



# UT01 Arquitecturas Web

👤 [darioaxel](#) ⌚ Alrededor de 21 min

---

**En este tema trabajaremos los siguientes RAs:**

- RA1. Selecciona las arquitecturas y tecnologías de programación Web en entorno servidor, analizando sus capacidades y características propias.

## UT01 Arquitecturas Web

### ¿Qué vamos a aprender en esta unidad?

En esta unidad vamos a aprender los conceptos básicos de las arquitecturas web. Para ello empezaremos analizando los principios SOLID, los patrones de diseño y las arquitecturas de software más comunes.

También veremos cómo se relacionan con el desarrollo de aplicaciones web del lado del servidor (backend) y cómo se utilizan en la práctica.

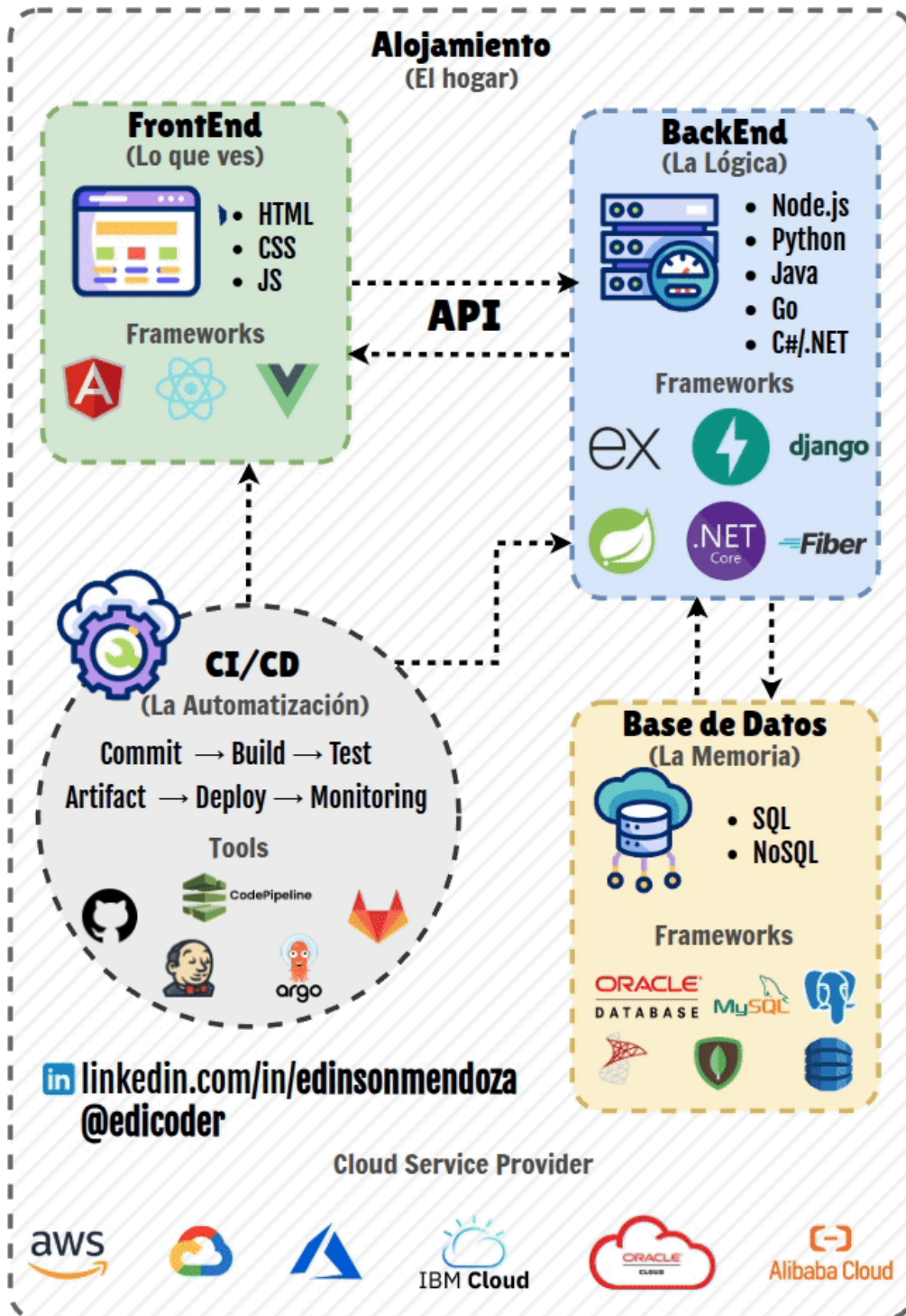
## Introducción

---

### El Desarrollo Web Actual Web

¿Algunas vez has pensado lo que pasa una vez le das al botón de "Enviar" en un formulario web? ¿O cómo es posible que puedas ver tu perfil en una red social, comprar en una tienda online o ver una película en streaming desde cualquier dispositivo? ¿O que pasa cuando pones una URL en el navegador? Detrás de estas acciones aparentemente simples, hay un complejo ecosistema de tecnologías y procesos que hacen posible la experiencia web moderna.

# Anatomía de una App Web moderna



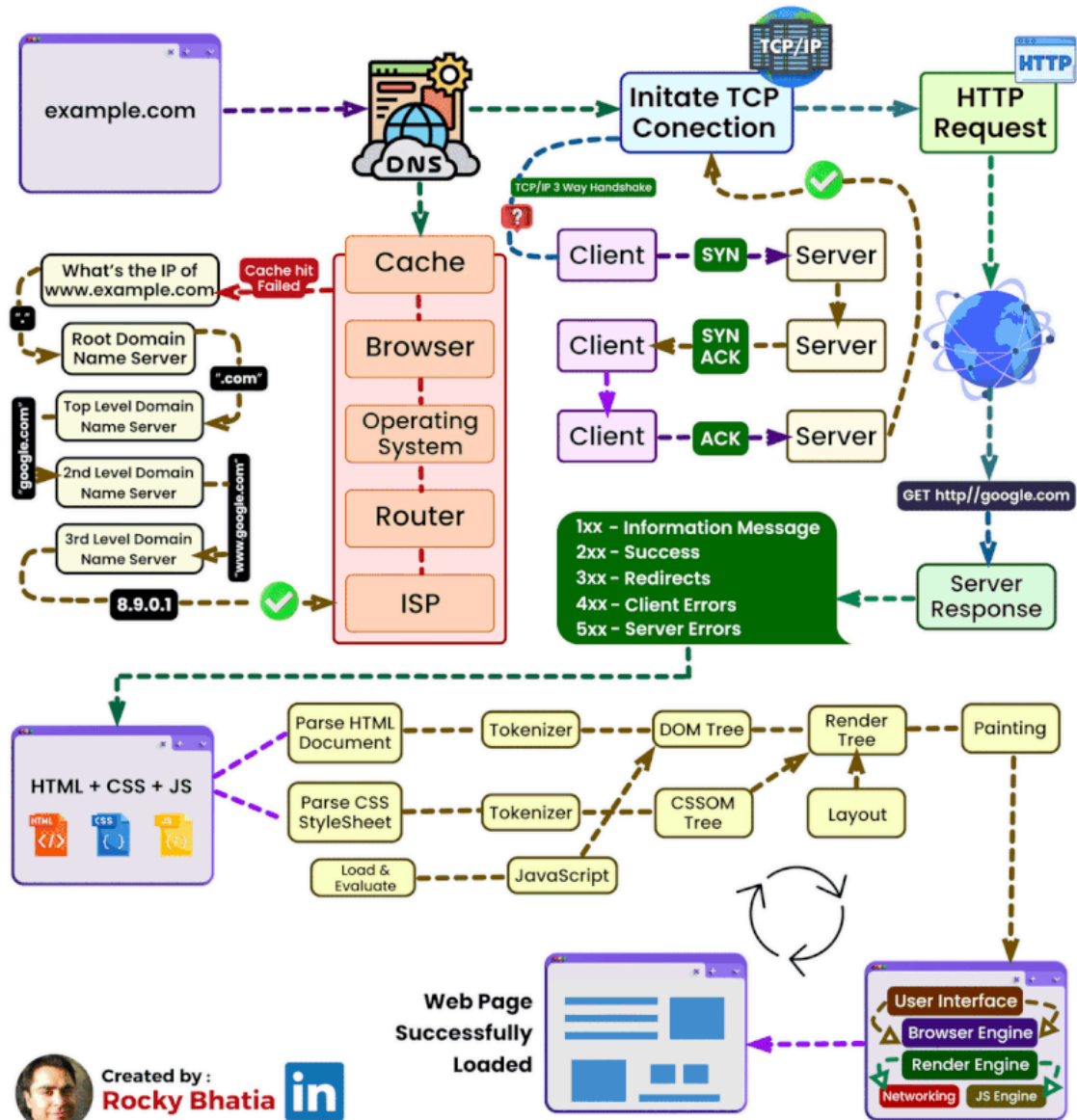
img

El desarrollo web moderno es un campo en constante evolución que abarca la creación y mantenimiento de sitios web y aplicaciones que operan a través de Internet. En la actualidad, el desarrollo no solo se enfoca en la funcionalidad, sino también en cómo estas apli-

caciones se pondrán a disposición de los usuarios, un proceso crucial conocido como **despliegue**. Este proceso es fundamental para la viabilidad de cualquier proyecto web, ya que permite que la aplicación pase del entorno de desarrollo a un entorno de producción, donde será accesible para los usuarios finales.

Los principales objetivos del despliegue son garantizar la **accesibilidad**, la **estabilidad**, la **escalabilidad** y la **seguridad** de las aplicaciones. Un despliegue eficiente contribuye a una **rapidez en el *Time-to-Market***, lo que permite a las empresas lanzar productos más rápidamente. Facilita la **iteración rápida** y la entrega continua de nuevas funcionalidades y mejoras, esencial para adaptarse a las necesidades del mercado y de los usuarios. La **automatización** de los procesos de despliegue reduce los errores humanos y aumenta la eficiencia, liberando a los equipos de desarrollo para tareas más estratégicas. Un despliegue eficaz mejora la **competitividad** de una empresa al proporcionar un servicio fiable y de alta calidad. Además, la **documentación** de todos los procesos de despliegue es indispensable para asegurar que puedan ser replicados, para la resolución de problemas y para la formación de nuevos miembros del equipo.

# What Happens When You Type In A URL in a Browser



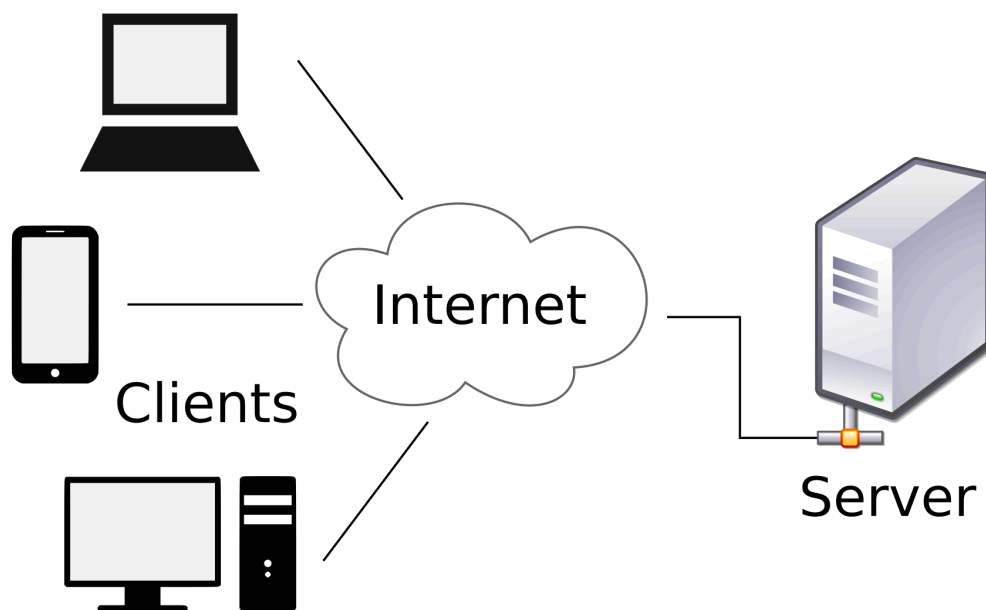
img

En el mundo de la informática establecemos que una **página web estática** es aquella cuyo contenido no cambia en función de la interacción del usuario o del contexto. Estas páginas están compuestas principalmente por archivos HTML y CSS, que se almacenan en un servidor y se envían tal cual al navegador del usuario cuando éste las solicita. El servidor simplemente "sirve" estos archivos, sin realizar ningún procesamiento adicional.

Este modelo es el más sencillo que se puede establecer para una página web. Sin embargo, las aplicaciones web modernas requieren una mayor interactividad y personalización, lo que nos lleva a la necesidad de páginas web dinámicas.

Estas páginas pueden cambiar su contenido en función de la interacción del usuario, datos de bases de datos o servicios externos.

Las aplicaciones web modernas se basan en la arquitectura cliente-servidor. En este modelo, uno o varios clientes (normalmente navegadores web) realizan peticiones al servidor, que responde enviando los recursos solicitados, tal y como podemos ver en el siguiente esquema.



Arquitectura Cliente-Servidor

El cliente (navegador) realiza peticiones al servidor, habitualmente usando el protocolo HTTP (puertos 80 o 443 para HTTPS). El servidor procesa la solicitud y responde con el contenido adecuado.

Existen dos formas principales de generar páginas dinámicas:

- **Procesamiento en el servidor:** El servidor genera el contenido dinámico utilizando lenguajes como PHP, Python, Ruby, Java, .NET, etc., y puede acceder a bases de datos o servicios externos para construir la respuesta.
- **Consumo de servicios externos desde el cliente:** El navegador ejecuta JavaScript para solicitar datos a servicios REST de terceros y actualizar la página dinámicamente, sin necesidad de recargarla por completo.

# Arquitectura Cliente-Servidor

---

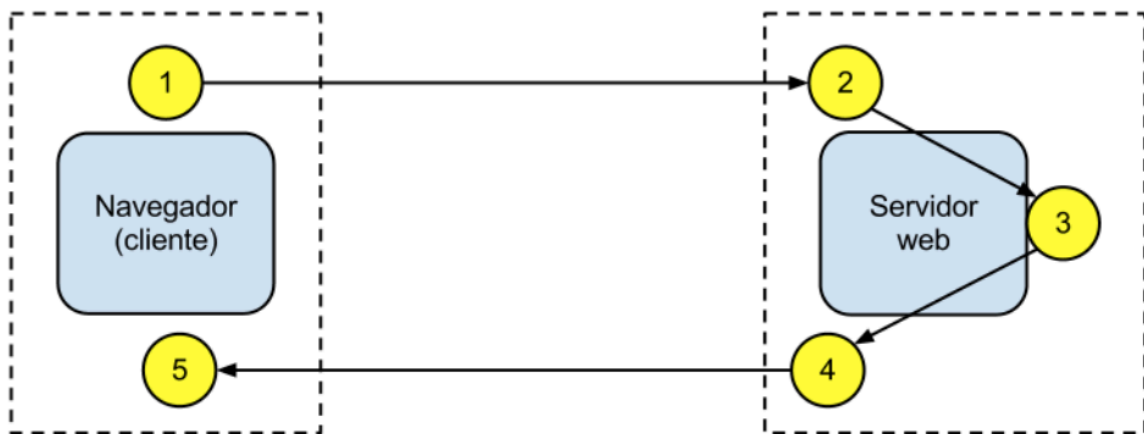
Las arquitecturas web son modelos que describen la forma en que los distintos elementos que participan en el intercambio y procesamiento de información a través de Internet se relacionan y funcionan. El modelo fundamental es la **Arquitectura Cliente-Servidor**, donde uno o varios clientes (navegadores web) solicitan servicios a un servidor.

El modelo cliente-servidor es un modelo que reparte tareas entre los proveedores de un recurso o servicio, llamados **servidores**, y los solicitantes/consumidores del servicio, llamados **clientes**.

Lo más frecuente es que los clientes y los servidores se comuniquen a través de una red de comunicaciones, pero ambos pueden residir en la misma máquina (normalmente en tareas de desarrollo).

El esquema de funcionamiento más básico del modelo cliente-servidor para una arquitectura web está basado en uno o varios clientes que solicitan una página web a un servidor web:

1. Desde el navegador web (o agente de usuario, que puede ser también una app nativa u otro servidor incluso) el usuario solicita un servicio web indicando su URL.
2. El servidor recibe la **petición** mediante el protocolo de aplicación HTTP, y la procesa mediante su **lógica de negocio**.
3. Produce una **respuesta** HTTP a la petición, que envía al cliente. Esta respuesta puede contener **ficheros** de distinta naturaleza: HTML, CSS, XML, JSON, ficheros multimedia, código JavaScript, etc.
4. El navegador web recibe la información enviada por el servidor y la interpreta. En función de la respuesta enviada, se respresenta en el navegador la respuesta al usuario (normalmente en forma de página web).



A continuación se muestran las ventajas y desventajas al respecto:

#### ***Ventajas:***

- **Centralización** del control: los accesos, recursos y la integridad de los datos son controlados por el servidor. Esta centralización también facilita la tarea de actualizar datos u otros recursos.
- **Escalabilidad**: se puede aumentar la capacidad de clientes y servidores por separado. Cualquier elemento puede ser aumentado (o mejorado) en cualquier momento, o se pueden añadir nuevos nodos a la red (clientes y/o servidores), siempre que el sistema esté diseñado para ello.
- **Portabilidad**: el hecho de que la aplicación web se ejecute en un navegador web, hace que se independice el software del sistema operativo sobre el que se ejecuta. De esta forma, se aprovecha el desarrollo para las diferentes plataformas.
- Fácil **mantenimiento**: al estar distribuidas las funciones y responsabilidades entre varios ordenadores independientes, es posible reemplazar, reparar, actualizar, o incluso trasladar un servidor, mientras que sus clientes no se verán afectados por ese cambio (o se afectarán mínimamente). Esta independencia de los cambios también se conoce como **encapsulación**.
- Existen **tecnologías**, suficientemente desarrolladas, diseñadas para el modelo de cliente-servidor que aseguran la seguridad en las transacciones, la usabilidad de la interfaz, y la facilidad de uso.

#### ***Desventajas:***

- La **congestión** del tráfico ha sido siempre un problema en esta arquitectura. Cuando una gran cantidad de clientes envían peticiones simultáneas al mismo servidor, se pueden producir situaciones de sobrecarga.
- Cuando un servidor está caído, las peticiones de los clientes **no pueden ser satisfechas**, ya que los recursos no están distribuidos.



- El software y el hardware de un servidor son generalmente muy determinantes. Normalmente se necesita **software y hardware específico**, dependiendo del tipo de servicio web, sobre todo en el lado del servidor. Esto aumentará el coste. Como alternativa, se dispone de servicios web en la nube, con diversos tipos de costes dependientes de la arquitectura web.

### Nota

Estas desventajas se refieren al caso en que los recursos del servidor no están replicados y/o distribuidos. Actualmente existen técnicas de escalado horizontal y vertical que pueden subsanar estos problemas.

## Ejemplo práctico

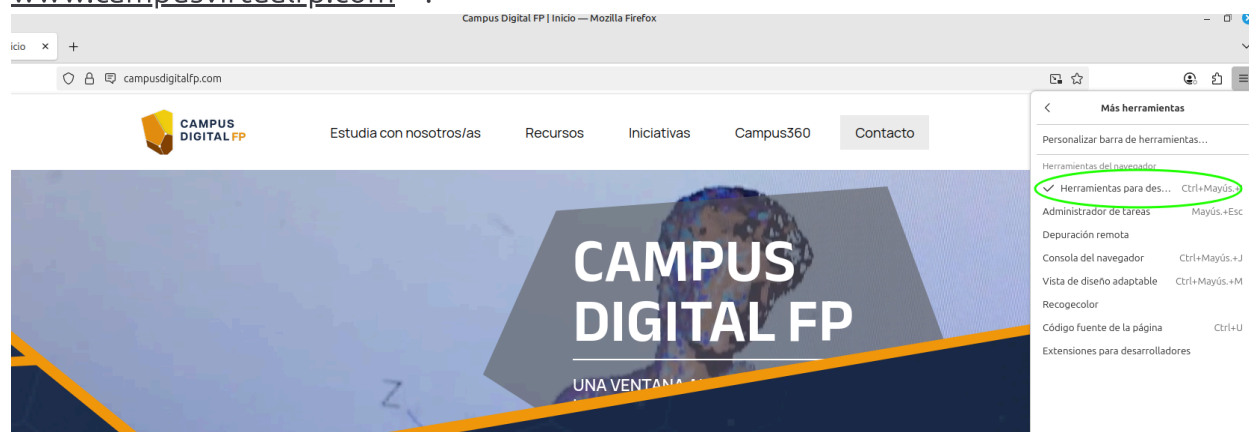
En este apartado vamos a tratar de indagar un poco más en qué sucede detrás de las cortinas cuando consultamos una URL.

Vamos a observar, a través de las herramientas de desarrollador del navegador web de Chrome (igual nos puede servir Firefox o cualquier otro), los 4 pasos que se detallaban en el apartado anterior.

Para ello vamos a utilizar la página web del Campus Virtual FP ([www.campusvirtualfp.com](http://www.campusvirtualfp.com)).

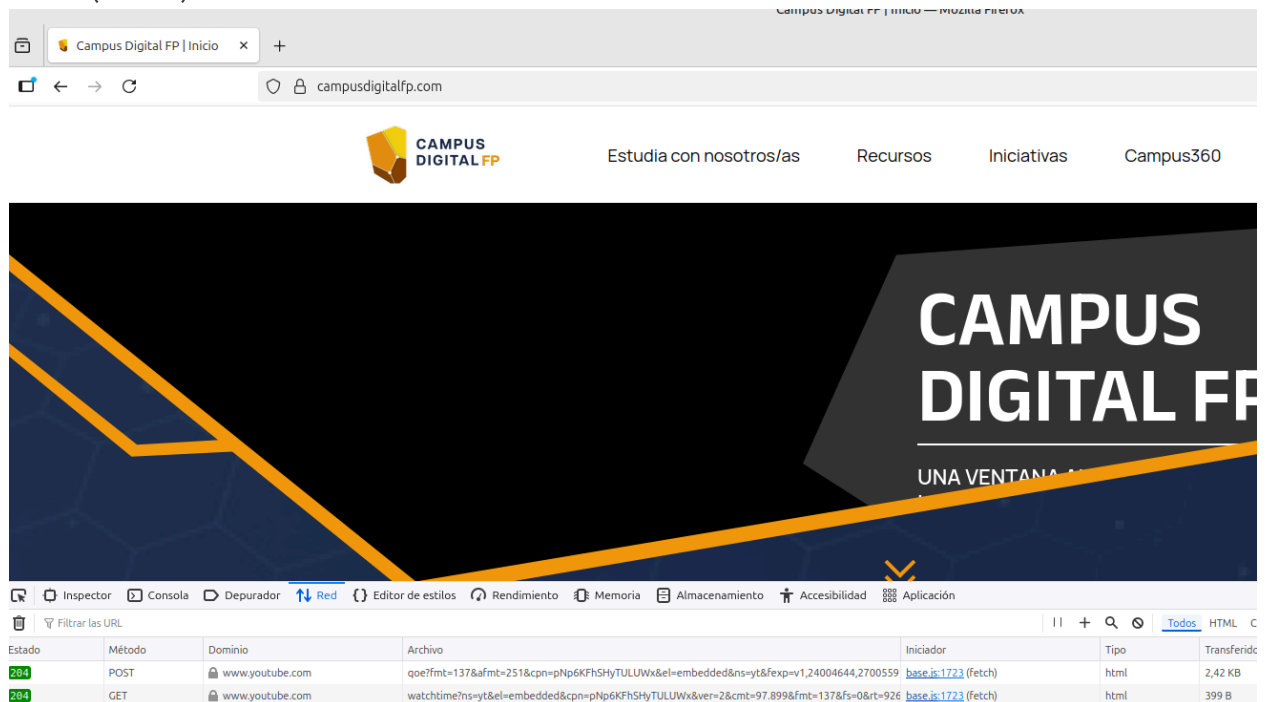
1. Abrimos una pestaña del navegador web e introducimos la URL

[www.campusvirtualfp.com](http://www.campusvirtualfp.com) .





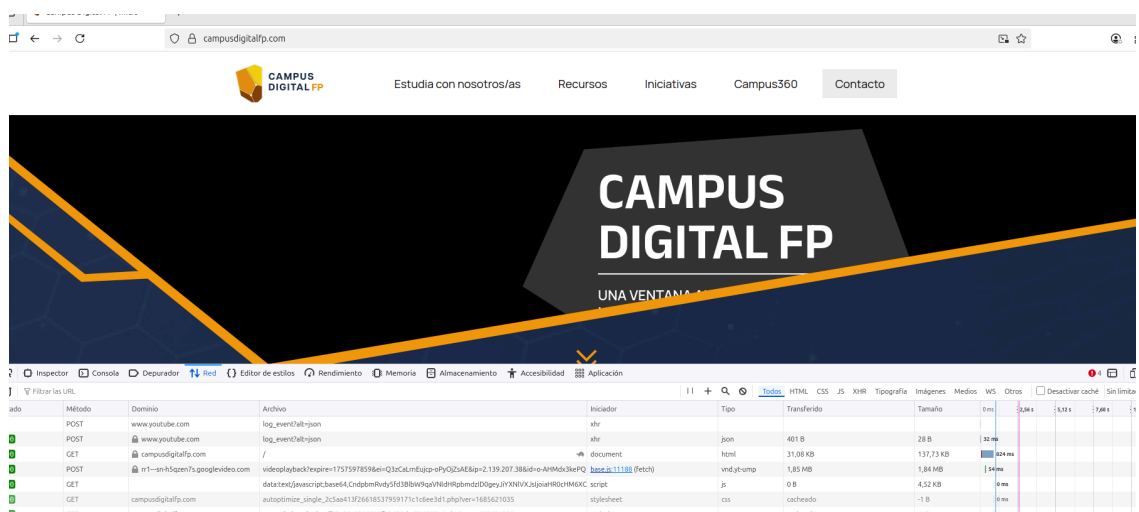
2. A continuación abrimos las herramientas de desarrollador y vamos a la pestaña Network (o Red)



En estos momentos sólo vemos unos pocos datos de llamadas a Youtube que realiza la web para mostrar los vídeos, porque no hemos hecho una petición al servidor con la ventana de herramientas de desarrollador activa.

Por tanto, refrescamos la página (equivalente a hacer una petición al servidor que gestiona [www.campusdigitalfp.com](http://www.campusdigitalfp.com)) y pasamos al punto siguiente.

3. Refrescamos la página y observamos:



Red

El servidor recibe la petición mediante el protocolo de aplicación HTTP, y la procesa mediante su lógica de negocio.

Realmente no podemos saber exactamente qué está sucediendo en el servidor durante este paso, a no ser que tuviésemos acceso al mismo, con los privilegios y herramientas correctas, pero sí podemos averiguar muchos datos mediante herramientas como:

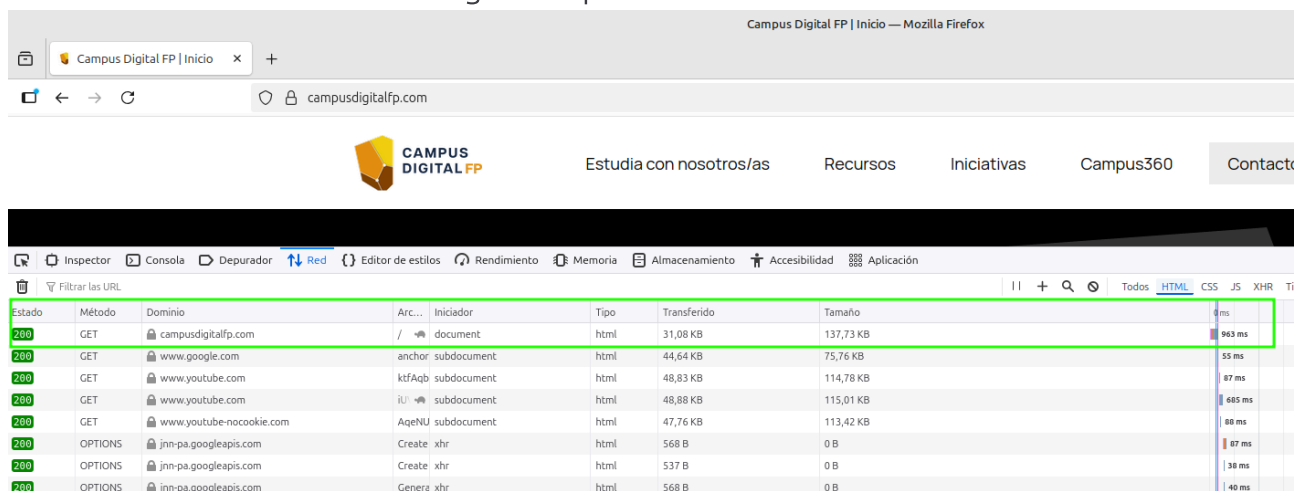
- **whois:** nos indica, entre otros datos, que la IP del servidor es 217.13.88.8 (registro DNS de tipo A).
- **builtwith:** nos indica, entre otras cosas, que la plataforma web dispone de dos servidores web (Apache y nginx).

### Aviso

Existe otro tipo de métodos para conocer más datos sobre el servidor podría implicar prácticas ilegales, en el ámbito de la ciberseguridad.

Este paso se llevaría a cabo en lo que se denomina Back-end.

Vamos a ver el resultado en el siguiente paso.



The screenshot shows a web browser with the address bar displaying 'campusdigitalfp.com'. The page content includes the 'CAMPUS DIGITAL FP' logo and navigation links: 'Estudia con nosotros/as', 'Recursos', 'Iniciativas', 'Campus360', and 'Contacto'. Below the page content, the Firefox Developer Tools Network tab is open, showing a list of network requests. The first request, from 'campusdigitalfp.com' with a status of 200, is highlighted with a green box. The table below represents the data visible in the Network tab.

Estado	Método	Dominio	Arc...	Iniciador	Tipo	Transferido	Tamaño	
200	GET	campusdigitalfp.com	/	document	html	31,08 KB	137,73 KB	0 ms
200	GET	www.google.com	anchor	subdocument	html	44,64 KB	75,76 KB	55 ms
200	GET	www.youtube.com	ktfAqb	subdocument	html	48,83 KB	114,78 KB	87 ms
200	GET	www.youtube.com	ilU	subdocument	html	48,88 KB	115,01 KB	685 ms
200	GET	www.youtube-nocookie.com	AqeNU	subdocument	html	47,76 KB	113,42 KB	88 ms
200	OPTIONS	jnn-pa.googleapis.com	Create	xhr	html	568 B	0 B	87 ms
200	OPTIONS	jnn-pa.googleapis.com	Create	xhr	html	537 B	0 B	38 ms
200	OPTIONS	jnn-pa.googleapis.com	Genera	xhr	html	568 B	0 B	40 ms

El servidor produce una respuesta a la petición del cliente, que la envía a través de internet y recupera nuestro navegador.

Ahora, si consultamos la pestaña Network después de refrescar la URL, podremos ver que han aparecido muchos registros, el primero de los cuales tiene como nombre www.campusdigitalfp.com :

Este registro corresponde a la primera petición que hemos realizado al servidor de www.elche.es . Si pinchamos sobre él podremos ver tanto los datos de la petición realizada, como la respuesta enviada por el servidor a través de Internet, mediante HTTP. Se

Red Accedida

Más abajo, en la misma respuesta, nos indica los parámetros con que se ha realizado la petición. Entre éstos, podemos ver el tipo de petición que hemos hecho (GET), la versión de nuestro navegador web, el lenguaje en que queremos recibir la información, entre otros detalles.

## Headers

# Generación de páginas web

---

## Estáticas

Una página web estática es un documento o conjunto de documentos (generalmente: HTML, CSS, contenido multimedia, código JavaScript) en el que no existe una actualización dinámica de su contenido al interactuar con el sistema (servidor, ya sea remoto o local) que provee el documento/s. Es decir, la misma petición a la misma URL (Uniform Resource Locator), aunque la repitamos en múltiples ocasiones a lo largo del tiempo, siempre va a devolver la misma información (a no ser que la modifique un desarrollador en el lado servidor, manualmente). Puede existir interacción con la página web estática (mediante código JavaScript), en forma de mensajes, eventos, actualizaciones de su apariencia...

En este caso, un navegador web es capaz de representar la página web en una máquina local, sin necesidad de disponer de un servidor web adicional.

## Dinámicas

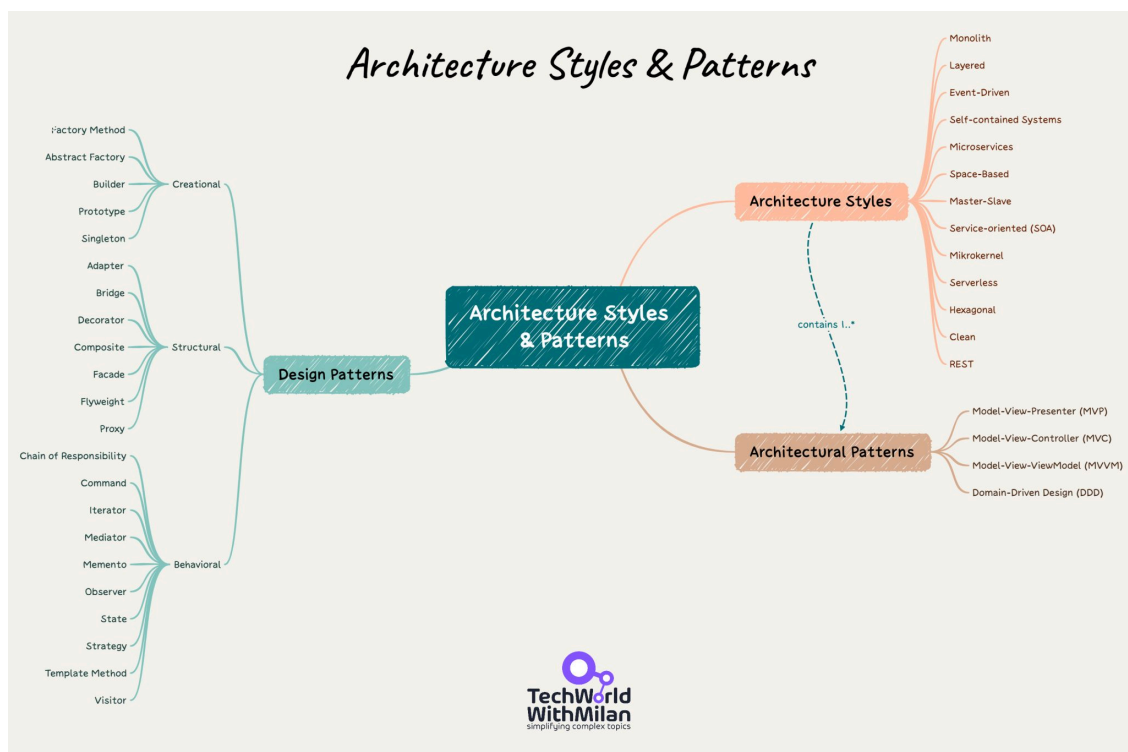
Una página web dinámica puede contener una parte estática, y además el contenido que se muestre dependerá del momento en el cual se realice la petición. Esto es debido a que el servidor conformará dicho contenido dependiendo de los datos de que se disponga en ese momento en un sistema de bases de datos. La comunicación entre el navegador web y el servidor será más compleja, ya que, además de consultar contenidos, se podrán realizar potencialmente operaciones de creación, modificación, y eliminación de datos.

Una aplicación Web es una herramienta software, formada por páginas web dinámicas (aunque también puede contener documentos web estáticos), basada en tecnologías web que la dotan de un carácter dinámico (interactúan con un sistema remoto) haciendo uso de servicios web (basados en la arquitectura TCP/IP), y que proporcionan al usuario un servicio o conjunto de servicios. Sería lo más parecido a una aplicación nativa o de escritorio, pero ejecutada en un navegador web. El hecho de ejecutarse en un navegador web las independiza del sistema operativo en el que se ejecutan, pero también presentan determinadas limitaciones debido a esta independencia.

En este caso, un navegador web NO es capaz de representar la página web en una máquina local sin un servidor web adicional y el resto de componentes que acompañan a esta arquitectura, como sí era el caso de una página web estática.

## Modelos de Arquitectura Software: Monolítica, de Capas, Microservicios y Serverless

Una arquitectura de software se refiere a la estructura organizativa fundamental de un sistema de software. Define cómo se dividen, combinan y coordinan sus componentes para lograr los objetivos del sistema. En el desarrollo de aplicaciones web del lado del servidor (*backend*), existen varias arquitecturas principales.



Arquitect

- **Arquitectura Monolítica:** Es un enfoque tradicional en el que todos los componentes de una aplicación web se agrupan en un solo bloque. La lógica de negocio, la interfaz de usuario y la capa de acceso a datos se encuentran dentro de la misma aplicación. Es fácil de desarrollar y desplegar inicialmente, pero puede volverse complejo y difícil de mantener a medida que la aplicación crece. Todos los componentes se ejecutan en el mismo proceso y comparten recursos. La escalabilidad puede ser un desafío, ya que la aplicación se ejecuta en una sola instancia. Los cambios en una parte de la aplicación pueden afectar a otras partes.

- **Arquitectura de Capas:** Esta arquitectura divide la aplicación en diferentes capas lógicas, donde cada capa tiene una responsabilidad específica. Las capas típicas incluyen la capa de presentación, la capa de lógica de negocio y la capa de acceso a datos. Cada capa se comunica con la capa adyacente a través de interfaces bien definidas. Mejora la modularidad y la reutilización del código, permite cambios en una capa sin afectar a las demás y facilita la escalabilidad y el mantenimiento del sistema.
  - **Capas Físicas (Tiers)** Una capa física o tier corresponde a un componente hardware separado dentro de la arquitectura. Por ejemplo, en una arquitectura de tres capas físicas (3-tier) se distinguen:
    - Servidor web
    - Servidor de aplicaciones
    - Servidor de base de datos
  - **Capas Lógicas (Layers):** Las capas lógicas organizan el código según su función:
    - Presentación: Interfaz de usuario.
    - Negocio/Aplicación: Lógica de negocio y procesamiento.
    - Datos/Persistencia: Gestión y almacenamiento de datos.

Cada capa puede implementarse con diferentes tecnologías y lenguajes, permitiendo flexibilidad y modularidad.

- **Arquitectura de Servicios Web:** Esta arquitectura se basa en la comunicación entre diferentes servicios a través de protocolos web estándar, como HTTP. Cada servicio es una unidad independiente que se puede desarrollar, desplegar y escalar de forma independiente. Los servicios se comunican entre sí para cumplir con los requisitos de la aplicación. Favorece la modularidad y la independencia de los servicios, permite la integración de diferentes tecnologías y lenguajes de programación y facilita la escalabilidad horizontal.
- **Arquitectura Basada en Microservicios:** Es una evolución de la arquitectura de servicios web, donde los servicios se dividen en componentes aún más pequeños y autónomos llamados microservicios. Cada microservicio se enfoca en una tarea específica y se comunica con otros microservicios a través de protocolos ligeros. Cada microservicio se puede desarrollar, desplegar y escalar de forma independiente. Mejora la flexibilidad y la agilidad del desarrollo y permite la adopción de diferentes tecnologías y enfoques dentro de cada microservicio. Un ejemplo notable es la arquitectura de *backend* de Netflix.

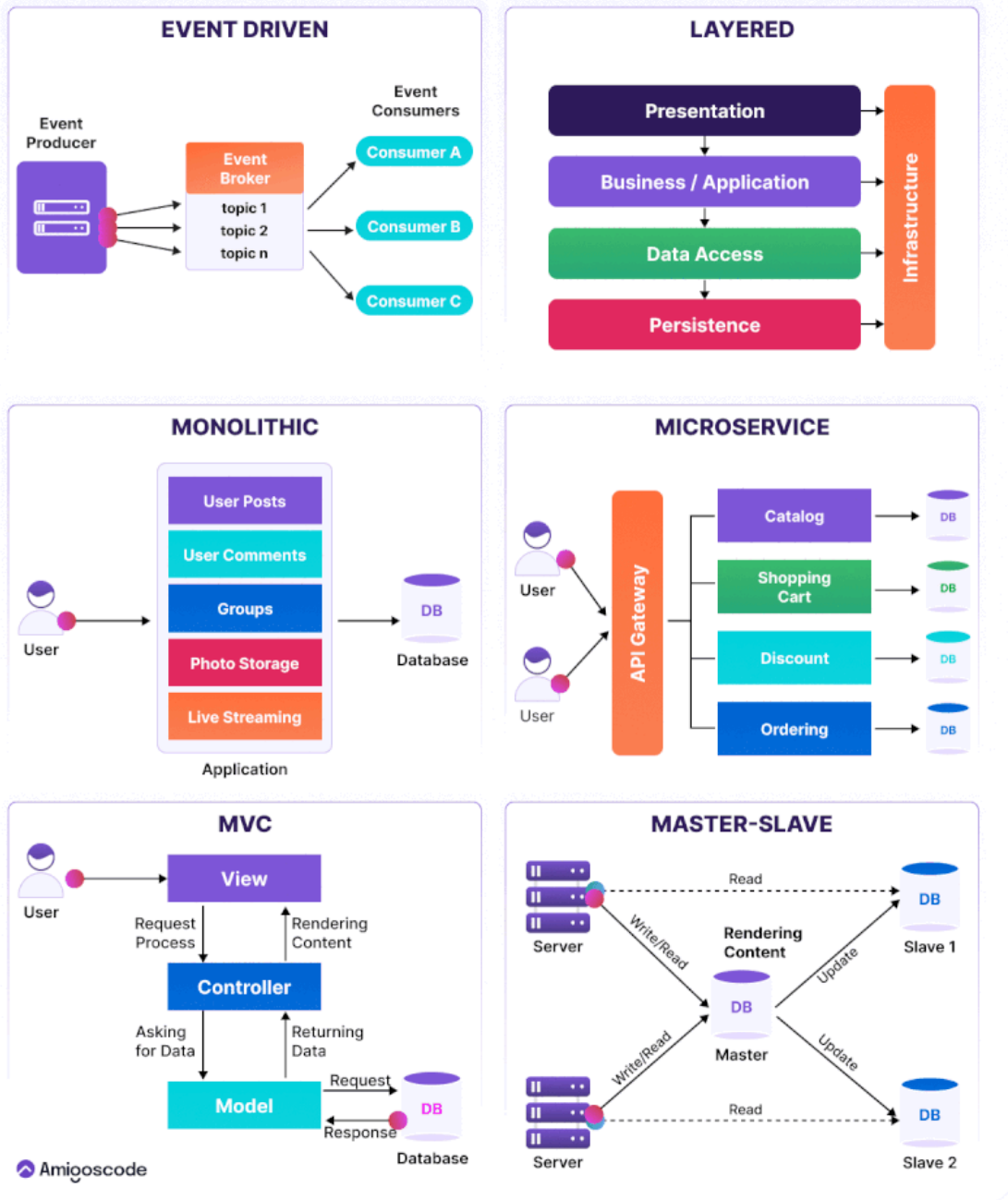
## Información

🤖 Aunque la utilización de arquitecturas basadas en microservicios tuvieron un hype importante durante los últimos años, identificando estas prácticas como la panacea para todo tipo de proyectos, la realidad es que no son adecuadas para todos los casos. Su complejidad y coste de mantenimiento puede ser excesivo para proyectos pequeños o medianos, donde una arquitectura monolítica o en capas puede ser más adecuada.

Actualmente, múltiples empresas están migrando sus arquitecturas de microservicios a arquitecturas monolíticas o en capas, buscando simplificar su mantenimiento y reducir costes, sobretodo en proyectos pequeños o medianos.

- **Arquitectura Serverless:** En este modelo, el proveedor de nube gestiona toda la infraestructura del servidor, y los desarrolladores solo se preocupan por escribir el código de la aplicación. Esto ofrece escalabilidad automática y pago por uso.
- **Service-Oriented Architecture (SOA):** Es un enfoque de diseño de software donde los componentes del sistema se organizan como servicios independientes que se comunican entre sí a través de interfaces bien definidas. Cada servicio es una unidad autónoma que realiza una función específica y puede ser reutilizado en diferentes aplicaciones. SOA promueve la interoperabilidad, la flexibilidad y la escalabilidad al permitir que los servicios se desarrollen, desplieguen y mantengan de forma independiente.
- **Event Driven Architecture (EDA):** En esta arquitectura, los componentes del sistema se comunican mediante eventos. Un componente emite un evento cuando ocurre una acción significativa, y otros componentes pueden suscribirse a estos eventos para reaccionar en consecuencia. Esto permite una mayor flexibilidad y desacoplamiento entre los componentes del sistema.





img

Tabla Comparativa de Arquitecturas Software

Característica	Monolítica	De Capas	Microservicios	Server
Complejidad Inicial	Baja (fácil desarrollo y despliegue)	Media (modular, pero aún una única aplicación)	Alta (gestión y despliegue complejos)	Media (abstracción pero de mayor complejidad)

Característica	Monolítica	De Capas	Microservicios	Serverless
<b>Escalabilidad</b>	Difícil de escalar componentes individuales	Buena (facilita la escalabilidad y mantenimiento)	Independiente por servicio (alta)	Automática (muy alta)
<b>Resiliencia</b>	Un fallo puede afectar toda la aplicación	Un fallo en una capa afecta a su funcionalidad	Alta (fallo de un servicio no afecta al resto)	Alta (puede gestionar y tolerar fallos)
<b>Mantenimiento</b>	Se vuelve complejo a medida que crece	Facilita el mantenimiento	Flexible y ágil, fácil de actualizar individualmente	Reducción de la administración del servidor
<b>Flexibilidad Tec.</b>	Baja (todo en una pila tecnológica)	Baja (puede permitir diferentes lenguajes por capa)	Alta (diferentes tecnologías por microservicio)	Muy alta (independencia tecnológica por función)
<b>Coste</b>	Puede ser bajo inicialmente	Moderado	Puede ser más alto por complejidad de infraestructura	Pago por uso (potencialmente bajo si es esporádico)
<b>Tiempo de Desp.</b>	Largos ciclos de despliegue	Moderados	Agilidad en despliegue de pequeños cambios	Muy rápido (funcionamiento individual)
<b>Comunicación</b>	En memoria (rápida)	En memoria o a través de interfaces bien definidas	Mayor sobrecarga entre servicios (red)	Vía eventos o API Gateway (latencia "arranque frío")
<b>Ideal para</b>	Proyectos pequeños, MVPs	Aplicaciones empresariales con requisitos claros	Aplicaciones complejas, grandes empresas (ej. Netflix)	Funciones esporádicas, microservicios específicos

### Macroservicios vs. Microservicios vs. Serverless vs SOA vs EDA:

El término "Macroservicios" no es una arquitectura formal, pero a menudo se usa para describir aplicaciones que, aunque modularizadas, no alcanzan la granularidad y autonomía de los microservicios, o bien para referirse a arquitecturas monolíticas grandes.

La tendencia hacia los **Microservicios** se debe a la necesidad de construir sistemas más **flexibles, escalables y resilientes** en un entorno de desarrollo ágil. Mientras que un monolito requiere escalar toda la aplicación incluso si solo una pequeña parte tiene alta demanda, los microservicios permiten escalar de forma independiente cada componente. Esto facilita que equipos pequeños trabajen de forma autónoma, elijan sus propias tecnologías y desplieguen con mayor frecuencia y menor riesgo. El fallo de un microservicio no afecta a toda la aplicación, mejorando la resiliencia. Sin embargo, esta flexibilidad viene con una **mayor complejidad de gestión y despliegue**.

**Serverless** va un paso más allá, eliminando la preocupación por los servidores. Aunque puede verse como una evolución de los microservicios, no todos los microservicios son Serverless. Serverless es ideal para funciones cortas y bajo demanda que pueden beneficiarse de la escalabilidad automática y el pago por uso, pero tiene limitaciones de tiempo de ejecución y posibles latencias de "arranque en frío".

**Service-Oriented Architecture (SOA)** es un enfoque más amplio que puede incluir microservicios, pero se centra en la interoperabilidad y la reutilización de servicios a través de una arquitectura orientada a servicios. SOA puede ser más adecuado para organizaciones grandes con sistemas heredados que necesitan integrarse.

**Event Driven Architecture (EDA)** es un enfoque que puede complementar tanto a los microservicios como a las arquitecturas Serverless. En EDA, los componentes del sistema reaccionan a eventos, lo que permite una mayor flexibilidad y desacoplamiento. Esto es especialmente útil en sistemas distribuidos donde la comunicación asíncrona puede mejorar la escalabilidad y la resiliencia.

## Modelo MVC (Modelo-Vista-Controlador)

El patrón **MVC (Model-View-Controller)** es una arquitectura que separa la lógica de negocio, la gestión de datos y la presentación visual. Al separar los componentes en elementos conceptuales permite reutilizar el código y mejorar su organización y mantenimiento. Sus elementos son:

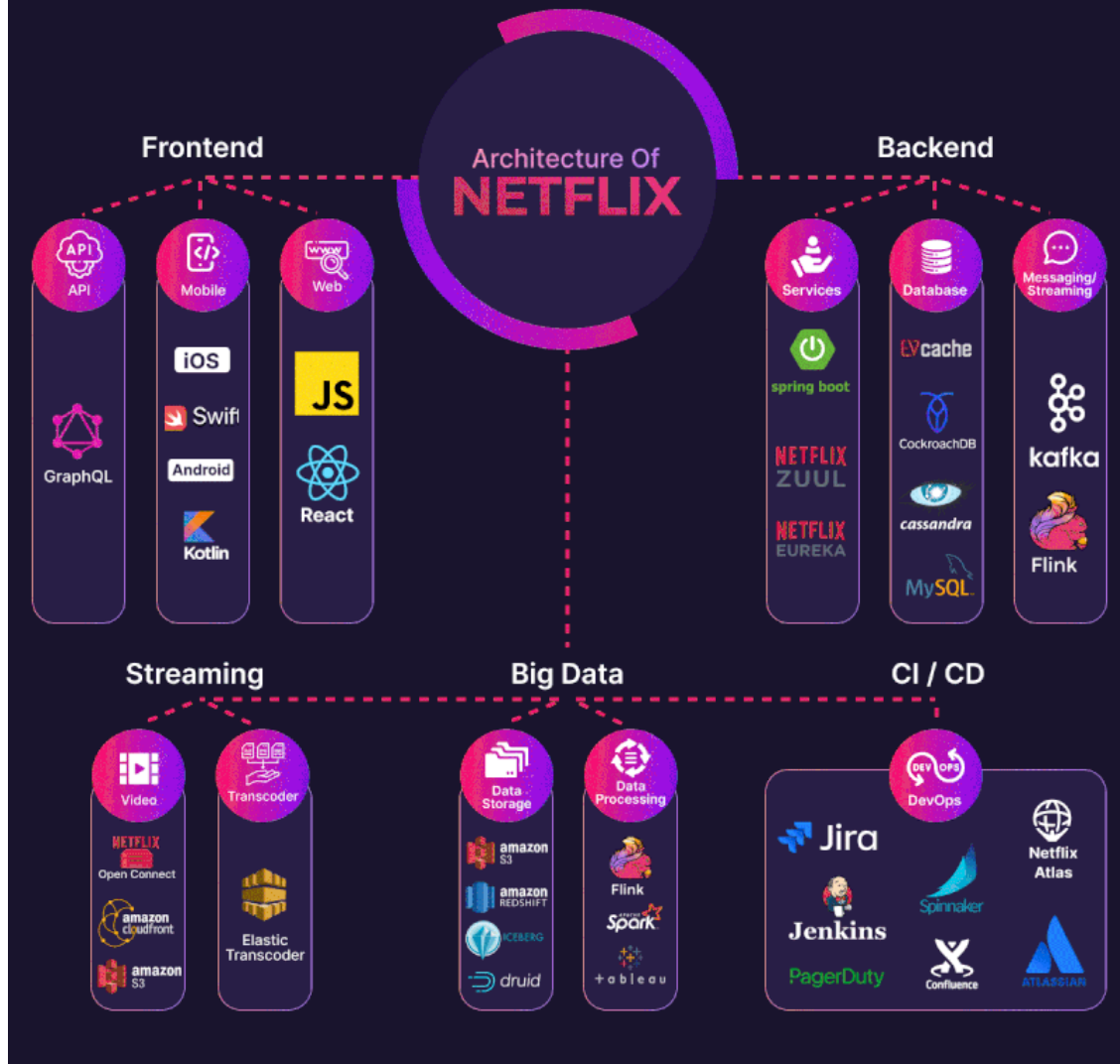
- **Modelo:** Gestiona los datos y su acceso, normalmente conectado a una base de datos.
- **Controlador:** Procesa las acciones del usuario y coordina la interacción entre el modelo y la vista.
- **Vista:** Presenta la información al usuario y recoge sus interacciones.

Esta separación facilita la organización, el mantenimiento y la reutilización del código.

En este modelo, es el servidor el que lleva el peso principal tanto del procesamiento de la información como de su representación. El cliente web se dedica a enviar las peticiones al servidor, recibir la respuesta y representarla en pantalla. La página web (código HTML, JavaScript, etc.) se predetermina en el lado del servidor. Con este modelo, cada petición del cliente al servidor implicará un refresco de la información que se visualiza en la pantalla, aunque su apariencia haya cambiado poco. Esto implica que se vuelvan a descargar todos los datos y ficheros que no se mantengan en la caché del navegador, con lo que los tiempos de respuesta serán mayores. El usuario final apreciará que, por un intervalo corto de tiempo, todos los elementos de la pantalla desaparecen y después se conforma de nuevo la interfaz de usuario. En este caso, se dice que la aplicación no es reactiva. Este modelo de programación MVC se ajustará al primer proyecto del curso.

## **Ejemplo de arquitectura: Netflix**

La arquitectura de backend de Netflix es conocida por ser altamente escalable y resiliente, diseñada para manejar grandes volúmenes de tráfico y garantizar la disponibilidad y el rendimiento de sus servicios. Netflix adopta una arquitectura basada en microservicios, donde las diferentes funcionalidades se dividen en servicios independientes. Cada microservicio se enfoca en una tarea específica y se comunica con otros servicios a través de interfaces bien definidas. Esto permite una mayor flexibilidad, escalabilidad y mantenimiento de los servicios individuales.



netflix

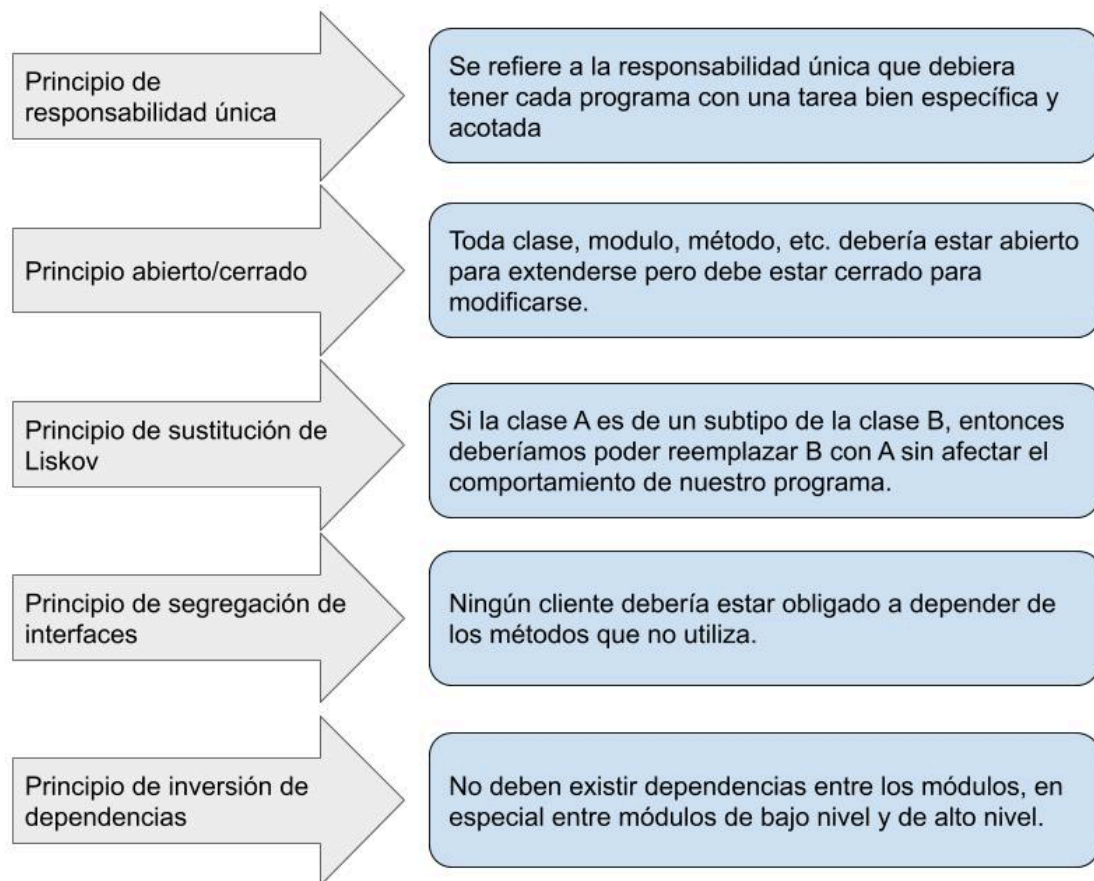
## Patrones y tipos de arquitecturas en Servidor.

Los patrones de diseño son soluciones generalmente aplicables a problemas comunes en el diseño de software. Proporcionan un enfoque probado y estructurado para resolver problemas recurrentes y mejorar la calidad y flexibilidad del código.

## Principios SOLID

Los cinco principios SOLID son un conjunto de reglas y mejores prácticas para el diseño de software orientado a objetos.

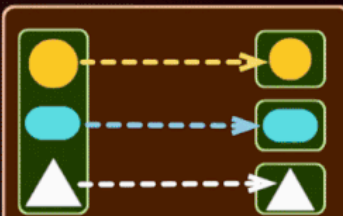
## Video SOLID



Solid

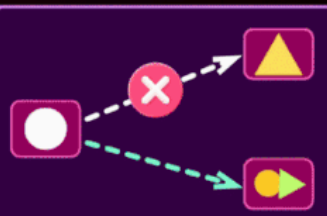


# Foundations of SOLID Principles



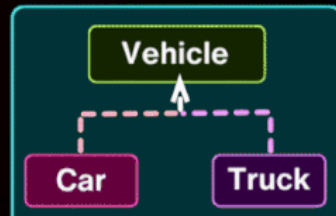
## S Single Responsibility Principle

Each piece of code should have only one job. This keeps code simple and focused, making it easier to understand and maintain.



## O Open-Closed Principle

Code should be open for new features to be added but closed for changes to existing code. It prevents problems when adding new functionality.



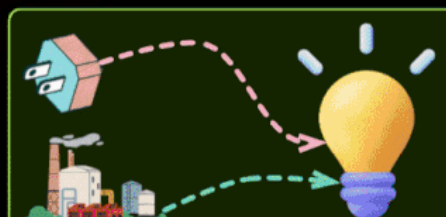
## L Liskov Substitution Principle

Subclasses should be able to replace their base classes without causing issues. This ensures that different parts of code can work together seamlessly.



## I Interface Segregation Principle

Rather than having one big interface, create smaller, specific interfaces. This makes it clear which methods are relevant for each class and avoids unnecessary complexity.



## D Dependency Inversion Principle

Use abstract interfaces to connect different parts of the system, instead of directly tying them together. This promotes flexibility and easier testing.

Solid2

Los principios son:

1. **Principio de responsabilidad única (Single Responsibility Principle, SRP):** Una clase debe tener una, y solo una, razón para cambiar. Esto significa que una clase debe tener solo una tarea o responsabilidad.



```

1  public class Informe {
2      public void generarInforme() {
3          // lógica para generar el informe
4      }
5  }
6
7  public class ImprimirInforme {
8      public void imprimir(Informe informe) {
9          // lógica para imprimir el informe
10     }
11 }

```

En este ejemplo, la clase `Informe` solo tiene la responsabilidad de generar el informe, mientras que la clase `ImprimirInforme` tiene la responsabilidad de imprimir el informe.

2. **Principio abierto/cerrado (Open/Closed Principle, OCP):** Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para la extensión, pero cerradas para la modificación.

```

1  public abstract class Forma {
2      abstract void dibujar();
3  }
4
5  public class Circulo extends Forma {
6      void dibujar() {
7          // lógica para dibujar un círculo
8      }
9  }
10
11 public class Cuadrado extends Forma {
12     void dibujar() {
13         // lógica para dibujar un cuadrado
14     }
15 }

```

En este ejemplo, la clase `Forma` está abierta para la extensión (se puede crear una nueva forma como `Circulo` o `Cuadrado`), pero cerrada para la modificación (no necesitamos cambiar la clase `Forma` para añadir una nueva forma).

3. **Principio de sustitución de Liskov (Liskov Substitution Principle, LSP):** Los objetos de una superclase deben poder ser reemplazados por objetos de una subclase sin afectar la corrección del programa.

```
1 public class Pajaro {
2     public void volar() {
3         // lógica para volar
4     }
5 }
6
7 public class Pinguino extends Pajaro {
8     @Override
9     public void volar() {
10         throw new UnsupportedOperationException();
11     }
12 }
```

java

Este es un ejemplo de violación del principio de sustitución de Liskov, ya que `Pinguino` es una subclase de `Pajaro`, pero no puede volar. Una solución sería tener una clase separada para pájaros que pueden volar.

4. **Principio de segregación de interfaces (Interface Segregation Principle, ISP):** Los clientes no deben ser forzados a depender de interfaces que no usan.

```
1 public interface Pajaro {
2     void comer();
3 }
4
5 public interface PajaroVolador {
6     void volar();
7 }
8
9 public class Pinguino implements Pajaro {
10     public void comer() {
11         // lógica para comer
12     }
13 }
14
15 public class Paloma implements Pajaro, PajaroVolador {
16     public void comer() {
17         // lógica para comer
18     }
19 }
```

java

```

19
20     public void volar() {
21         // lógica para volar
22     }
23 }

```

En este ejemplo, `Pinguino` no está forzado a implementar un método `volar()` que no necesita, ya que hemos segregado las responsabilidades en dos interfaces diferentes.

5. **Principio de inversión de dependencias (Dependency Inversion Principle, DIP):** Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.

```

1     public interface BaseDeDatos {
2         void guardar(String datos);
3     }
4
5     public class MySQLDB implements BaseDeDatos {
6         public void guardar(String datos) {
7             // lógica para guardar datos en MySQL
8         }
9     }
10
11    public class Aplicacion {
12        private BaseDeDatos db;
13
14        public Aplicacion(BaseDeDatos db) {
15            this.db = db;
16        }
17
18        public void guardarDatos(String datos) {
19            db.guardar(datos);
20        }
21    }

```

En este ejemplo, la clase `Aplicacion` no depende directamente de la clase `MySQLDB`. En lugar de eso, ambos dependen de la abstracción `BaseDeDatos`. De esta manera, si queremos cambiar la base de datos en el futuro, solo necesitamos crear una nueva implementación de `BaseDeDatos`, sin tener que cambiar la clase `Aplicacion`.

### Para saber más..

Los principios **SOLID** fueron introducidos por el famoso científico informático Robert J. Martin (también conocido como Uncle Bob) en su artículo de 2000. Sin embargo, el acrónimo SOLID fue introducido posteriormente por Michael Feathers.

[Robert C. Martin](#) publicó en el año 2000 un ensayo titulado “Principios de diseño y patrones de diseño”, en el que sentó las bases de lo que más tarde se conocería como los principios SOLID. [Documento](#) .

En ese artículo no se mencionaba el acrónimo SOLID. Fue **Michael Feathers**, en 2004, quien acuñó el término "SOLID" para referirse a estos cinco principios fundamentales de la programación orientada a objetos, basándose en el trabajo de Martin.

Robert C. Martin también publicó uno de los libros de cabecera que casi todo programador debería leer una vez en la vida: [Clean Code](#) .

## Patrones de Diseño

Un patrón de diseño de software es una solución generalmente aplicable a un problema común en el diseño de software. Los patrones de diseño proporcionan un enfoque probado y estructurado para resolver problemas recurrentes y mejorar la calidad y flexibilidad del código.

Existen varios tipos de patrones de diseño de software, entre ellos:

1. **Patrones de creación:** Estos patrones se centran en la creación de objetos de manera flexible y eficiente. Algunos ejemplos son: Singleton, Factory Method, Abstract Factory y Builder.
2. **Patrones estructurales:** Estos patrones se ocupan de la composición y estructura de las clases y objetos. Algunos ejemplos son: Adapter, Decorator, Composite y Proxy.
3. **Patrones de comportamiento:** Estos patrones se centran en la interacción y comunicación entre objetos. Algunos ejemplos son: Observer, Strategy, Template Method y Command.
4. **Patrones arquitectónicos:** Estos patrones abordan la estructura y organización de sistemas de software a gran escala. Algunos ejemplos son: Modelo-Vista-Controlador (MVC), Capas y Microservicios.

5. **Patrones de concurrencia:** Estos patrones se utilizan para gestionar la concurrencia y la comunicación entre hilos. Algunos ejemplos son: Mutex, Semaphore, Productor-Consumidor y Monitor.

Estos son solo algunos ejemplos de patrones de diseño de software. Cada patrón tiene un propósito específico y puede aplicarse en diferentes contextos para resolver problemas particulares en el diseño y la implementación de software.

Puedes aprenderlos en [Refactoring Guru](#) y [Entornos de Desarrollo 8](#).

#### Para saber más..

Los patrones de diseño fueron popularizados por el libro "Design Patterns: Elements of Reusable Object-Oriented Software" publicado en 1994 por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, conocidos como la "Gang of Four" (GoF).

## El Protocolo HTTP y HTTPS

---

### Características y Ventajas del Protocolo HTTP

El **Protocolo HTTP (HyperText Transfer Protocol)** es la base de la comunicación en la World Wide Web. Es un protocolo no orientado a la conexión, lo que significa que cada petición entre cliente y servidor es independiente y no requiere mantener una conexión continua.

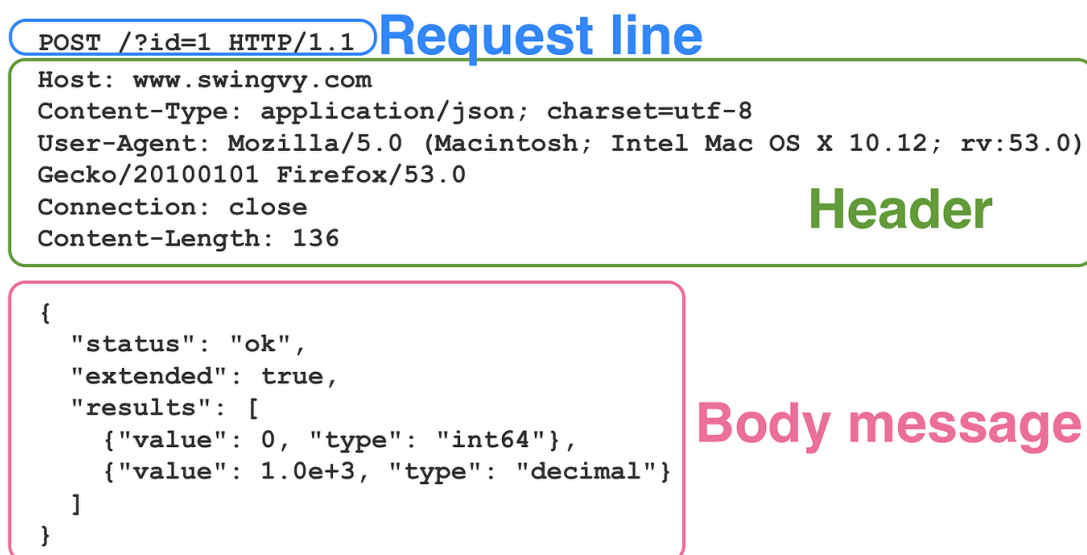
Sus principales características son:

- **Sencillo:** Es en modo texto y fácil de usar directamente por una persona.
- **Extensible:** Se pueden enviar más metadatos que los que están por defecto.
- **Sin estado:** Cada petición es independiente. Esto es un problema para sitios como un carrito de la compra, pero se soluciona con cookies y sesiones.

HTTP es fundamental en arquitecturas distribuidas como los microservicios y es la base para la creación de APIs REST. Ofrece ventajas como la mejora de la velocidad al controlar la caché de las páginas, la autenticación de usuarios, el uso transparente de proxies y el mantenimiento del estado entre peticiones gracias a las sesiones. También permite indicar el formato de lo que se envía, pide y retorna.

## Formato de Peticiones y Respuestas HTTP

La interacción en la web se basa en un intercambio constante de peticiones y respuestas HTTP entre el navegador del cliente y el servidor. Una petición HTTP tiene una primera línea que incluye el método (ej. GET), la ruta del recurso solicitado (ej. /index.html), y la versión del protocolo (ej. HTTP/1.1), seguida de varias líneas con cabeceras que proporcionan metadatos. La respuesta HTTP del servidor comienza con la versión del protocolo (ej. HTTP/1.1), seguida de un código de estado (ej. 200 OK) y un texto que indica el resultado de la operación. Después de una línea vacía, se incluye el contenido del recurso solicitado (ej. HTML).



Protocolo HTTP

## Cabeceras HTTP

Las cabeceras HTTP son mensajes adicionales que se envían tanto en las peticiones como en las respuestas para proporcionar información clave sobre la comunicación.

### Cabeceras de Petición Comunes:

- **Accept** : El formato MIME type en el que se quieren los datos (ej., `text/html` , `application/json` ).
- **Accept-Language** : El idioma preferido para la respuesta (ej., `fr` ).
- **Host** : El dominio al que se dirige la petición, muy útil para alojar varios dominios en un mismo servidor.

- Content-Type : Describe el formato y la codificación de los datos que se envían en el cuerpo de la petición.
- Content-Length : Tamaño en bytes de los datos que se envían.
- User-Agent : Información sobre el navegador del cliente.

### Cabeceras de Respuesta Comunes:

- Content-Type : El formato y la codificación de los datos que se retornan (ej., text/html; charset=utf-8 ), crucial para que el navegador interprete correctamente el contenido.
- Content-Language : El idioma de los datos que se retornan.
- Content-Length : Tamaño en bytes de los datos que se retornan.
- Cache-Control : Cuánto tiempo pueden estar cacheados los datos.
- Server : Indica información del servidor (ej. Apache/2.2.3).

## Métodos/Verbos HTTP (GET, POST, PUT, DELETE, HEAD)

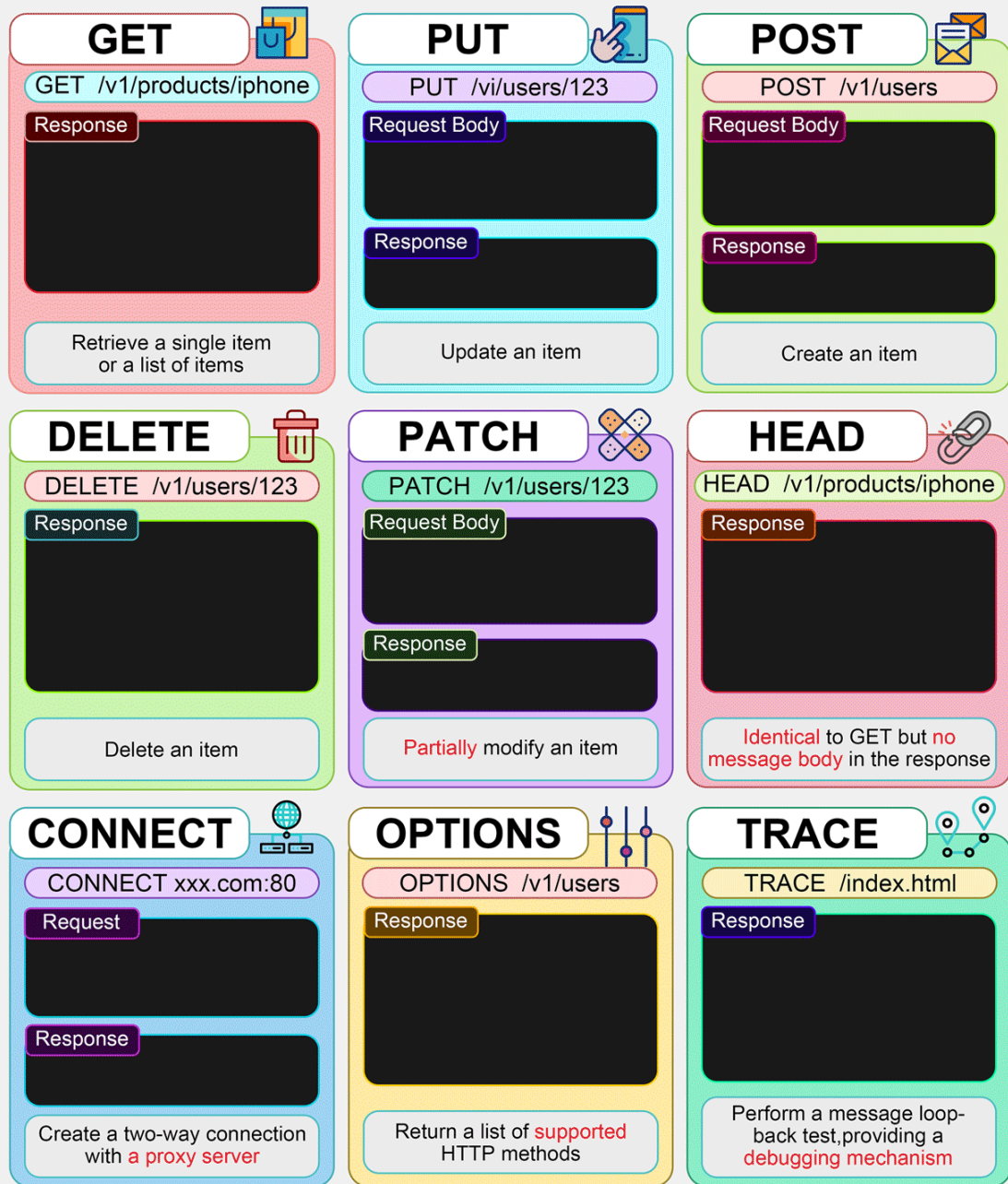
Los métodos HTTP, también llamados verbos, definen la acción que un cliente desea realizar sobre un recurso en el servidor.

- **GET**: Se utiliza para **obtener** o recuperar un recurso. Generalmente, no se envían datos en el cuerpo de la petición; cualquier parámetro se adjunta a la URL como una cadena de consulta (*query string*).
- **POST**: Se usa para **añadir** un nuevo recurso o **enviar** datos al servidor. Los datos se incluyen en el cuerpo de la petición, después de las cabeceras, y no son visibles en la URL.
- **PUT**: Se utiliza para **actualizar** o **reemplazar** completamente un recurso existente en el servidor con los datos proporcionados.
- **DELETE**: Se usa para **borrar** un recurso o entidad específica del servidor.
- **HEAD**: Solicita las mismas cabeceras de respuesta que un método GET, pero sin el cuerpo de la respuesta. Es útil para verificar la existencia de un recurso o sus metadatos sin descargar el contenido completo.



# Top 9 HTTP Request Methods

ByteByteGo



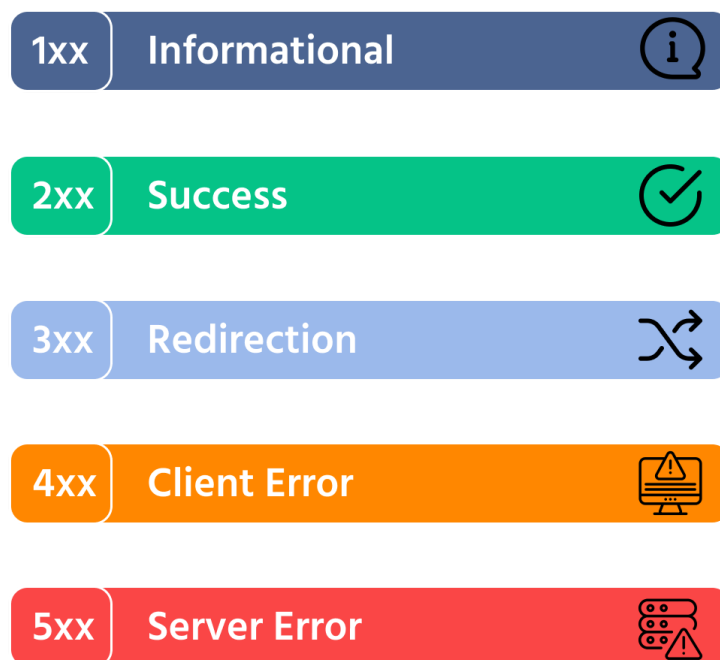
img

## Códigos de Estado HTTP

Después de cada petición, el servidor envía una respuesta que incluye un código de estado HTTP. Este código es un número de tres dígitos que indica el resultado y el estado de la petición.

- **1XX (Informativas):** La petición ha sido recibida y el proceso continúa.

- **2XX (Éxito):** La acción del cliente fue recibida, entendida y aceptada. Por ejemplo, **200 OK** indica que la petición se ha procesado correctamente.
- **3XX (Redirección):** El cliente necesita realizar una acción adicional para completar la petición (ej., el recurso se ha movido).
- **4XX (Error del Cliente):** La petición contiene un error o no puede ser completada debido a un problema en el lado del cliente (ej. **403 Forbidden**, **404 Not Found**).
- **5XX (Error del Servidor):** El servidor falló al completar una petición aparentemente válida.



img

## El Protocolo HTTPS (SSL/TLS y Certificados Digitales)

**HTTPS** (HyperText Transfer Protocol Secure) es la versión segura del protocolo HTTP, esencial para la transferencia confidencial y segura de información entre el cliente y el servidor. A diferencia de HTTP, que transmite datos en texto claro y vulnerable a la interceptación, HTTPS **cifra** la información, asegurando su privacidad.

La seguridad en HTTPS se basa en el uso de **certificados digitales**. Estos documentos electrónicos vinculan una clave pública a la identidad de un propietario (servidor web). Son emitidos por **Autoridades de Certificación (AC)**, que son entidades de confianza que firman digitalmente los certificados para validar su autenticidad. Los navegadores web confían en estas AC y alertan al usuario si un certificado no es válido, está autofirmado o no coincide con el sitio, lo que puede generar advertencias de seguridad.

El proceso de cifrado utiliza el **cifrado de clave pública o asimétrico**. El navegador cifra la información con la clave pública del servidor, y solo el servidor, con su clave privada correspondiente, puede descifrarla, garantizando así la confidencialidad. Los protocolos **SSL/TLS** (Secure Sockets Layer/Transport Layer Security) son los estándares criptográficos que hacen posibles estas conexiones seguras, proporcionando autenticación y privacidad. El cifrado requiere recursos computacionales, lo que puede tener un impacto mínimo en el rendimiento del servidor web. HTTP y HTTPS pueden convivir en un mismo dominio.

## API Web

---

Una API web (*Application Programming Interface*) es un conjunto de reglas y protocolos que permite a diferentes aplicaciones o sistemas comunicarse y compartir datos entre sí a través de la web. Proporciona un conjunto de funciones y métodos que permiten a los desarrolladores acceder y manipular los datos de una aplicación o servicio específico.

En el contexto de las aplicaciones web, una API web permite que el backend de una aplicación exponga ciertas funcionalidades y datos a otras aplicaciones o servicios, como aplicaciones móviles, sitios web o sistemas externos. Esto permite la integración y la creación de aplicaciones más complejas y robustas.

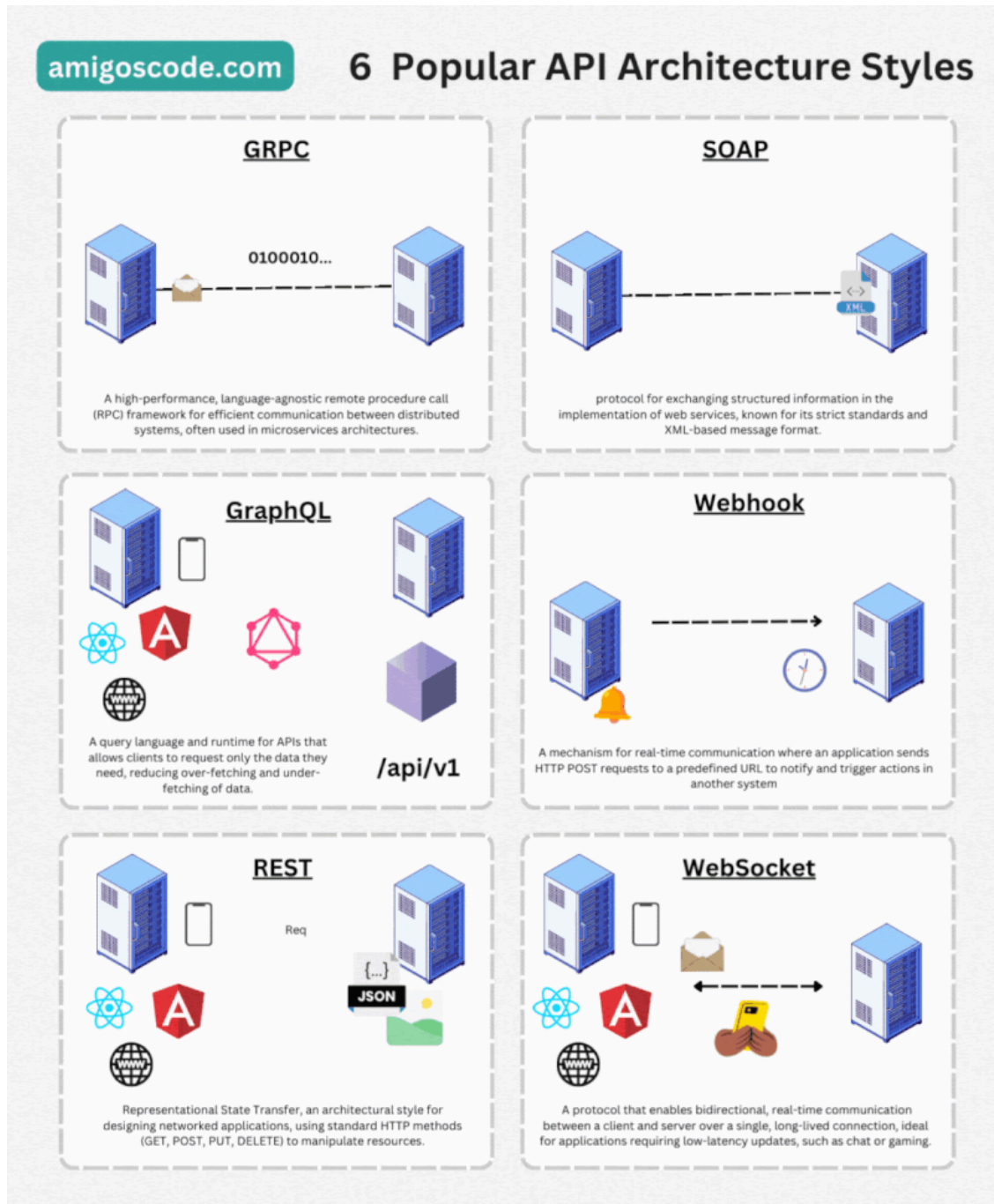
Las API web se basan en protocolos estándar de la web, como HTTP (Hypertext Transfer Protocol), y utilizan formatos de intercambio de datos como JSON (JavaScript Object Notation) o XML (eXtensible Markup Language) para enviar y recibir información.

Algunos ejemplos comunes de API web incluyen:

1. **API RESTful:** Es un estilo arquitectónico que utiliza los métodos HTTP (GET, POST, PUT, DELETE, etc.) para acceder y manipular recursos en un sistema. Se basa en la representación de recursos a través de URLs (Uniform Resource Locators) y utiliza formatos como JSON para el intercambio de datos.
2. **API GraphQL:** Es un lenguaje de consulta y una especificación para las APIs web. Permite a los clientes solicitar y recibir solo los datos necesarios, lo que mejora la eficiencia y reduce la cantidad de datos transferidos.
3. **API WebSocket:** Utiliza el protocolo de comunicación de WebSockets para permitir una comunicación bidireccional y en tiempo real entre un cliente y un servidor a través de una conexión persistente. A diferencia de las API tradicionales basadas en HTTP, que siguen un modelo de solicitud-respuesta, las API con WebSockets establecen una conexión continua entre el cliente y el servidor, lo que permite una comunicación más eficiente y en tiempo real.

## Importante

Las API web son fundamentales en el desarrollo de aplicaciones modernas, ya que permiten la integración de diferentes servicios y la creación de aplicaciones más flexibles y escalables.



apis

## Créditos y reconocimientos

