# Arquitectura del Software

Jesús Sánchez Cuadrado
Última actualización: 17/02/2026

Este tema se introduce el concepto de arquitectura de software y su importancia en el desarrollo de sistemas. Se presentaran los elementos fundamentales de la arquitectura de software, así como los estilos arquitectónicos más comunes.

Esta primera versión contiene:

- Breve introducción a la arquitectura de software.
- Arquitectura Hexagonal (Ports & Adapters) y su relación con Domain-Driven Design (DDD).
- Arquitectura REST y el protocolo HTTP.

## Índice



## 1. Introducción

La arquitectura del software es la estructura fundamental de un sistema de software, compuesta por sus componentes, las relaciones entre ellos y los principios y decisiones que guían su diseño y evolución.

En otras palabras, describe cómo está organizado el software y cómo interactúan sus partes para cumplir los requisitos funcionales y no funcionales (rendimiento, seguridad, escalabilidad, mantenibilidad, etc.).

En la arquitectura el foco ya no son unidades pequeñas del código (ej., clases) sino que el foco está puesta en unidades más grandes (ej., módulos, servicios, capas, componentes). Por tanto, la arquitectura tiene un granularidad mayor que el diseño de software.

Los elementos implicados en la arquitectura de software son:

- Paquetes
- Módulos
- Componentes
- Capas
- Servicios
- Sistemas y subsistemas

### 1.1. Definiciones de arquitectura del software

Existen multitud de definiciones de arquitectura del software, más o menos formales. A continuación se presentan algunas de las más conocidas.

#### 1.1.1. Basada en las relaciones entre los elementos

> La arquitectura del software de un sistema es el conjunto de estructuras necesarias para razonar sobre el sistema, las cuales comprenden los elementos de software, las relaciones entre ellos y las propiedades de ambos."
>
> – Software Architecture in Practice, Bass, Clements y Kazman (2013)

### 1.1.2. Basada en las decisiones principales

> "La arquitectura del software trata de las decisiones importantes, aquellas que son difíciles de cambiar."
>
> – Philippe Kruchten

> "Architecture is about the important stuff. Whatever that is."
>
> – Ralph Johnson

Esta definición pone el foco en la arquitectura refleja las decisiones críticas del proyecto, aquellas que deben ser correctas desde el principio del proyecto.

Esencialmente, la arquitectura del software se refiere a las decisiones fundamentales sobre la estructura y organización de un sistema de software y el impacto de estas decisiones.

## 1.2. Elementos de la arquitectura del software

Por tanto, la **arquitectura del software** es el conjunto de estructuras necesarias para razonar sobre un sistema software (a alto nivel). Cada estructura está formada por elementos software, relaciones entre ellos y ciertas propiedades de los elementos y relaciones. Estas estructuras principales reflejan las decisiones críticas del proyecto, aquellas que deben ser correctas desde el principio del proyecto.

"Los planos del software", que incluyen:

- Estructura
- Comportamiento
- Interacciones
- Propiedades no funcionales

La arquitectura del software tiene como objetivo abordar los requisitos no funcionales. La arquitectura del software trata con los factores de calidad del software (ver Introducción), como por ejemplo si el software es extensible, tolerante a fallos, seguro, etc.

Pero, ¿por qué los requisitos no funcionales deben abordarse a nivel de arquitectura del software? Por que los requisitos funcionales se pueden abordar agregando, modificando o eliminando código. Sin embargo, los requisitos no funcionales suelen ser "cross-cutting concerns", esto es, afectan a múltiples partes del sistema. Por tanto, requieren una planificación y diseño cuidadosos a nivel arquitectónico para garantizar que se cumplan de manera efectiva en todo el sistema.

### 1.2.1. Utilidad de la arquitectura del software

Nos ayuda a razonar sobre el sistema (responder preguntas)

- ¿Cuáles son las partes del sistema que pueden cambiar?
- ¿Realizar este cambio "romperá" (afectará) a otra parte del sistema?
- ¿Dónde debo incluir cierta funcionalidad?

- ¿Qué significa esta decisión que hay en el código? Las decisiones arquitecturales terminan llegando al código

## 2. Arquitectura hexagonal

La arquitectura hexagonal fue propuesta por Alistair Cockburn en 2005, que la ha ido refinando hasta su versión actual denominada "Arquitectura de puertos y adaptadores" (Ports & Adapters). La última versión está documentada en [1].

El objetivo que pretende conseguir esta arquitectura es aislar el núcleo de la aplicación (la lógica de negocio) de las tecnologías externas (bases de datos, interfaces de usuario, servicios externos, etc.), para ello hay que diseñar la aplicación para que cumpla con las siguientes características:

- Pueda ejecutarse sin interfaz de usuario.
- Pueda ejecutarse sin base de datos.
- Pueda ser controlada por tests automáticos o scripts.
- Sea posible permitir cambiar tecnologías sin tocar el dominio.

La Figura 1 muestra el objetivo de la arquitectura hexagonal, que es aislar el núcleo de la aplicación de las tecnologías externas. La aplicación establece un límite claro dentro del cual (el hexágono) se encuentra la lógica de negocio, mientras que las tecnologías externas se encuentran fuera del hexágono. La aplicación se comunica con el exterior a través de interfaces bien definidas (puertos) y adaptadores que traducen las llamadas entre la aplicación y el exterior. De esta manera, se puede desarrollar y probar la lógica de negocio de manera independiente de las tecnologías externas, lo que facilita el desarrollo incremental e independiente, la creación de pruebas y la evolución del sistema a lo largo del tiempo.

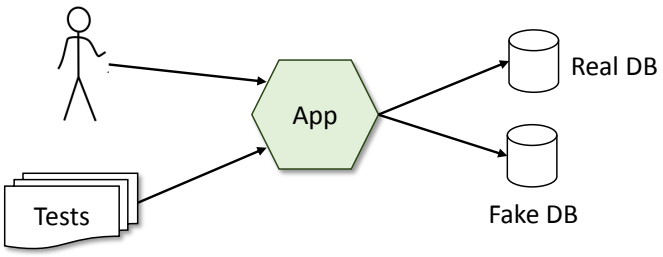


Figura 1: Objetivo de la arquitectura hexagonal: aislar el núcleo de la aplicación de las tecnologías externas.

### 2.1. Elementos

La arquitectura tiene cuatro elementos básicos:

- La aplicación o sistema en sí (lo que está dentro del hexágono).
- Los puertos que definen la interfaz de comunicación entre la aplicación y el exterior.
- Los actores externos que dirigen la aplicación o son dirigidos por la aplicación.
- Los adaptadores que se encargan de traducir las llamadas entre la aplicación y los actores externos a través de los puertos.

Además, desde un punto de vista práctico debe haber un elemento adicional que configure la conexión entre los cuatro elementos anteriores, pero eso

está fuera de la arquitectura propiamente dicha puesto que es un elemento de infraestructura.

En esta arquitectura un **aplicación** se ve como un componente reutilizable que no tiene dependencias tecnológicas externas. Esto quiere decir que esencialmente tiene lógica de negocio y lógica de coordinación (servicios de aplicación), pero no tiene lógica de infraestructura (bases de datos, interfaces de usuario, servicios externos, etc.)[1]..

[1]La aplicación puede ser fácilmente distribuida como una librería.

Esencialmente cualquier elemento físico con el que interacciona la aplicación está fuera de la aplicación. Esto incluye bases de datos, interfaces de usuario, servicios externos, redes, etc.

Para comunicarse con el exterior la aplicación define **puertos** (ports) que son interfaces que definen cómo la aplicación interactúa con el exterior. Los puertos son de dos tipos: puertos de entrada (*driving ports*) y puertos de salida (*driven ports*). Los puertos de entrada son los que utilizan los actores externos para interactuar con la aplicación, mientras que los puertos de salida son los que utiliza la aplicación para interactuar con los actores externos.

Esencialmente un puerto declara una o más interfaces que definen cómo la aplicación interactúa con el exterior. Hay dos tipos de interfaces: provided interfaces y required interfaces.

- La provided interface es la interfaz que la aplicación ofrece al exterior para que el pueda "dirigir" la aplicación.
- La required interface es la interfaz que la aplicación requiere del exterior para que la aplicación pueda "conducir" ciertos elementos.

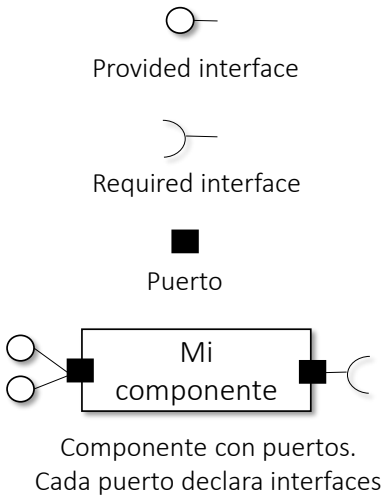

Figura 2: Símbolos UML para describir componentes.

### ⚠️ ¿Quién define las interfaces?

La aplicación define ambos tipos de interfaces puesto las interfaces representan el contrato que establece con el exterior. La diferencia es que las "provided interfaces" son el contrato que la aplicación se compromete a cumplir, mientras que las "required interfaces" son el contrato que la aplicación necesita o exige que el exterior cumpla.

Para interactuar con la aplicación se crean **adaptadores**. Un adaptador es un componente que implementa una interfaz de puerto (provided o required) y se encarga de transformar las llamadas entre la aplicación y el actor externo.

Algunos ejemplos de adaptadores para los puertos conducidos son:

- Un adaptador de base de datos que implementa la required interface del puerto de salida para permitir a la aplicación almacenar y recuperar datos de una base de datos ej., MySQL.
- Un adaptador de servicio web que implementa la required interface del puerto de salida para permitir a la aplicación comunicarse con una API externa (ej., una API de un LLM).

Estos adaptadores se implementan fuera de la aplicación en sí pero se despliegan junto con la aplicación.

Algunos ejemplos de adaptadores para los puertos dirigidos son:

- Una interfaz de usuario (UI) que invoca la provided interface del puerto de entrada para permitir a un usuario interactuar con la aplicación.
- Un conjunto de pruebas automatizadas que invoca la provided interface del puerto de entrada para probarlas

Es importante observar que la implementación concreta de la "provided interface" **está dentro** de la aplicación. El adaptador no la conoce porque se comunica a través de la interfaz. Su labor es traducir las llamadas del actor externo a la interfaz de la aplicación.

## 2.2. Ejemplo de arquitectura hexagonal y DDD

Supongamos que se quiere implementar una aplicación de pedidos para una tienda online. La funcionalidad principal es la de crear un pedido y cuando esté listo procesar el pago y guardar el pedido en una base de datos. Para ello necesita:

- Implementar el modelo de dominio que representa los pedidos y la lógica de negocio asociada.
- Definir un puerto de entrada para la operación de crear pedido, que será invocado por una interfaz de usuario o por pruebas automatizadas.
- Definir puertos de salida para interactuar con la base de datos para guardar los pedidos y para interactuar con un servicio de pago externo para procesar los pagos.
- Implementar adaptadores para la base de datos (ej., Postgresql) y para el servicio de pago (ej., Stripe).
- Implementar una interfaz de usuario (ej., una API REST) que invoque el puerto de entrada para crear pedidos.

### 2.2.1. Diseño e implementación de la aplicación con DDD

En la arquitectura hexagonal para el modelo de dominio se puede seguir el enfoque que se desee. Sin embargo, en nuestro caso utilizaremos Domain-Driven Design (DDD) de manera que el dominio se organizará según los patrones tácticos de DDD.

Definiremos entidades como `Pedido`, `LineaPedido`, `Cliente`, etc., junto con la lógica de negocio asociada. Además habrá que definir servicios de aplicación para representar las operaciones que pueden realizar los usuarios de la aplicación.

```
public class Pedido {

  private final PedidoId id;
  private final ClienteId clienteId;
  private final List<LineaPedido> items;
  private final LocalDateTime fechaCreacion;
  private EstadoPedido estado;

  // Resto del código
  public void addLineaPedido(ProductoId productoId, int qty) {
    // Lógica para añadir una línea de pedido
  }

  public void marcarComoPagado() {
    // Lógica para marcar el pedido como pagado
  }

}
```

Para poder realizar pagos necesitamos interactuar con un **servicio de pago externo**. Este es un servicio de infraestructura, para el cual definiremos una interfaz dentro de la aplicación pero que será implementada por un adaptador externo. Este servicio se usará en el servicio de aplicación correspondiente (ver debajo).

```
/** Definición de un servicio de infrastructura */
public interface ServicioDePago {

    EstadoPago procesarPago(PedidoId pedidoId, Dinero total);
}
```

El **servicio de aplicación** se encarga de coordinar la lógica de negocio para gestionar los pedidos. Tiene dos dependencias (ver constructor): el repositorio de pedidos para guardar y recuperar los pedidos de la base de datos, y el servicio de pago para procesar los pagos.

```
public class ServicioDePedidos {

  // Dependencias del servicio. Durante la configuración
  // de la aplicación se inyectarán las implementaciones
  // concretas de estas dependencias.
  private final RepositorioDePedidos repositorio;
  private final ServicioDePago servicioDePago;

  public ServicioDePedidos(RepositorioDePedidos repositorio,
                           ServicioDePago servicioDePago) {
      this.repositorio = repositorio;
      this.servicioDePago = servicioDePago;
  }

  // ...

  @Override
  public Pedido pagarPedido(String pedidoId)
    throws PagoRechazadoException {

    Pedido pedido = repositorio.findPorId(pedidoId);
    Preconditions.checkNotNull(pedido, "Pedido no encontrado");

    EstadoPago estadoPago = servicioDePago.procesarPago(pedidoId,
pedido.getTotal());

    if (estadoPago == EstadoPago.APROBADO) {
        pedido.marcarComoPagado();
        repositorio.guardar(pedido);
    } else {
        throw new PagoRechazadoException("El pago fue rechazado
para el pedido: " + pedidoId);
    }

    return pedido;
  }
}
```

### 2.2.2. Puertos y adaptadores

La aplicación definida en las clases anteriores es independiente de las tecnologías externas, pero para que pueda funcionar es necesario definir

los puertos y adaptadores correspondientes. Como se muestra en la Fig. Figura 3, la aplicación tiene:

Un puerto de entrada (***driving port***):

- `ServicioDePedidos` que define la interfaz para gestionar los pedidos.

Dos puertos de salida (***driven ports***):

- `RepositorioDePedidos` que define la interfaz para almacenar de manera persistente los pedidos.
- `ServicioDePago` que define la interfaz para interactuar con el servicio de pago externo

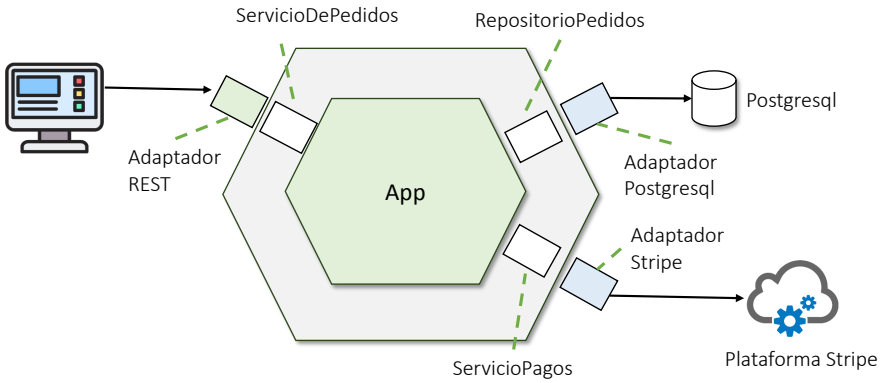


Figura 3: Ejemplo de arquitectura hexagonal cuyo nucleo es la aplicación creada con DDD para pedidos.

### 2.2.3. Correspondencia entre DDD y la arquitectura hexagonal

Los elementos de DDD tienen una correspondencia directa con los elementos de la arquitectura hexagonal:

- El modelo de dominio (entidades, objetos valor, servicios de dominio, etc.) se encuentra dentro del núcleo de la aplicación.
- Los servicios de aplicación (como `ServicioDePedidos`) también se encuentran dentro de la aplicación ("dentro del hexágono"), pero se intepretan como un puerto de entrada para que los actores externos puedan interactuar con la aplicación.
- Los repositorios corresponden a puertos de salida o ***driven ports*** que la aplicación utiliza para interactuar con la base de datos.
- Los servicios de infraestructura (como `ServicioDePago`) que necesite la aplicación también corresponden a puertos de salida o ***driven ports***.

## 2.3. Ventajas y desventajas

Los beneficios de esta arquitectura son los siguientes:

1. **Pruebas**. Se pueden escribir y ejecutar pruebas a nivel de sistema sin necesidad de configurar o conectar a las tecnologías de producción, lo cual facilita la creación de pruebas.
2. **No hay contaminación de detalles tecnológicos**. La propia arquitectura asegura que las dependencias externas no afecten a la lógica de negocio. Esto es así porque las pruebas van a demostrar que los detalles de la UI o de tecnología (ej., base de datos) no contaminan la lógica de negocio. ¿Por qué? Porque si así fuera no podrían ejecutarse las pruebas.

3. **Desarollo incremental e independiente**. Diferentes equipos pueden desarrollar sus partes del sistema de manera independiente, probarlas por separado y conectarlas mediante interfaces bien definidas y probadas. Además, el sistema se puede empezar a construir sin necesidad de tener todas las conexiones externas listas.
4. **Mantenibilidad**: Se pueden reemplazar las conexiones externas a medida que cambian las necesidades tecnológicas y de negocio a lo largo de los años.
5. **Diseño centrado en el dominio**: Al mantener los elementos tecnológicos fuera del núcleo de la aplicación, se puede centrar en el diseño del dominio, por ejemplo utilizando Domain-Driven Design (DDD).
6. **No hay necesidad de recompilar**. El núcleo de la aplicación no depende de las tecnologías externas, por lo que no es necesario recompilar el sistema para cambiar entre entornos de prueba y producción o para actualizar las tecnologías externas. En otras palabras el sistema es "externamente extensible" como un sistema de plugins.

Sin embargo, existen algunos costes asociados a esta arquitectura:

1. Es necesario diseñar y definir las interfaces entre el núcleo de la aplicación y las tecnologías externas, lo cual requiere tiempo y esfuerzo adicional.
2. Hay una sobrecarga inicial en la configuración del proyecto y la estructura del código para implementar la arquitectura hexagonal.
3. Puede haber una ligera disminución del rendimiento debido a la capa adicional de abstracción entre el núcleo de la aplicación y las tecnologías externas.

## 3. Arquitectura REST

REST (REpresentational State Transfer) es un estilo arquitectónico para sistemas distribuidos, especialmente para servicios web. Fue definido formalmente por Roy Fielding en su tesis doctoral en el año 2000 [2].

REST es independiente de la tecnología, pero en la práctica se suele implementar utilizando el protocolo HTTP y formatos de datos como JSON o XML para la representación de los recursos.

En esencia, un servicio REST se organiza como un sitio web en el sentido de que los recursos se identifican mediante URIs. Se accede a los recursos a través de peticiones que un cliente realiza a un servidor (para realizar las peticiones se utilizan los métodos HTTP estándar, GET, POST, PUT, DELETE, etc.). Además, los recursos pueden estar relacionados entre sí mediante enlaces (hypermedia).

Una URI (Uniform Resource Identifier) es una cadena de caracteres que identifica de manera única un recurso en la web. Por ejemplo, `https://tienda.com/productos/123` es una URI que identifica el recurso del producto con ID 123 en la tienda online. Una URL (Uniform Resource Locator) es un tipo específico de URI que no solo identifica un recurso, sino que también proporciona la información necesaria para localizarlo en la red (normalmente identifica el protocolo, el host y la ruta).

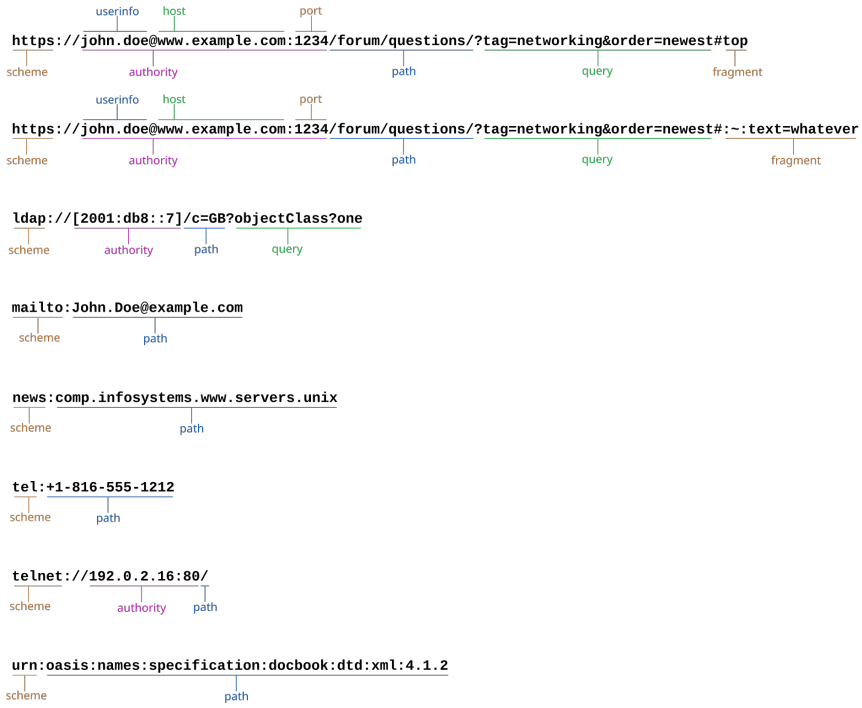


Figura 4: Ejemplos de URIs. https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

Por ejemplo, en una tienda online, los productos pueden ser recursos accesibles mediante URLs como `https://tienda.com/productos/123`, y las categorías de productos pueden estar relacionadas con los productos mediante enlaces. Por ejemplo,

Ante una petición del cliente como:

```
GET https://tienda.com/productos/123
```

El servidor devolvería la representación del recurso en cierto formato, por ejemplo JSON, como se muestra a continuación:

```
{
  "id": 123,
  "nombre": "Camiseta",
  "precio": 19.99,
  "categoria": {
    "id": 5,
    "nombre": "Ropa",
    "enlace": "https://tienda.com/categorias/5"
  }
}
```

Entonces, el cliente puede realizar las acciones necesarias, incluyendo seguir el enlace a la categoría para obtener más información sobre ella.

Antes de presentar en más detalle la arquitectura REST, es importante entender el protocolo HTTP, que se presenta a continuación.

## 3.1. El protocolo HTTP

El protocolo HTTP (Hypertext Transfer Protocol) fue creado en 1989 en el CERN por Tim Berners-Lee, junto con el lenguaje HTML, con el objetivo de compartir documentos científicos entre investigadores. Es la base de la World Wide Web (WWW), el sistema que hizo posible la navegación mediante enlaces (hipertexto).

HTTP permite que los clientes (como los navegadores web) se comuniquen con los servidores (ej., que alojan los sitios web) para solicitar y recibir recursos, como páginas web, imágenes, videos, etc. Para ello, HTTP define el tipo de peticiones soportadas (GET, POST, PUT, DELETE, etc.), el formato de las peticiones y respuestas, y los códigos de estado que indican el resultado de las peticiones.

La Figura 5 muestra un ejemplo de la arquitectura cliente-servidor de la Web, donde un navegador (actuando como cliente) realiza una petición HTTP a un servidor para obtener un recurso (una páginas web).

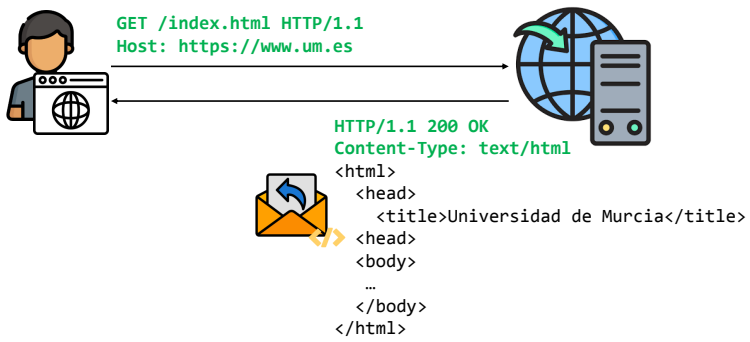


Figura 5: Ejemplo de la arquitectura cliente-servidor de la Web.

Se recomienda consultar la referencia de MDN para HTTP (https://developer.mozilla.org/es/docs/Web/HTTP). En particular, los métodos HTTP (https://developer.mozilla.org/es/docs/Web/HTTP/Methods).

### 3.1.1. Peticiones y respuestas

La comunicación entre cliente y servidor se realiza mediante peticiones HTTP. Las peticiones HTTP incluyen un método (GET, POST, PUT, DELETE, etc.), una URL que identifica el recurso, encabezados que proporcionan metadatos adicionales y, opcionalmente, un cuerpo que contiene datos.

Una **petición** (Request) es una solicitud enviada por el cliente al servidor para realizar una acción específica sobre un recurso.

Una **respuesta** (Response) es la información que el servidor envía de vuelta al cliente en respuesta a una petición. La respuesta incluye un código de estado HTTP, encabezados y un cuerpo con los datos solicitados o el resultado de la acción.

### 3.1.2. Métodos HTTP

HTTP está basado en operaciones estandarizadas que indican la acción sobre un recurso. Estas operaciones se denominan **métodos HTTP**. Así, cada

**Semántica de las operaciones**

Se dice que una operación tiene **efectos laterales** (*side effects*) cuando la ejecución de la operación modifica el estado del servidor, esto es, si añade, modifica o elimina un recurso.

Un método HTTP es **idempotente** cuando se puede ejecutar la misma petición repetidamente y el estado resultante es el mismo.

Algunos ejemplos de idempotencia son:

- `f(x) = x` porque `f(f(x)) = f(x) = x`
- Multiplicar por 1 por que `1 * 1 * 1 = 1 * 1 = 1`
- Botón de un ascensor: pulsar varias veces el botón no cambia el estado del ascensor (no hará que llegue antes).

petición indica el método que se desea ejecutar sobre el recurso identificado por la URL. Cada método tiene una semántica específica que define su comportamiento y efectos sobre los recursos.

Los métodos están basados en las operaciones CRUD (Create, Read, Update, Delete), que son las operaciones básicas para gestionar recursos en un sistema, aunque también hay algunos métodos de diagnóstico.

A continuación se muestra una tabla con los métodos HTTP más comunes.

| Método | Descripción | Efecto lateral | Idempotente |
|---|---|---|---|
| **GET** | Obtener información | No | Sí |
| **POST** | Crear un recurso | Sí | No |
| **PUT** | Reemplazar recurso completo | Sí | Sí |
| **PATCH** | Modificar parcialmente | Sí | No |
| **DELETE** | Eliminar recurso | Sí | Sí |

### Método GET

Se utiliza para la lectura de recursos. Un método GET nunca cambia el estado del recurso, es decir, no tiene efectos laterales. Puesto que es idempotente, la respuesta a una petición GET puede ser cacheada por el cliente o por intermediarios (proxies) para mejorar el rendimiento y reducir la carga en el servidor.

Ejemplos de peticiones GET (formato informal):

```
GET /empleados      # Listar todos los empleados
GET /empleados/1    # Obtener los detalles del empleado 1
```

### Método POST

Se utiliza para crear recursos. Cada invocación de POST crea un nuevo recurso y por tanto no es idempotente.

```
POST /empleados # Crea un nuevo empleado.

# La respuesta incluirá una cabecera Location con la URL del nuevo
recurso creado, por ejemplo:
HTTP/1.1 201 Created
Location: /empleados/12
```

### Método PUT

Se utiliza para crear o actualizar (reemplazar) un recurso. Por tanto, en la petición hay que incluir la descripción completa del recurso en la petición. A diferencia de PATCH, PUT no está pensado para actualizaciones parciales. Al contrario que POST, el método PUT sí que es idempotente.

```
PUT /empleados/1 # actualiza el empleado 1
PUT /empleados/1 # puede significar crear el empleado 1 si no
existe (menos común)
```

La diferencia entre el método PUT y el método POST es que PUT sí es un método idempotente: llamarlo una o más veces de forma sucesiva tiene el mismo efecto (sin efectos secundarios), mientras que una sucesión de peticiones POST idénticas pueden tener efectos adicionales, como enviar una orden varias veces.

Entonces, ¿Cómo puede ser que PUT sea idempotente cuando se usa para crear recursos? ¿Qué impacto tiene esto en la práctica?

La razón es que PUT debe incluir el identificador del recurso en la URL. Eso significa que si PUT se utiliza para crear recursos se está asumiendo que el cliente conoce la manera de generar el identificador del recurso, lo cual no es común, ya que normalmente el servidor es el encargado de generar los identificadores.

### 3.1.3. Formato de los mensajes

En HTTP, tanto las peticiones como las respuestas siguen un formato específico. En https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Messages se puede encontrar una descripción detallada del formato de los mensajes HTTP.

Tanto las peticiones como las respuestas comparten una estructura similar (ver Fig. Figura 6) que consta de los siguientes componentes:

- **start-line**: Es la primera línea del mensaje y contiene información sobre el método HTTP (en el caso de las peticiones) o el código de estado (en el caso de las respuestas).
- **headers**: Son líneas adicionales que proporcionan metadatos sobre el mensaje, como el tipo de contenido, la longitud del cuerpo, las cookies, etc.
- **empty line**: Es una línea vacía que indica el final de los encabezados y el comienzo del cuerpo del mensaje.
- **body**: Es la parte del mensaje que contiene los datos asociados a la petición o respuesta. Por ejemplo, en una petición POST, el cuerpo contiene los datos que se quieren enviar al servidor, mientras que en una respuesta, el cuerpo contiene la representación del recurso solicitado. En el caso de una petición GET, el cuerpo está vacío, ya que no se envían datos al servidor.

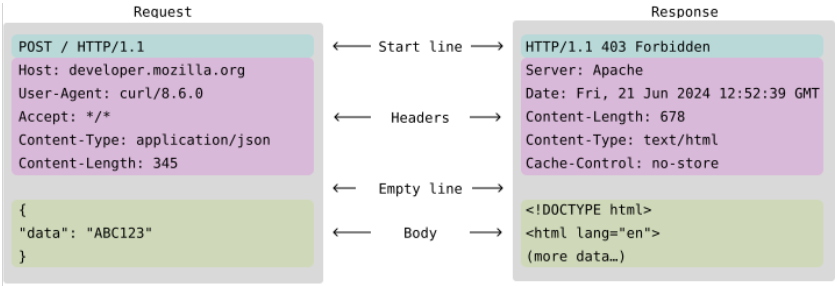


Figura 6: Formato de los mensajes HTTP. Extraído de https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Messages

HTTP ofrece **negociación de contenido** para que el cliente y el servidor puedan acordar el formato de los datos que se intercambian. Para ello:

- El cliente indica el formato de los datos que está enviando al servidor mediante el encabezado `Content-Type` en las peticiones que incluyen un cuerpo (como POST o PUT). Por ejemplo, si el cliente está enviando datos en formato JSON, incluirá el encabezado `Content-Type: application/json`.
- Además, el cliente puede incluir en los encabezados de la petición un campo `Accept` que indique los formatos de datos que puede procesar (por ejemplo, `application/json`, `text/html`, etc.).
- El servidor puede responder con un encabezado `Content-Type` que indique el formato de los datos que está enviando en el cuerpo de la respuesta.

### 3.1.4. Códigos de estado HTTP

HTTP estandariza los códigos de respuesta que el servidor envía al cliente para indicar el resultado de la petición. Estos códigos de estado son números de tres dígitos que se agrupan en diferentes categorías según su significado.

- **2xx (éxito)**: Significan que la petición ha sido procesada con éxito.
  - **200 OK**: La petición se ha procesado correctamente y se devuelve la información solicitada.
  - **201 Created**: La petición se ha procesado correctamente y se ha creado un nuevo recurso.
  - **204 No Content**: La petición se ha procesado correctamente pero no hay contenido que devolver.
- **3xx (redirección)**: Indican que el recurso solicitado está en otro lugar y el cliente debe realizar una nueva petición a la URL proporcionada en la respuesta u obtener el recurso por otro medio (ej., cuando está en caché).
  - **301 Moved Permanently**: El recurso solicitado ha sido movido permanentemente a una nueva URL.
  - **304 Not Modified**: El recurso solicitado no ha sido modificado desde la última vez que se solicitó, por lo que el cliente puede usar la versión en caché.
- **4xx (error del cliente)**: Indican que hubo un error en la petición realizada por el cliente. Esto puede deberse a una solicitud mal formada, falta de autorización, recurso no encontrado, etc.
  - **400 Bad Request**: La petición no se pudo entender o procesar debido a una petición mal formada.
  - **401 Unauthorized**: La petición requiere autenticación y el cliente no ha proporcionado credenciales válidas.
  - **403 Forbidden**: El servidor entiende la petición pero se niega a autorizarla.
  - **404 Not Found**: El recurso solicitado no se encontró en el servidor.
- **5xx (error del servidor)**: Indican que hubo un error en el servidor al procesar la petición. Esto puede deberse a un error interno, una sobrecarga, un tiempo de espera agotado, etc.
  - **500 Internal Server Error**: El servidor encontró una condición inesperada que le impidió cumplir con la petición.
  - **503 Service Unavailable**: El servidor no está disponible temporalmente, generalmente debido a mantenimiento o sobrecarga.

Entender los códigos de estado HTTP más comunes es fundamental para desarrollar aplicaciones web y servicios RESTful, ya que permiten a los desarrolladores manejar las respuestas del servidor de manera adecuada y proporcionar una mejor experiencia al usuario.

### 3.2. La arquitectura REST

La arquitectura REST se basa en los mismos principios de la WWW, pero en los servidores en lugar de servir páginas web (ficheros HTML), sirven recursos representados en algún formato estructurado como JSON o XML.

De acuerdo a [2], REST se basa en seis restricciones arquitectónicas fundamentales:

1. Arquitectura cliente-servidor
2. Sin estado (Stateless)
3. Cache
4. Interfaz uniforme
5. Organización en capas

6. Código bajo demanda (opcional)

Estas restricciones definen las reglas y principios que deben seguirse para diseñar sistemas RESTful. A continuación se describen cada una de estas restricciones.

### 3.2.1. Arquitectura cliente-servidor

El sistema debe tener dos tipos de componentes principales: clientes y servidores, es decir, se debe seguir el patrón arquitectónico cliente-servidor.

La principal razón es la **separación de responsabilidades** de manera que el cliente es responsable de la interfaz de usuario y la experiencia del usuario, mientras que el servidor maneja la lógica de negocio y el almacenamiento de datos. Esta separación permite una mayor escalabilidad y flexibilidad en el desarrollo y despliegue de aplicaciones.

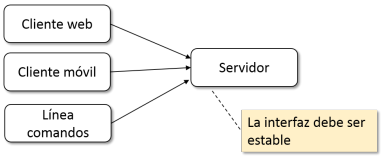

Figura 7: Clientes y servidores en la arquitectura REST

En la Fig. Figura 7 se muestra como tres clientes diferentes pueden acceder a un mismo servicio REST. Para que esto pueda funcionar es importante que la interfaz entre los clientes y el servidor esté bien definida y se mantenga estable (es decir, si se cambiar la interfaz los clientes pueden verse afectados)

### 3.2.2. Sin estado (Stateless)

Cada petición del cliente al servidor debe contener toda la información necesaria para que el servidor pueda entender y procesar la petición. Es decir, el estado de la sesión debe almacenarse totalmente en el cliente.

El servidor no debe almacenar ningún estado del cliente entre peticiones. Por "estado del cliente" ser refiere a información sobre las peticiones e interacciones previas del cliente. Por ejemplo, en un proceso de compra no se almacenará la etapa del proceso o información de autenticación del cliente en el servidor, sino que toda esa información se incluirá en cada petición que el cliente realice al servidor. Esto no quiere decir que el servidor no almacene estado: el estado que almacena el servidor es el estado de los recursos, pero no el estado de las interacciones con los clientes.

**Tesis de Roy Fielding, Seccion 5.1.3**: Like most architectural choices, the stateless constraint reflects a design trade-off. The disadvantage is that it may decrease network performance by increasing the repetitive data (per-interaction overhead) sent in a series of requests, since that data cannot be left on the server in a shared context. In addition, placing the application state on the client-side reduces the server's control over consistent application behavior, since the application becomes dependent on the correct implementation of semantics across multiple client versions.

Hay tres razones importantes para esta restricción:

1. **Visibilidad**. A la hora de monitorizar y depurar el sistema, es más sencillo porque no es necesario considerar el estado del servidor, es suficiente con analizar las peticiones y respuestas.
2. **Fiabilidad**. Si el servidor no mantiene estado, es más sencillo recuperarse de fallos, ya que cualquier instancia del servidor puede manejar cualquier petición sin necesidad de tener en cuenta el estado previo.
3. **Escalabilidad**. Al no tener que mantener el estado de los clientes, el servidor puede manejar un mayor número de peticiones simultáneas y distribuir la carga entre múltiples servidores de manera más eficiente.

Una pregunta muy relevante es cómo es posible conseguir que el servidor no tenga estado cuando muchas operaciones requieren mantener un estado, como por ejemplo el ***login*** o el carrito de la compra en una tienda online[2].

[2] https://chatgpt.com/share/6943dbab-dca8-8010-8e3a-07397edbbd40

### 3.2.3. Cache

El servidor puede marcar las respuestas como cacheables o no cacheables. Si una respuesta es cacheable, el cliente puede almacenar la respuesta y reutilizarla para peticiones futuras, evitando así tener que contactar con el servidor nuevamente para obtener el mismo recurso.

La principal cuestión es cómo determinar en qué momento expira la cache para evitar que el cliente utilice datos obsoletos.

Por ejemplo, en Spring Boot podríamos utilizar dos mecanismos para controlar la cache de las respuestas: `Cache-Control` y `ETag`. El siguiente ejemplo muestra cómo utilizar ambos mecanismos en una respuesta a una petición GET:

```
@GetMapping("/book/{id}")
public ResponseEntity<Book> showBook(@PathVariable Long id) {

  Book book = findBook(id);
  String version = book.getVersion();

  return ResponseEntity
      .ok()
      .cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))
      .eTag(version)
      .body(book);
}
```

En este ejemplo:

- `Cache-Control: max-age=2592000` indica que la respuesta es cacheable durante 30 días. El navegador no pediría de nuevo

el libro durante ese tiempo, a menos que el usuario borre la cache o se utilice un mecanismo de actualización forzada (como refrescar la página CTRL+SHIFT+R). Este mecanismo es útil para recursos que no cambian con frecuencia, como imágenes o archivos estáticos.

- `ETag: "version"` proporciona una etiqueta única para la versión del recurso. El cliente puede incluir esta etiqueta en una petición futura utilizando el encabezado `If-None-Match`. Para esto, el servidor debe tener una manera de generar

versiones de los recursos, por ejemplo utilizando un campo de versión en la base de datos o un hash del contenido del recurso. El funcionamiento (asumiendo solo el ETag) es como sigue:

- El Cliente pide el libro.
- Recibe el JSON, con un ETag: “v1”.
- El Cliente guarda el ETag en caché.
- En la base de datos el libro cambia, y pasa a tener la versión “v2”.
- El Cliente vuelve a pedir el libro:
- Su navegador envía el header If-None-Match: “v1”.
- El servidor compara “v1” con la versión actual del libro “v2”. La condición falla y el servidor envía el nuevo JSON con el nuevo ETag.

### 3.2.4. Interfaz uniforme

El principo de **interfaz uniforme** establece que la interfaz entre los clientes y el servidor debe seguir un conjunto de reglas y convenciones que permitan a los clientes interactuar con el servidor de manera consistente y predecible.

Por tanto, cada recurso de una API REST debe seguir una estructura predecible y estandarizada. Para ello, lo habitual en un API REST es hacer uso de HTTP para conseguir esta uniformidad, siguiendo las siguientes reglas:

1. Identificación de recursos: Cada recurso debe ser identificado mediante una URI.
2. Uso de métodos HTTP estandar: La interacción con los recursos se hace utilizando la semántica proporcionada por los métodos HTTP (GET, POST, PUT, DELETE, etc.)[3].
3. Mensajes auto-descriptivos: Cada mensaje (petición o respuesta) debe contener toda la información necesaria para que el receptor pueda entenderlo y procesarlo correctamente, sin necesidad de información adicional.
4. HATEOAS (Hypermedia as the Engine of Application State): El servidor debe incluir hipervínculos en las respuestas que permitan a los clientes descubrir y navegar por los recursos disponibles de manera dinámica, sin necesidad de conocer la estructura completa de la API de antemano.

[3] Observese que esta aproximación es diferente a otras aproximaciones donde el desarrollador define un conjunto de métodos específicos lo que hace la interfaz menos predecible.

La principal desventaja de esta aproximación es que puede resultar menos eficiente, ya que la uniformidad de la interfaz hace que la interacción no siempre utilice el mecanismo más óptimo para caso específico.

### 3.2.5. Organización en capas

El sistema debe estar organizado en capas jerárquicas, de manera que un cliente no puede conocer qué hay detrás del servidor con el que se está comunicando.

Con esta organización el cliente no puede saber si está comunicandose directamente con el servidor final o con un intermediario (proxy, gateway, etc.). Cada capa tiene una función específica y puede interactuar solo con las capas adyacentes.

Esta decisión arquitectural tiene varias ventajas:

1. Extensilibilidad: Se pueden añadir nuevas capas o modificar las existentes sin afectar a otras partes del sistema **siempre que se mantenga la misma interfaz**.
2. Seguridad: Las capas intermedias pueden actuar como filtros o firewalls, protegi
3. Escalabilidad: Un servidor puede distribuir la carga entre múltiples servidores intermedios para mejorar el rendimiento y la disponibilidad.

### 3.2.6. Código bajo demanda (opcional)

El servidor puede enviar código ejecutable (como scripts JavaScript) al cliente para que este lo ejecute. Esto permite extender la funcionalidad

del cliente de manera dinámica sin necesidad de actualizar el software del cliente.

## Bibliografía

[1] A. Cockburn y J. M. G. de Paz, «Hexagonal architecture explained», ***How the ports & adapters architecture simplifies your life, and how to implement it. Humans and Technology Inc***, 2024.

[2] R. T. Fielding, ***Architectural styles and the design of network-based software architectures***. University of California, Irvine, 2000.