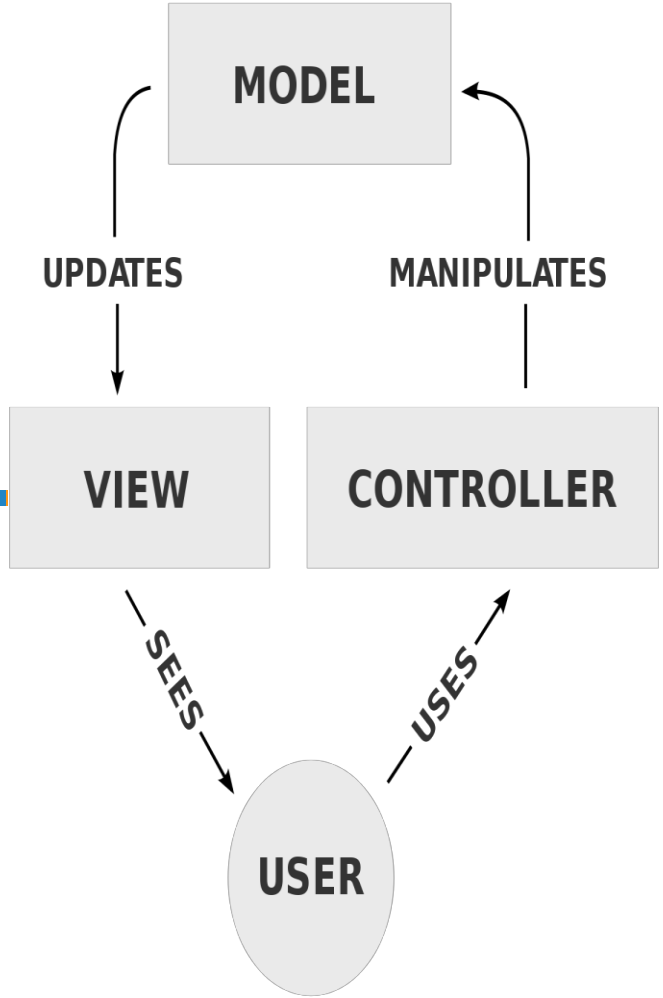
UNIVERSIDAD D CÓRDOBA

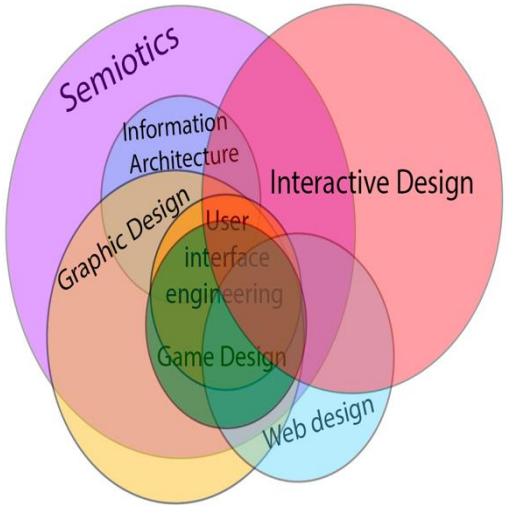
PROGRAMACIÓN WEB – SEMINARIO 3

# Arquitecturas Web




Dr. José Raúl Romero Salguero
jrromero@uco.es

UNIVERSIDAD DE CÓRDOBA




PROGRAMACIÓN WEB – S3.1

# Principios de diseño y concepto de arquitectura

Dr. José Raúl Romero Salguero
jrromero@uco.es

# Principios de diseño de Tim Berners-Lee

- Simplicidad. "*Keep it simple stupid!*". No confundir con la facilidad de entender el diseño
- Modularidad. Dividir nuestro sistema en un grupo de características cercanas y bien comunicadas
- Ser parte de un diseño modular. Debemos considerar nuestro sistema como parte de otro sistema mayor, ofreciendo las interfaces apropiadas (más complejo que modularidad)
- Tolerancia. "*Be liberal in what you require but conservative in what you do*". No implica llegar a romper más allá de lo necesario (ej. dejar de utilizar estándares)

Fuente: https://www.w3.org/DesignIssues/Principles.html

- Descentralización. Se está diseñando un sistema distribuido para la sociedad.
  - Cualquier punto común que esté involucrado en alguna otra operación tenderá a limitar la forma en que escala el sistema, y producirá un punto único de error total
- Test de invención. "*If someone else had already invented your system, would theirs work with yours?*" Relacionado con la modularidad de dentro hacia afuera
- Principio de la potencia mínima. La elección del lenguaje es un criterio de diseño. El extremo de menor potencia suele ser sencillo de diseñar, implementar y utilizar. El extremo de mayor potencia suele ser atractivo y permitir hacer cualquier cosa, sólo limitado por la imaginación del programador. Al principio, los lenguajes eran muy potentes. Hoy en día, se recomienda utilizar el lenguaje menos potente posible. Cuanto menos potente el lenguaje, más se podrá hacer con los datos: "*Elegí HTML para que no fuera un lenguaje de programación porque quería que diferentes programas hicieran diferentes cosas con él*" (T. Berners-Lee)

# Arquitectura software

- En un sentido amplio, la arquitectura del software es el diseño de más alto nivel de la estructura de un sistema, programa y aplicación.
- Los objetivos de la arquitectura del software son:
  - Identificar los módulos principales del sistema, sin considerar aspectos de implementación.
  - Identificar la funcionalidad y responsabilidades de cada módulo.
  - Definir las interacciones posibles entre los módulos haciendo uso de mecanismos de control y flujos de datos, secuenciación de información, protocolos de interacción y comunicación, etc.

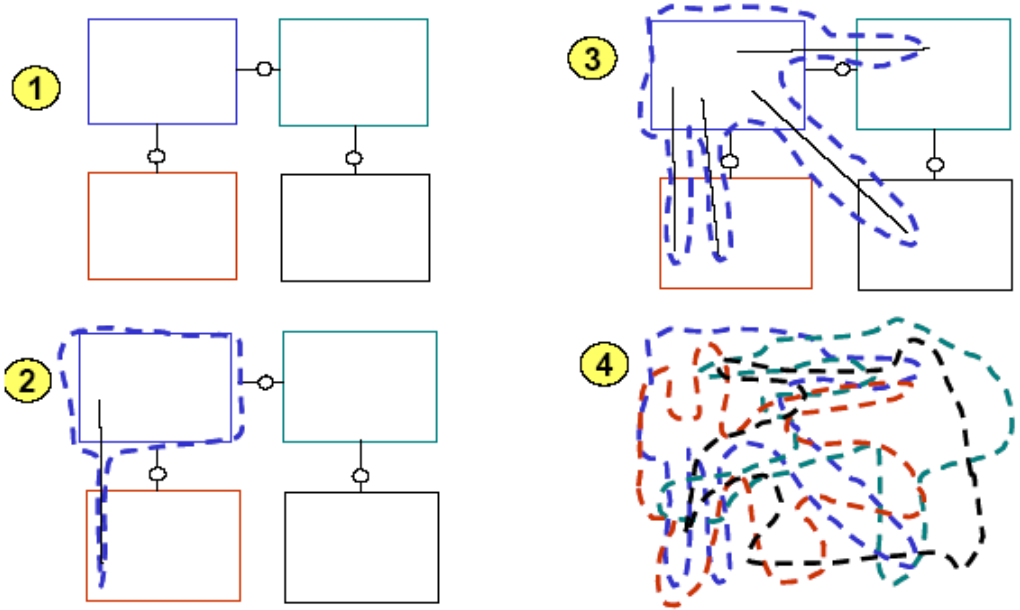



*Figure 1: Illustrating how architectures are undone. Each decision to by-pass interfaces creates coupling. The system quickly devolves into a tangled mass of code.*

Fuente: Architecture as a Business Competency. Bredemeyer Consulting

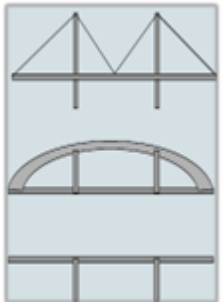
# Arquitectura del Software

*"la arquitectura del software involucra la descripción de los elementos a partir de los cuales se construyen los sistemas, las interacciones entre dichos elementos, patrones que guían en su composición y restricciones sobre esos patrones"*

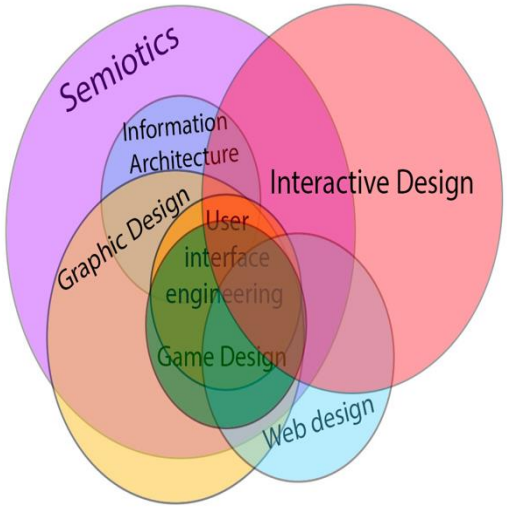
Shaw & Garlan, 1996

# Estilo arquitectónico

- En el diseño arquitectónico, se observan ciertas regularidades en la estructura, estilo y elementos utilizados dando respuesta a demandas similares
- Estas regularidades recurrentes –tomando nomenclatura de arquitectura civil– se denominan estilos arquitectónicos
- Los estilos arquitectónicos representan formas diferentes de estructurar un sistema usando componentes y conectores, siguiendo decisiones esenciales sobre los elementos arquitectónicos y estableciendo restricciones importantes sobre tales elementos y sus posibles relaciones
- Ejemplos: modelo C/S, *peer-to-peer*, arquitectura en capas…



UNIVERSIDAD DE CÓRDOBA




PROGRAMACIÓN WEB – S3.2

# Modelos fundamentales de la web

Dr. José Raúl Romero Salguero
jrromero@uco.es

# Modelo Cliente/Servidor

- En el modelo cliente-servidor (C/S), el sistema se organiza como un conjunto de servicios y servidores asociados, más unos clientes que acceden y utilizan los servicios
- Los principales componentes del modelo C/S son:
  - Conjunto de servidores que ofrecen servicios a otros subsistemas: servidores de impresión, servidores de archivos, servidores de bases de datos, …
  - Conjunto de clientes que llaman a los servicios ofrecidos por los servidores:
    - Invocan los servicios ofrecidos por los servidores mediante un protocolo HTTP de **petición-respuesta**
    - Pueden existir **múltiples instancias** de un cliente ejecutándose concurrentemente
    - Tienen que conocer los nombres de los servidores disponibles y los servicios que suministran, pero los servidores no conocen a los clientes

# Modelo Cliente/Servidor__ Ventajas e inconvenientes

- Este modelo ofrece notables ventajas:
  - Flexibilidad, pudiéndose ejecutar sobre distintas plataformas hardware y software
  - Fácil integración de elementos
  - Fácil mantenimiento local
- Debemos considerar los inconvenientes que plantea:
  - Difícil mantenimiento a nivel global
  - Los servidores son potenciales cuellos de botella
  - Dificulta el manejo de errores y consistencia de datos si se replican (*mirroring*)
  - Difícil distribución de datos
  - Baja confidencialidad y eficiencia. Empeora si hay duplicidad de datos

# Modelo Cliente/Servidor_ _ Flujo básico

- El cliente genera la solicitud - *request*:
  - Un usuario escribe una URL en el agente (*browser*) y hace clic
  - El agente envía una solicitud HTTP – `GET /producto?id=15`

- El servidor recibe la solicitud en el puerto habilitado (80, por defecto) y la procesa:
  - Puede simplemente buscar un fichero estático (servidor web)
  - Puede ejecutar lógica y generar una respuesta dinámicamente (servidor de aplicaciones)

- El servidor emite la respuesta HTTP – *response*
  - La respuesta contiene una cabecera (metadatos) y el cuerpo
  - La respuesta la procesa el cliente

# Modelo Cliente/Servidor__
## *Rendering-1*

- ***Rendering*** es el proceso de transformar código (instrucciones) y datos en una representación visible para el usuario, esto es, hace que el código y los datos se conviertan en la página (*layout*, píxeles, etc.) que se ve en el agente
  - ❑ Depende de la carga y de la distancia física
  - ❑ Incide directamente en la experiencia de usuario (sensación de "lentitud")

- ***Rendering* en el navegador**:
  - ❑ Descarga **HTML**, **CSS** y **javascript**
  - ❑ Construye **DOM** y **CSSOM**
  - ❑ Une ambas estructuras en el **árbol de render** y calcula el ***layout***
  - ❑ Dibuja la página en pantalla

# Modelo Cliente/Servidor__ *Rendering-2*

- ***Rendering*** en aplicaciones actuales:
  - ❑ En los *frameworks* actuales, "***rendering***" significa construir la interfaz de usuario a partir de datos y plantillas
  - ❑ Puede hacerse en el cliente (**CSR**, *client-side rendering*) o en servidor (**SSR**, *server-side rendering*)
  - ❑ También puede hacerse antes del despliegue (**SSG**, *static site generator*)

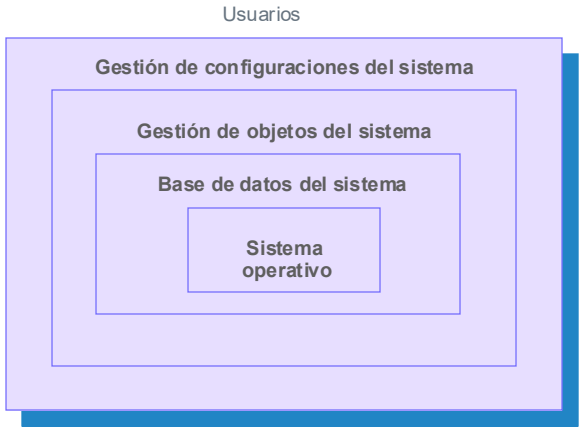
```
Datos: { nombre: "Global Tronics Manta Burrito", precio: 29.99,
         link : "https://www.amazon.es/dp/B08YF5HSNT" }

Plantilla: <h1><a href="{link}">{nombre}</a></h1><p>A {precio}€</p>
```

# Modelo Cliente/Servidor__ Términos básicos

- **Latencia**: tiempo que tarda el mensaje en ir del cliente al servidor, procesarse y volver
  - Depende de la carga y de la distancia física
  - Incide directamente en la experiencia de usuario (sensación de “lentitud”)
- **TTFB** (*Time To First Byte*):
  - Tiempo desde que el navegador hace la petición hasta que recibe el primer byte de la respuesta
  - Si es bajo (<100ms), servidor y red responden rápido
  - Si es alto (>1s), congestión en red o renderizado lento en el servido
- **Caché**: Almacenamiento temporal de respuestas
  - Tipos: caché en navegador, caché en servidor, caché CDN
  - Clave en la mayoría de las soluciones actuales

# Arquitectura en capas

- La arquitectura en capas (también denominada arquitectura estratificada) modela la interacción entre los subsistemas organizando un sistema en una serie de capas (*layers*)
- Cada capa presta servicios a la capa inmediatamente superior y actúa como cliente de la inferior
- Los conectores se definen mediante los protocolos que determinan las formas de la interacción entre cada par de capas
- Este estilo soporta el desarrollo incremental de sistemas:
  - Cada capa desarrollada queda disponible para los usuarios o para incorporar nuevas capas superiores

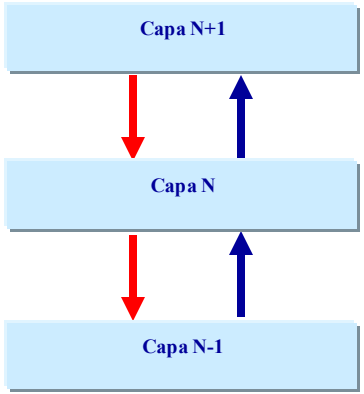



**Modelo de capas de un sistema de gestión de versiones.**
Fuente: [*Sommerville, 2005: Figura 11.4*]

# Arquitectura en capas__ Ventajas e inconvenientes

☺ La arquitectura estratificada presenta las siguientes ventajas:

- ❑ La arquitectura es cambiable y portable
- ❑ Si la interfaz de una capa se mantiene, la capa es reemplazable por otra
- ❑ Si la interfaz se cambia, sólo se afectará la capa adyacente.
- ❑ Para implementar el sistema en otras computadoras sólo es necesario recodificar las capas internas (más dependientes de la máquina)

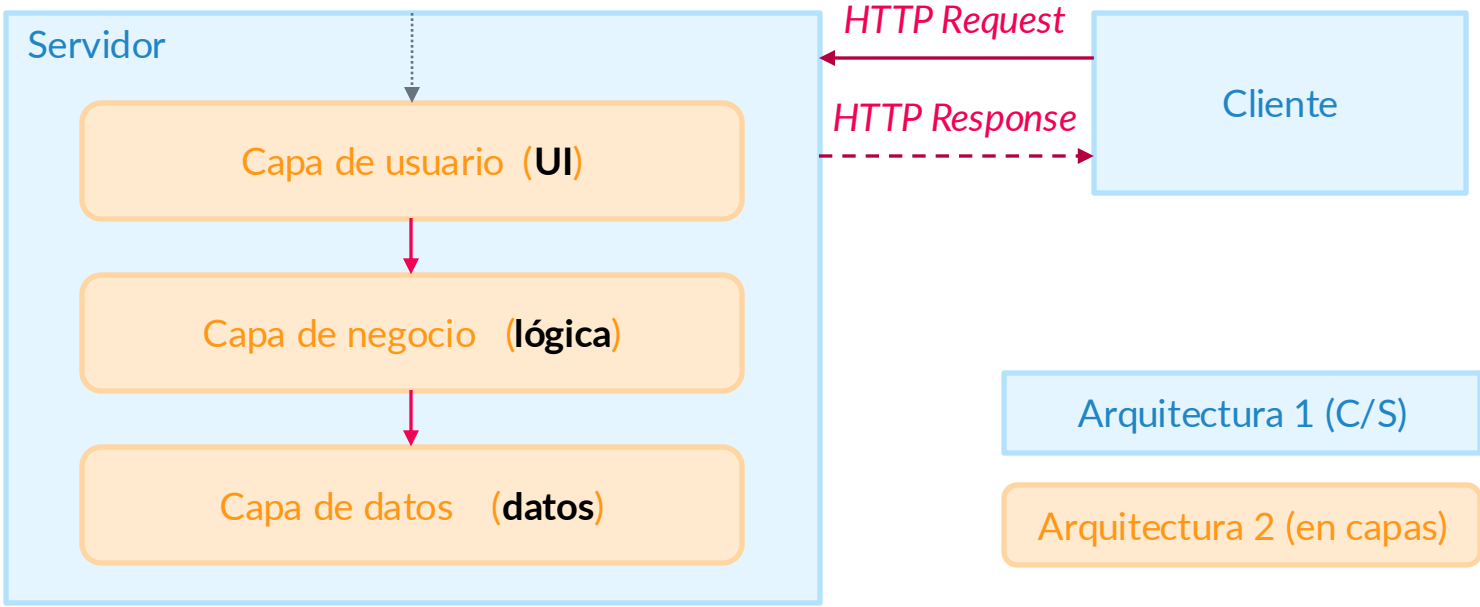



☹ Pero también presenta inconvenientes:

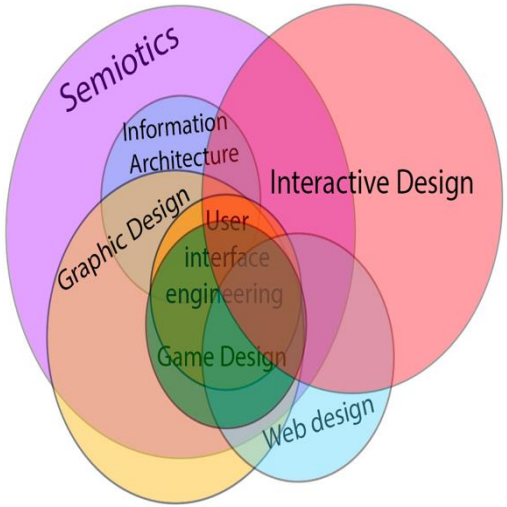
- ❑ Es difícil estructurar los sistemas en capas (p.ej., el usuario puede requerir el acceso a capas internas, como la base de datos, lo que pervierte el modelo)
- ❑ El rendimiento puede resultar afectado por los múltiples niveles de interpretación de órdenes y protocolos que se requieren

# Arquitectura C/S en capas

En desarrollo web se utiliza el estilo jerárquicamente heterogéneo C/S en capas



UNIVERSIDAD D CÓRDOBA




PROGRAMACIÓN WEB – S3.3

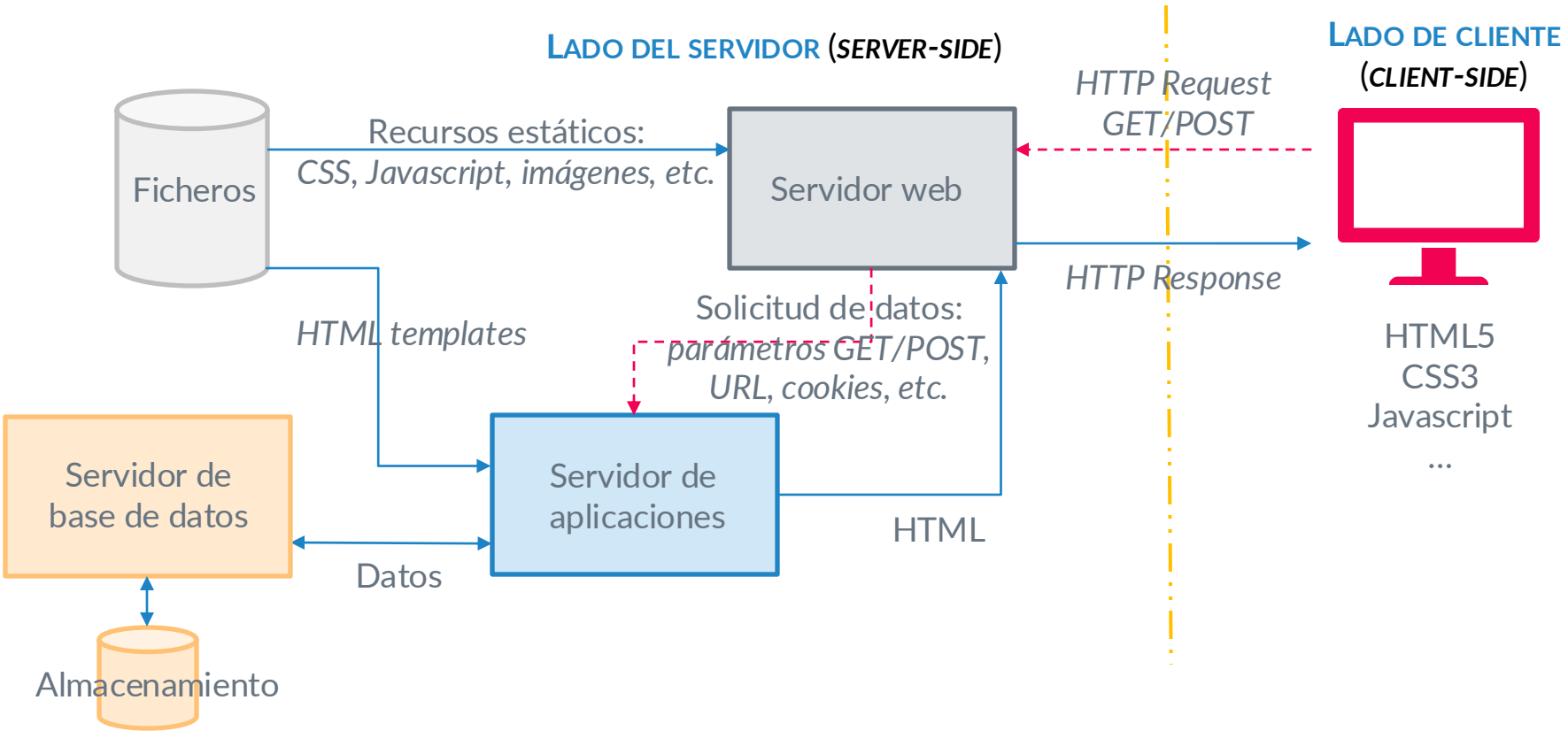
# Localizaciones en la web

Dr. José Raúl Romero Salguero
jrromero@uco.es

# Sitios web dinámicos

- La respuesta devuelta por el servidor (*HTTP Response*) se genera para cada petición en base a los datos específicos del *HTTP Request*
- Por ejemplo, si el catálogo está almacenado, la solicitud identificará el ID del producto, que se buscará en la base de datos (no son páginas individuales pre-creadas) y se construirá dinámicamente la respuesta al cliente insertando los datos en una plantilla de HTML
  - Una plantilla (*HTML template*) puede generar miles de páginas
  - Fácilmente extensible, modificable
  - Fácil de implementar servicios de búsqueda
  - Necesita la instalación de un **servidor de aplicaciones** para construir las páginas a partir de las plantillas

# Sitios web dinámicos



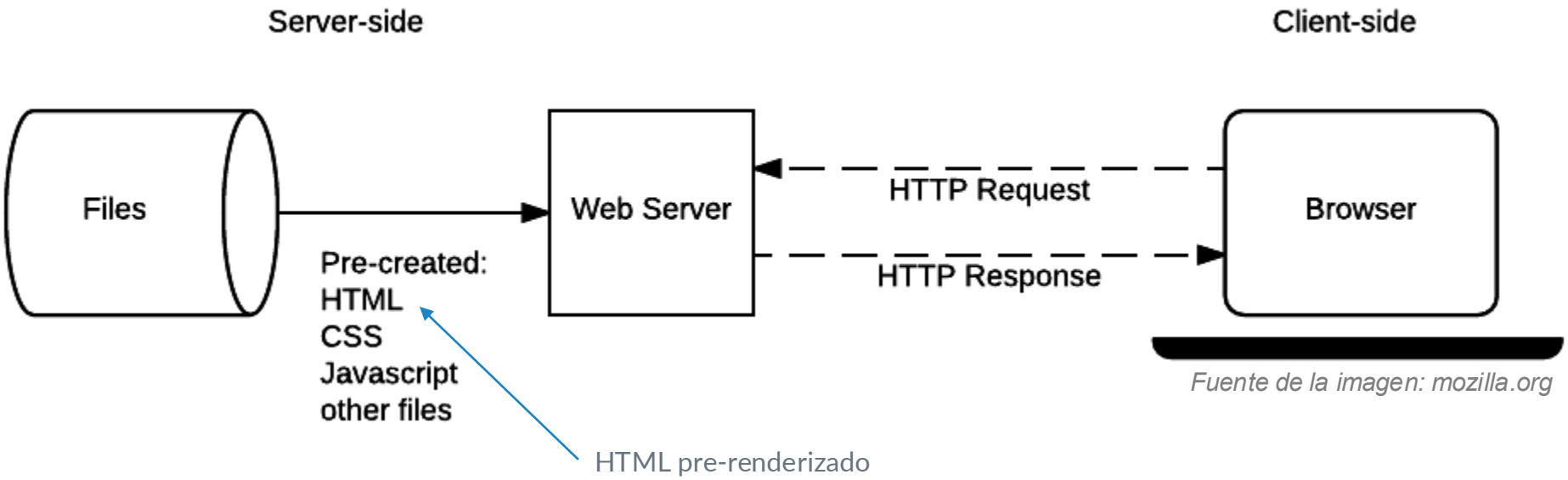
# Sitios web dinámicos

- Los servidores de aplicaciones reciben una solicitud GET/POST y construyen dinámicamente el recurso que sirve de respuesta HTTP
  - No sólo generan HTML, también pueden generar otros tipos de medios, como texto, ficheros PDF, CSV, JSON, XML, etc.
  - Permiten gestionar la redundancia de datos
  - Ofrecen características adicionales: alta disponibilidad, balanceo de carga, gestión de usuarios y permisos de acceso, seguridad, gestión centralizada, gestión de caché, etc.
- Constituyen un modelo de capa de servicio, por lo que ayudan a los desarrolladores web a focalizarse en la lógica de negocio
  - Se desarrolla en lenguajes tipo Java (J2EE), .NET, PHP, etc.

# Sitios web dinámicos

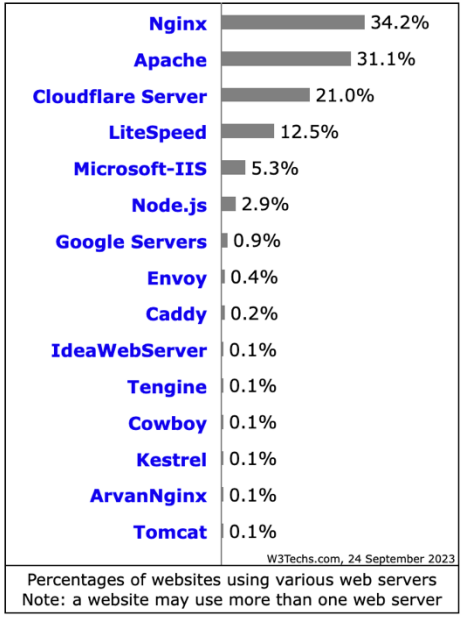
- Variedad de servidores de aplicaciones – según lenguaje: https://en.wikipedia.org/wiki/List_of_application_servers
  - En el lado de cliente, **javascript** es el lenguaje más utilizado
  - En el lado de servidor, **php** es el lenguaje más utilizado
  - En el lado de servidor, **Java** (J2EE – *Java 2 Enterprise Edition*) es el lenguaje utilizado en sitios empresariales
  - Es importante conocer no sólo las características del lenguaje (incluyendo posibles *frameworks* de desarrollo), sino también las del servidor de aplicaciones
- ¡**OJO**! HTML y CSS **no** son lenguajes de programación, sino lenguajes de marcas

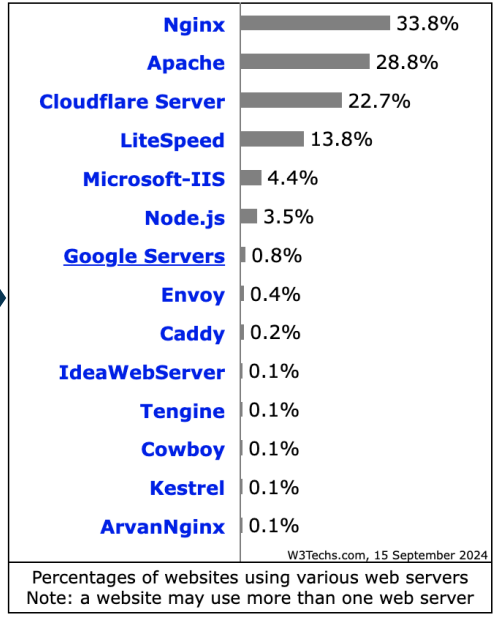
# Sitios web estáticos

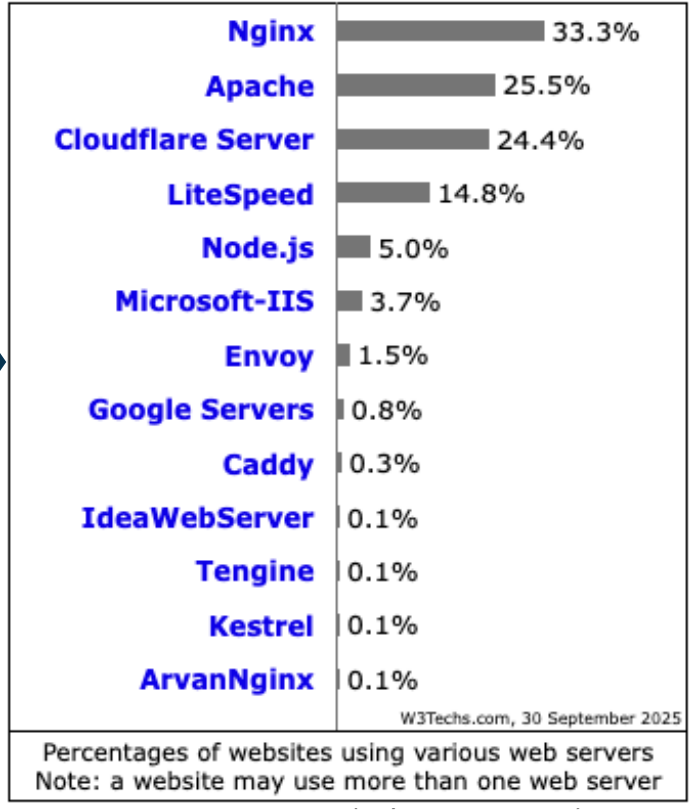



*Fuente de la imagen: mozilla.org*

# Servidores web



# SSG: *Static Site Generators*

- **Técnica de pre-renderizado** en la que el contenido de un sitio web se convierte en archivos HTML estáticos en el momento de compilación
- Cada página se genera antes del despliegue y se sube a un servidor web o a un CDN

✅ Ventajas:

**Rendimiento excelente**: HTML servido directo desde CDN → bajo TTFB.

**Escalabilidad**: una vez generado, puede atender millones de visitas sin sobrecargar servidor.

**Seguridad**: no hay *backend* ejecutando lógica en *runtime*.

**SEO-friendly**: el HTML ya incluye contenido, visible por *crawlers*.

# SSG: *Static Site Generators*

SSG es pre-renderizar el sitio web completo en el *build*, de manera que en producción solo entregamos archivos HTML estáticos. Es ultra rápido y escalable, pero menos flexible para contenido dinámico.

❌ Inconvenientes:

**Contenido poco dinámico**: si la página cambia con frecuencia, hay que hacer *rebuild* del sitio.

**Tiempo de *build***: en sitios con miles de páginas, el *build* puede tardar mucho.

**Datos en tiempo real**: no es ideal para *dashboards*, *ecommerce* con stock cambiante, etc.

**ISR** (*Incremental Static Regeneration*) aparece como solución intermedia

# ISR: *Incremental Static Regeneration*

- Técnica que permite **actualizar páginas estáticas individualmente,** bajo demanda o tras un tiempo definido, sin reconstruir todo el sitio.
- Introducida por el *framework* **`next.js`** - punto intermedio SSG - SSR
  - Se sirven páginas estáticas desde CDN (**SSG**)
  - Pueden regenerarse en segundo plano de forma incremental (**SSR**)

✅ Ventajas:

**Rapidez**: HTML estático desde CDN → bajo TTFB

**Escalabilidad**: solo se regeneran páginas que cambian, no todo el sitio como en SSG

**SEO-friendly**: el HTML disponible desde la primera `response`

❌ Inconvenientes:

**Complejidad añadida**: Políticas de regeneración, además de invalidar caché y aspectos de consistencia

**Tiempo de *build***: en sitios con miles de páginas, el *build* puede tardar mucho.

**Ligado a frameworks**: funciona con Next.js, Gatsby y frameworks muy concretos

# *Single-page application* (SPA)

Una **SPA** es un tipo de aplicación web de cliente grueso, que no requiere recarga de la página durante su uso (p.ej. Gmail), y aprovecha tecnologías AJAX y de (micro-)servicios

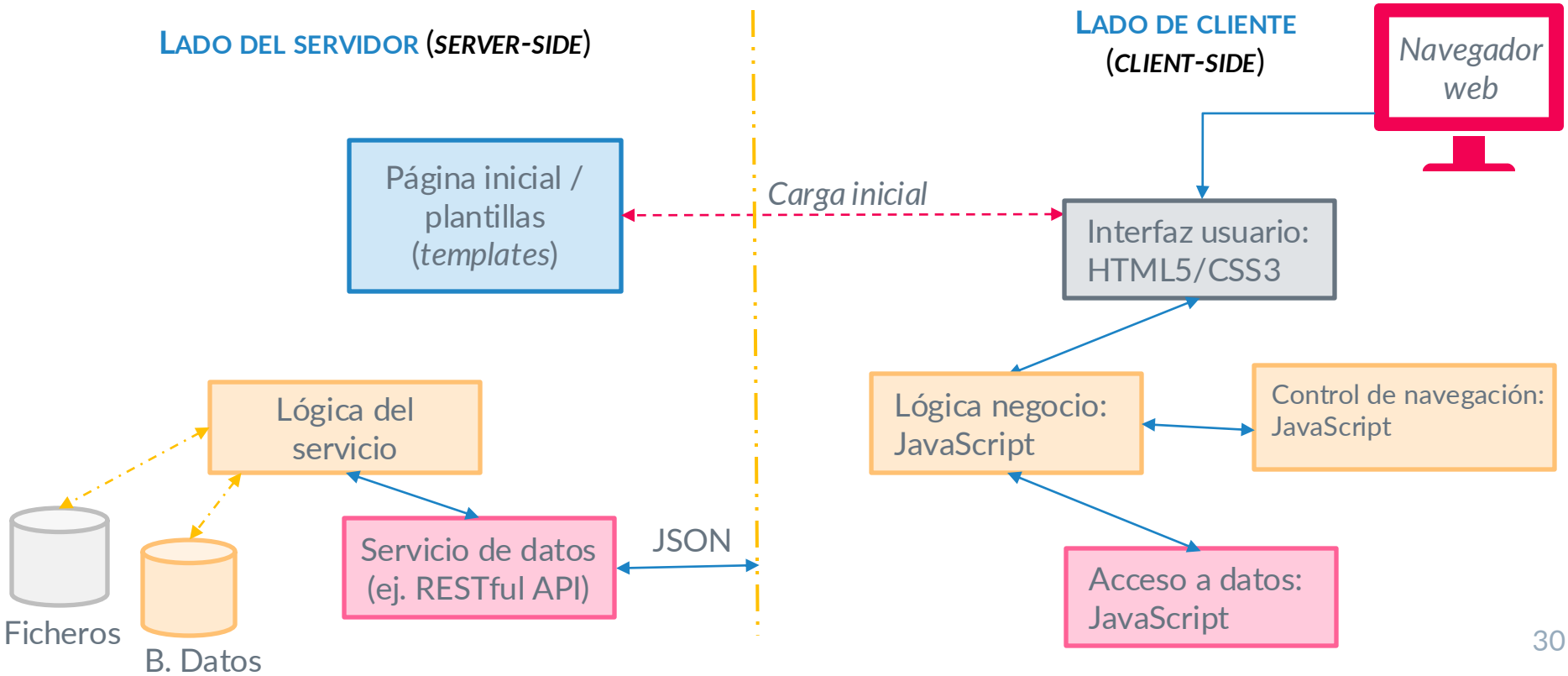
## Ventajas

- Carga rápida (la mayoría de los recursos se cargan al principio)
- Desarrollo más simple, más sencillo de depurar y monitorizar
- Permite transformar a aplicación móvil
- Puede guardar datos en caché de forma efectiva (permitiendo trabajo off-line)

## Inconvenientes

- En caso de optimización SEO, es preferible la renderización en el lado del servidor – mayor dificultad de desarrollo
- SPA es menos seguro (p.ej. *cross-site scripting*, XSS)
- Puede presentar problemas en la pila de memoria en Javascript, que ralentice la aplicación

# *Single-page application* (SPA)



# Sitios Vs Aplicaciones Vs SPAs__

## El dilema del *rendering*

La cuestión no es QUÉ se renderiza (contenido), sino dónde y cuándo se hace

- **SSR**: en servidor antes de enviar la página
- **CSR**: en navegador después de recibir HTML mínimo y ejecutar javascript

Antes de las SPAs, todo ocurría en el servidor (SSR por defecto):

✅ HTML completo (bueno para SEO), primera carga rápida (contenido ya construido)

❌ Cada cambio de página implica recarga completa y navegación menos fluida

En las **SPAs**, la aplicación se carga una única vez y la navegación interna ocurre en el cliente (CSR) obteniendo contenido dinámico vía APIs

- El servidor devuelve un `index.html` mínimo y se construye con javascript la interfaz de usuario, buscando datos con APIs
- Experiencia fluida (similar a una app nativa) y menos tráfico con servidor (solo datos)

# Recopilemos…

| Estrategia | SEO | Rendimiento inicial | Refresco de datos | Escalabilidad | Complejidad |
|---|---|---|---|---|---|
| CSR | Débil | Lento | Muy bueno | Alta | Intermedia |
| SSR | Bueno | Rápido | Siempre actualizados | Limitada al *backend* | Alta |
| SSG | Bueno | Muy rápido | Requiere *rebuild* | Muy alta (todo en *build-time*) | Baja |
| ISR | Bueno | Rápido | Refresco cada x tiempo | Alta | Intermedia |

# Servicios web

La esencia de los servicios web es la **reutilización** de un recurso de computación o de información por la misma o distintas aplicaciones, accedido mediante una representación estándar –definida por la interfaz– y siempre independiente del cliente que lo utiliza

- Aligera las aplicaciones web – desarrollo, mantenimiento, etc.
- Los servicios pueden ser de desarrollo propio o de terceros y se vinculan dinámicamente en ejecución, lo que facilita su disponibilidad y reemplazo
- Los servicios web permiten el pago por uso y limitación de acceso

Los **servicios web** son componentes, débilmente acoplados e independientes de la implementación

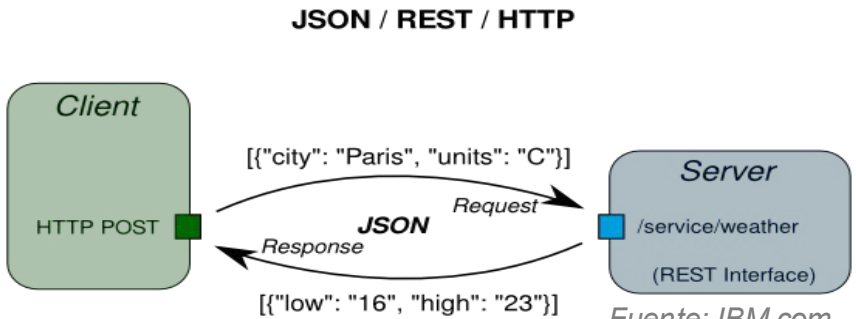
# Servicios REST

Servicios REST (*Representational State Transfer*)

- Funcionan únicamente sobre HTTP con métodos GET, POST, PUT, DELETE (operaciones CRUD)
  - Ofrece una semántica mejor definida, con operaciones ligadas a los ***endpoints***
- Estructura flexible de los mensajes, no fuertemente tipados, lo que permite comunicar múltiples formatos y tipos de medios
  - Desventaja: Posibles errores en el caso de implementaciones con lenguajes fuertemente tipados (Java, C#)
- **Ventajas**: Servicios más livianos e interpretado de forma natural en lenguajes como Javascript → Adoptado en Cloud

# Servicios REST

- **REST** utiliza los métodos HTTP de forma explícita, ofreciendo servicios sin estado
- Permite exponer las URIs como una estructura de directorios, siendo la base de una API/REST
- Transfieren los datos de la respuesta en formato XML, JSON (*JavaScript Object Notation*), o ambos



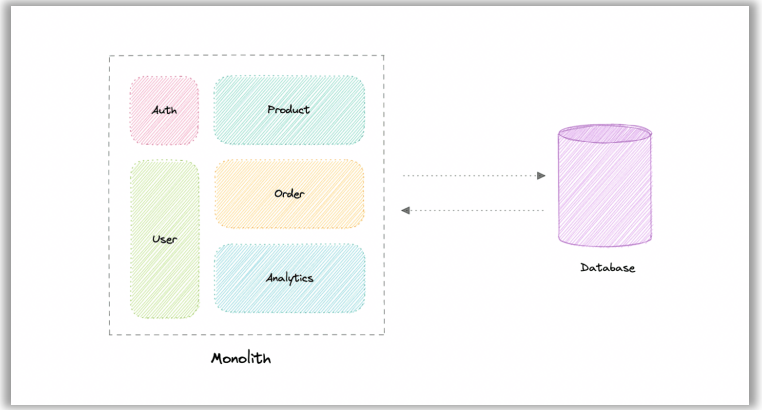

*Fuente: IBM.com*

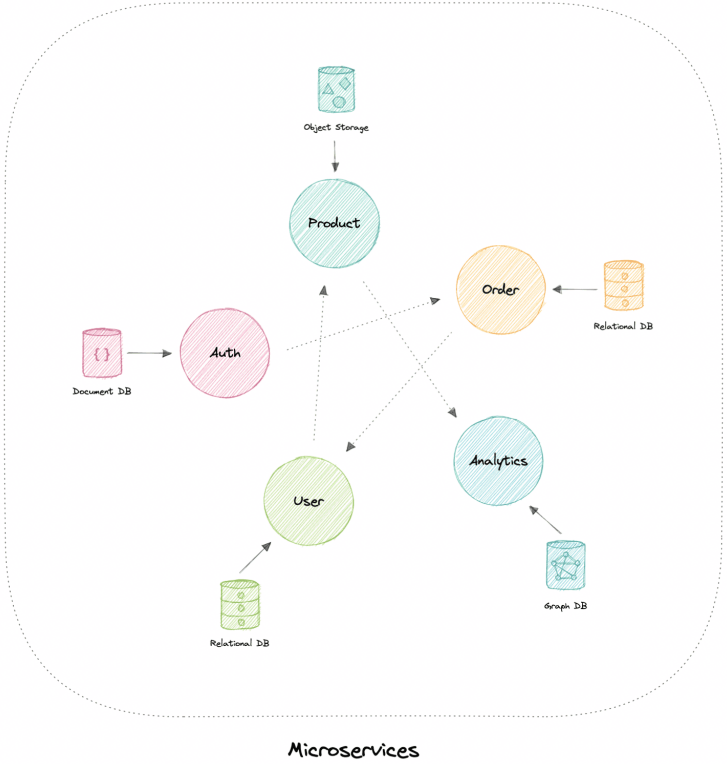
# Microservicios

- Arquitectura que consiste en una colección de pequeños servicios autónomos, donde cada servicio es auto-contenido y se refiere a una operación de negocio única dentro de un contexto limitado.
  - Puede desplegarse independientemente del resto
  - Son débilmente acoplados
  - Se focalizan en una única tarea de forma eficaz
  - Son independientes del lenguaje de programación y cuentan con su propia base de código (*codebase*)
  - Requieren un contexto muy limitado

# Microservicios

Un **microservicio** es una unidad de descomposición funcional de un sistema en un servicio gestionable e independientemente desplegable

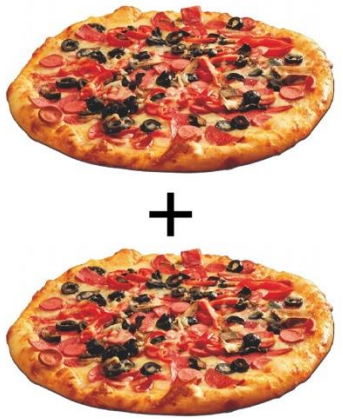



Fuente: “System Design: Monoliths and Microservices”, K. Pratap.

Un microservicio es una **unidad de trabajo**

# Microservicios__
## Características

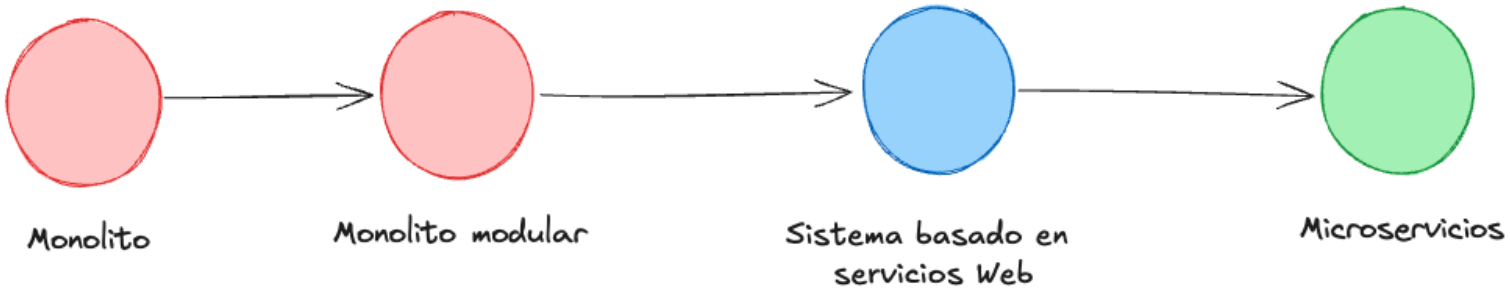


- **Pequeños** y focalizados
  - Regla de las 2 pizzas
  - Tratados como una aplicación o producto independiente
  - La granularidad depende de las necesidades del negocio
- Despliegue "***zero coordination***"
- Un MS **no conoce la implementación** del monolito base ni la de otros MS vecinos
- Se **pueden ejecutar múltiples copias** de un MS en distintas máquinas
- El **despliegue debe externalizarse**

# Microservicios__
## ¿Cuándo utilizarlos?

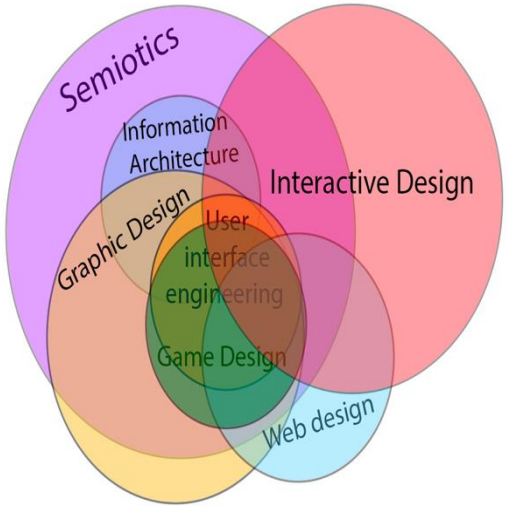
NO es el estilo adecuado para comenzar una aplicación

- Solución a problemas de escalabilidad, no para nuevos diseños




Evolución habitual de una aplicación compleja en la web

UNIVERSIDAD DE CÓRDOBA



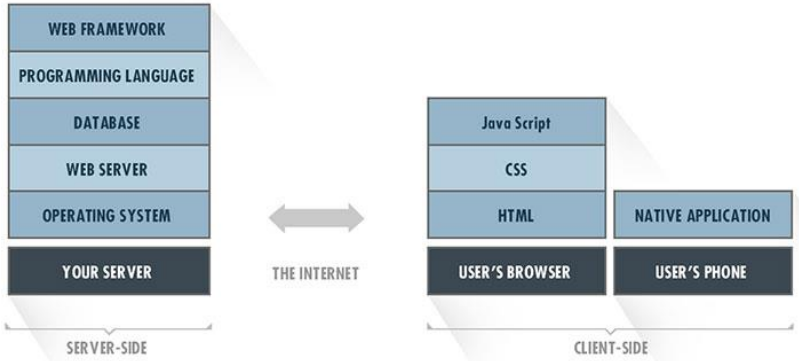

PROGRAMACIÓN WEB – S3.4

# Pila de tecnología

Dr. José Raúl Romero Salguero
jrromero@uco.es

# Pila de tecnología

- En desarrollo web, el término "pila de tecnología" (***tech stack***) se refiere al conjunto de herramientas, lenguajes y software que se emplea durante las fases de desarrollo y despliegue de un producto web o aplicación móvil
- En general, la pila se compone de la combinación de tecnologías utilizadas en el lado del servidor (*back-end*) y en el lado del cliente (*front-end*)
- *Ejemplo de pila*: LAMP (Linux + Apache + MySQL + PHP)




*Fuente: Silicon Valley Software Group*

# Pila de tecnología__
# *Back-end* Vs. *Front-end*

El ***back-end*** contiene la lógica de negocio de la aplicación

- El *back-end* nunca es accedido por el usuario directamente, sino a través del *front-end*
- El *back-end* se desarrolla con lenguajes de programación, según las necesidades del proyecto (Java, Ruby, PHP, etc.)

El ***front-end*** es la parte visual con la interactúa el usuario final

- El *front-end* se desarrolla predominantemente con lenguajes de marcado (HTML, CSS) y el lenguaje de programación Javascript

# Lenguajes habituales en Back-end (2025)

**Back-end (Server-side) table in most popular websites**

| Websites | C# | C | C++ | D | Elixir | Erlang | Go | Hack | Haskell | Java | JavaScript | Perl | PHP | Python | Ruby |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Google | No | Yes | Yes | No | No | No | Yes | No | No | Yes | Yes | No | No | Yes | No |
| Facebook | No | No | Yes | Yes | No | Yes | No | Yes | Yes | Yes | No | No | Yes | Yes | No |
| YouTube | No | Yes | Yes | No | No | No | Yes | No | No | Yes | No | No | No | Yes | No |
| Yahoo | No | No | No | No | No | No | No | No | No | No | No | No | Yes | No | No |
| Etsy | No | No | No | No | No | No | No | No | No | No | No | No | Yes | No | No |
| Amazon | No | No | Yes | No | No | No | No | No | No | Yes | No | Yes | No | No | No |
| Wikipedia | No | No | No | No | No | No | No | No | No | No | No | No | Yes | No | No |
| Fandom | No | No | No | No | No | No | No | No | No | No | No | No | Yes | No | No |
| X | No | No | Yes | No | No | No | No | No | No | Yes | No | No | No | No | Yes |
| Bing | Yes | No | Yes | No | No | No | No | No | No | No | No | No | No | No | No |
| eBay | No | No | No | No | No | No | No | No | No | Yes | Yes | No | No | No | No |
| MSN | Yes | No | No | No | No | No | No | No | No | No | No | No | No | No | No |
| LinkedIn | No | No | No | No | No | No | No | No | No | Yes | Yes | No | No | No | No |
| Pinterest | No | No | No | No | Yes | Yes | No | No | No | No | No | No | No | Yes | No |
| WordPress.com | No | No | No | No | No | No | No | No | No | No | No | No | Yes | No | No |
| Netflix | No | No | No | No | No | No | No | No | No | Yes | No | No | No | Yes | No |

Fuente: https://en.wikipedia.org/wiki/Programming_languages_used_in_most_popular_websites

# *Tech stacks* clásicas (muy extendidas)

**LAMP**: Linux + Apache + MySQL + PHP

- Muy extendida en gestores de contenidos (CMS) como WordPress, Drupal, Joomla

**MEAN**: MongoDB + Express.js + Angular + Node.js

- Primera pila full-stack “todo JS” popular

**.NET Stack**: ASP.NET + C# + SQL Server

**Java Stack**: Java, Spring Boot, PostgreSQL, Linux, NGINX

- Utilizadas en entornos corporativos y aplicaciones empresariales

# *Tech stacks* modernas

**MERN**: MongoDB + Express.js + React + Node.js

- Una de las más usadas en start-ups y SaaS.

**JAMstack**: JavaScript + APIs + Markup

- SSG (Next.js, Gatsby, Nuxt, Astro) + APIs (*headless* CMS, *serverless*).

**Next.js + Node.js + PostgreSQL**

- Muy común en proyectos React con SSR/SSG/ISR.

**Nuxt (Vue) + Node.js + PostgreSQL/MongoDB**

- Alternativa moderna para quienes usan Vue.

**SvelteKit + Node.js + PostgreSQL**

- Emergente por eficiencia y bundles muy pequeños.

# Recursos y lecturas

- Renderización en la web

  https://web.dev/articles/rendering-on-the-web
- "*The acronyms of rendering on the web*"

  https://dev.to/whitep4nth3r/the-acronyms-of-rendering-on-the-web-2g8h
- Renderización de HTML e interactividad del cliente

  https://web.dev/articles/client-side-rendering-of-html-and-interactivity
- " *Server-side rendering (SSR)*"

  https://nextjs.org/docs/pages/building-your-application/rendering/server-side-rendering
- "*How to implement Incremental Static Regeneration (ISR)*"

  https://nextjs.org/docs/pages/guides/incremental-static-regeneration
- "*3 ways of rendering the web*"

  https://dev.to/lobunto/3-ways-of-rendering-on-the-web-5b6h

# Recursos y lecturas

- Estilo de arquitectura de microservicios

  https://learn.microsoft.com/es-es/azure/architecture/guide/architecture-styles/microservices

- *Decomposing monoliths into microservices*

  https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-decomposing-monoliths/welcome.html

- "*Monolith First*" (M. Fowler)

  https://martinfowler.com/bliki/MonolithFirst.html

- "*Microservices*" (M. Fowler)

  https://nextjs.org/docs/pages/building-your-application/rendering/server-side-rendering