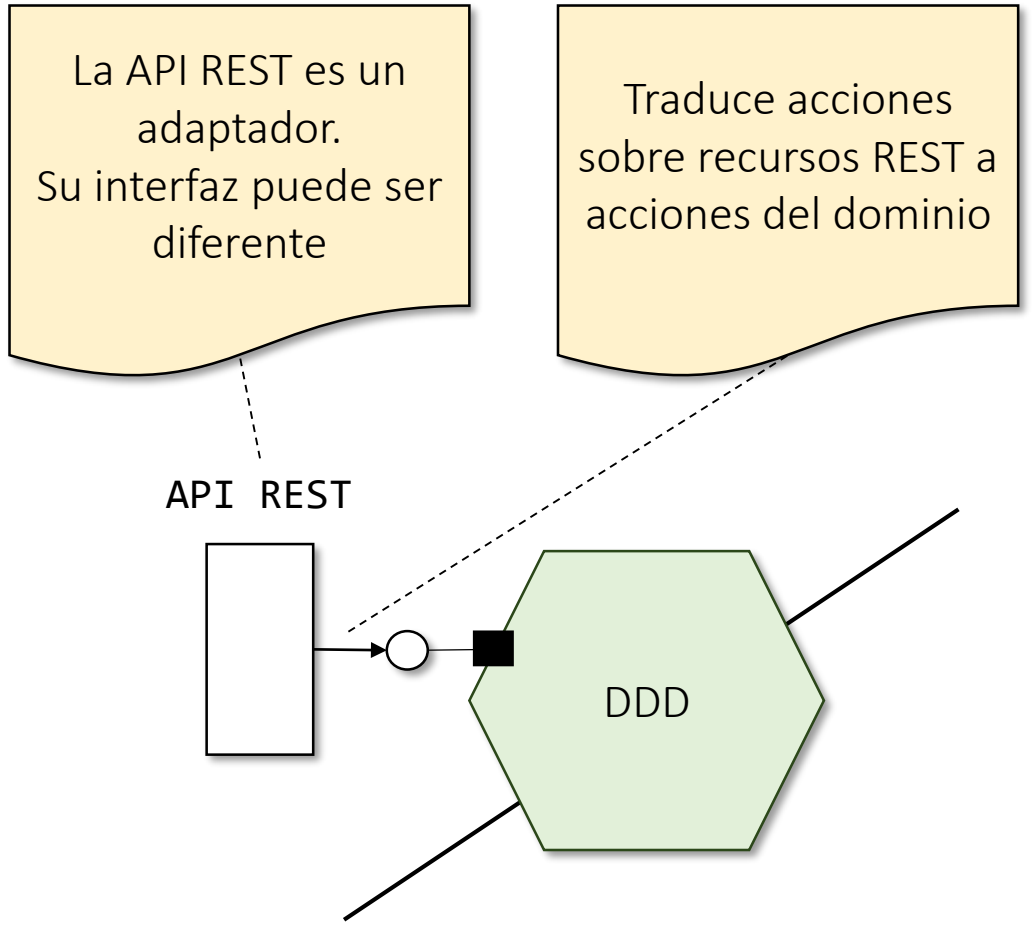
Nos lleva a modelos del dominio anémicos.

Opción 2. Programar la API REST con acciones

Pero las APIs no están orientada a recursos.

4. REST y DDD – Soluciones

- Evitar mapeo directo.
  - No tener solo operaciones CRUD
- La API REST expresa las acciones posibles utilizando un estilo Restful.
- El controlador las traduce a invocaciones de los servicios de aplicación DDD.



4. REST y DDD – Soluciones

(1) Para servicios de aplicación que parten de un agregado.

- Crea una ruta como `/<agregado>/:id`
- Para las operaciones CRUD *(sólo si hay)* – POST|PUT|GET `/<agregado>/:id`
- Para las acciones de negocio – POST|PUT `/<agregado>/:id/<acción>`

```
POST /suscripcion/37/cancelacion

try {
   servicioSuscripciones.cancelar(id);
} catch (CancelacionNoPosible e) {
   // Devolver código de error adecuado
}
return 201, OK
```

```
public class ServicioSuscripciones {
  public void cancelar(SuscripcionId id) {
    var suscp = repoSuscripciones.findById(id);
    suscp.aplicarCancelacion();
    repoSuscripciones.save(suscp);
  }
}
```

4. REST y DDD – Soluciones

(2) Para servicios de aplicación que no parten de un agregado

- Crear una ruta **POST** `/<nombre-acción>`

```
POST /procesar-pago
POST /generar-informe
POST /importar-datos
```

- El controlador redirige al servicio de aplicación correspondiente.
- El controlador devuelve el resultado adecuado.

5. Diseño de APIS

Herramientas y buenas prácticas.

Algunas herramientas para el desarrollo de API RESTs

5. Diseño de APIs – Motivación

Múltiples alternativas para implementar APIs para distintos lenguajes.

¿Qué aportan?

- Anotaciones para controladores REST.
- Serialización automática JSON.
- Integración con seguridad.
- Integración con OpenAPI.
- Gestión de dependencias simplificada.

5. Diseño de APIs – SpringBoot

SpringBoot es el framework Java que vamos a usar durante esta asignatura.

- Facilita el desarrollo de API RESTs.
- Dispone de un servidor web embebido.
- Parsea peticiones HTTP y mapea rutas automáticamente.
- Serializan/deserializan JSON.
- Ecosistema de plug-ins
  plantillas, validación y seguridad, envío de emails, JPA, etc.

5. Diseño de APIs – SpringBoot

Ejemplo con Spring boot.

```
GET /usuarios/u_8f3a92
```


```java
@RestController
@RequestMapping("/usuarios")
public class UsuarioController {

  @GetMapping("/{id}")
  public UsuarioDTO obtener(@PathVariable String uid) {
    return servicio.buscar(uid);
  }
}
```

5. Diseño de APIs – SpringBoot – Anotaciones

Varios frameworks usan sistemas de anotaciones para facilitar el desarrollo.

- Una anotación es una marca declarativa en el código que proporciona metainformación al framework.
- No ejecuta lógica directamente.
- Indica cómo debe comportarse el sistema.

HTTP	Spring
Ruta base	@RequestMapping
GET	@GetMapping
POST	@PostMapping
PUT	@PutMapping
DELETE	@DeleteMapping
Parámetro en URL	@PathVariable
JSON en cuerpo	@RequestBody
Controlador REST	@RestController

5. Diseño de APIs – SpringBoot – Ejemplo

```java
@RestController
@RequestMapping("/v1/pedidos")
public class ControladorPedidos {

  private final ServicioPedidos servicio;

  public ControladorPedidos(ServicioPedidos servicio) {
    this.servicio = servicio;
  }
}
```

En este ejemplo, veremos un controlador para tener una API REST de Pedidos.

- Creamos una clase Controlador.
- `@RestController` indica que devuelve JSON.
- `@RequestMapping` define la ruta base.
- Usamos **`ServiciosPedido`** para encapsular el acceso a los items (mediante inyección de dependencias).

5. Diseño de APIs – SpringBoot – Ejemplo - GetMapping

```
// GET http://localhost:8080/v1/pedidos/{uid}
@GetMapping("/{uid}")
public PedidoDTO obtener(@PathVariable String uid) {
  return servicio.obtener(uid);
}

// GET http://localhost:8080/v1/pedidos?page=0&size=20
@GetMapping
public List<PedidoDTO> listar(
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "20") int size) {
    return servicio.listar(page, size);
}
```

- `{uid}` identifica al recurso (es un UID, no un ID secuencial).
- `@PathVariable` extrae el valor de la URL.
- Devuelve automáticamente 200 OK.
- No modifica el estado (idempodente).

5. Diseño de APIs – SpringBoot – Ejemplo- PostMapping

```
// POST http://localhost:8080/v1/pedidos
@PostMapping
public PedidoDTO crear(@RequestBody CrearPedidoDTO body) {
    return servicio.crear(body);
}

// Alternativa: valor de retorno explícito
@PostMapping
public ResponseEntity<PedidoDTO>
       crear(@RequestBody CrearPedidoDTO body) {
  PedidoDTO nuevoPedido = servicio.crear(body);
  return ResponseEntity.status(HttpStatus.CREATED).
         body(nuevoPedido); }
}
```

- `@RequestBody` mapea JSON a un objeto Java.
- POST permite crear un nuevo recurso.
- El servidor genera el UID.
- No es idempotente.

5. Diseño de APIs – SpringBoot – Ejemplo - DeleteMapping

```
// DELETE http://localhost:8080/v1/pedidos/{uid}
@DeleteMapping("/{uid}")
public void eliminar(@PathVariable String uid) {
  servicio.eliminar(uid);
}
```

- Elimina el recurso identificado.
- Es idempotente.
- Devuelve `204 No Content` automáticamente.
- No devuelve cuerpo.

5. Diseño de APIs – Buenas prácticas – Contrato.

¿Qué es el contrato?

Es la definición **pública y estable** de:

Endpoints y métodos, parámetros, respuestas y códigos de estado, formatos, errores o seguridad.

Un contrato estable permite:

- Evolucionar sin romper.
- Trabajar en paralelo.
- Versionar correctamente.
- Garantizar compatibilidad hacia atrás.

5. Diseño de APIs – Buenas prácticas – Autenticación

Autenticación *¿Quién eres?*

El cliente demuestra su identidad.

- Se puede hacer con usuario y contraseña (Basic Auth)
- Tokens

Autorización *¿Qué puedes hacer?*

El servidor decide si tienes permiso para acceder al recurso.

- Ejemplo
  Usuario autenticado, pero no puede borrar pedidos.
  `403 Forbidden`

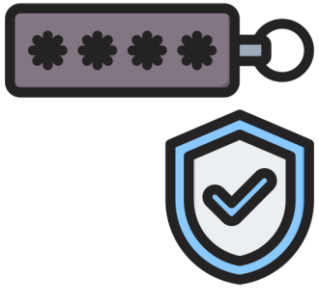
5. Diseño de APIs – Buenas prácticas – Autenticación

Autenticación

- **HTTP es stateless**
  - Cada petición es independiente.
  - Hay que enviar los datos de autenticación en cada llamada.
- **HTTP Basic no está cifrado (Base64 solo codifica)**
  - Si no se usa HTTPs, la contraseña está visible en la red.
- **Difícil de revocar y de hacer una expiración automática.**
  - No hay caducidad.

5. Diseño de APIs – Buenas prácticas – Autenticación

Autenticación mediante tokens



- El usuario se autentica (login).
- El servidor genera un token y se lo da al usuario.
- El cliente lo envía en cada petición.

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

Ventajas de usar tokens

Se pueden firmar digitalmente

Puede contener permisos

Se pueden revocar y expirar.

Implementaciones potentes.

5. Diseño de APIs – Buenas prácticas- Caché

HTTP permite reutilizar respuestas para evitar peticiones innecesarias.

- Ideales para recursos que cambian poco.
- Reduce latencia y la carga en el servidor.

Estrategias

Por tiempo

```
Cache-Control: max-age=3600
```

(válido para una hora)

Versionado con ETAG

- *El servidor puede etiquetar recursos*
  `ETag: "v3"`
- El cliente puede preguntar después
  `If-None-Match: "v3"`
- *El servidor puede deolver*
  `304 Not mofidied`

5. Diseño de APIs – Buenas prácticas - Documentación

- **OpenAPI**

Estándar para describir APIs REST de forma estructurada.

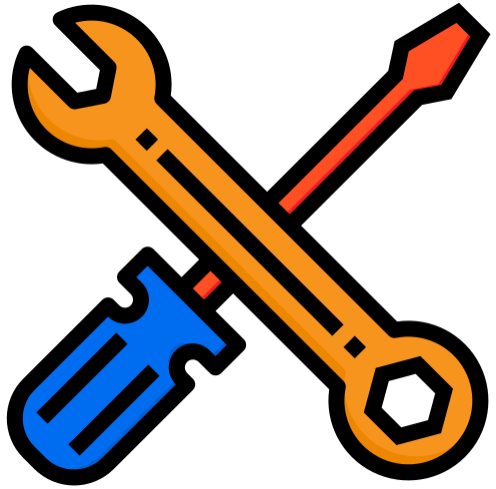
- **Swagger**

  Conjunto de herramientas que implementan OpenAPI

- Documentación sincronizada con el código
- Ejemplos reales de request/response.
- Una API sin documentación no es usable.

5. Diseño de APIs – Herramientas

- Diseño y construcción de APIs.
- Contrato y documentación.
- Pruebas manuales.
- Consumo programático.
- Seguridad.
- Monitorización.



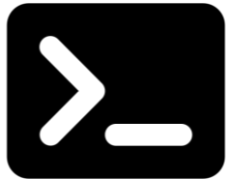
5. Diseño de APIs – Herramientas – Validación

¿Para qué sirven?

- Verificar rápido que un endpoint funciona.
- Depurar errores (códigos, cabeceras, body).
- Probar casos límite sin escribir código.
- Explorar la API durante el desarrollo.

Herramientas típicas

(GUI)
Postman
https://www.postman.com/



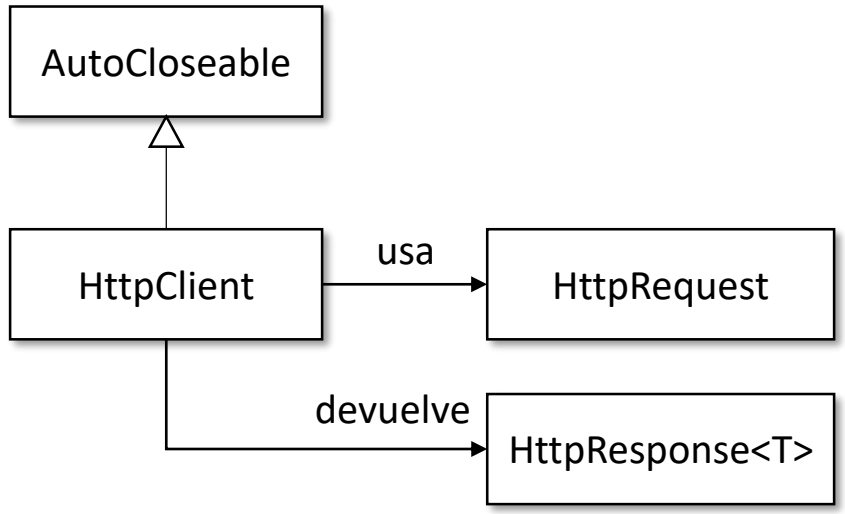
(CLI)
cURL
https://curl.se/

(Browser)
DevTools
https://firefox-dev.tools/

5. Diseño de APIs – Herramientas – Cliente HTTP

Cliente HTTP de Java

- Disponible desde Java 11.
- Soporta HTTP/1.1 y HTTP/2.
- Permite peticiones síncronas y asíncronas.
- API moderna basada en un Builder.




```
try (HttpClient client = HttpClient.newHttpClient()) {
  HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://api.vatlookup.eu/rates/ES"))
    .GET()
    .build();

  HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

  response.statusCode()
  response.body()
}
```

5. Diseño de APIs – Herramientas – Seguridad

Distintos niveles de herramientas de seguridad.

Spring Security

- Integración con Spring Boot.
- Distintas técnicas de autenticación.
- Control de acceso por roles.

Librerías JWT

- Generar tokens.
- Firmarlos.
- Validarlos.

OAuth2

- Herramientas para delegar la autenticación.
- Gestión centralizada de usuarios.

5. Diseño de APIs – Herramientas – Monitorización

Permite medir el comportamiento de una API en ejecución.

¿Para qué sirve?

- Detectar errores en producción.
- Medir rendimiento (latencia).
- Saber cuántas peticiones recibe.
- Detectar cuellos de botella.
- Ver cuándo algo se cae.
- IDEA: ELK Stack para monitorizar logs.

EJEMPLO

https://monitorumu.um.es/

Referencias

- Architectural Styles for Network-based Applications. *Roy Fielding (2000)*
  https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf
- IETF – HTTP Semantics (RFC 9110)
  https://www.rfc-editor.org/rfc/rfc9110
- MDN Web Docs (HTTP).
  https://developer.mozilla.org/es/
- OpenAPI Specification
  https://spec.openapis.org/
- REST explicado de forma didáctica
  https://kennethlange.com/the-little-book-on-rest-services/