Desarrollo web en entorno servidor - DWES

TEMA 1 EJERCICIOS

con glosario de términos

ENRIQUE NIETO LORENZO
6-10-2025

Preguntas Tema 1

1. Protocolos de comunicaciones: IP, TCP, HTTP, HTTPS.

Los 4 pilares de la comunicación web: de la dirección a la entrega segura de datos. Los protocolos IP, TCP, HTTP y HTTPS forman una jerarquía de reglas que permite a los dispositivos localizarse, establecer conexiones fiables, intercambiar información y, finalmente, protegerla, constituyendo la base sobre la que se construye toda la World Wide Web.

¿Qué es?

Los protocolos de comunicaciones son un conjunto de reglas y estándares que definen cómo se comunican los dispositivos en una red. No son un bloque monolítico, sino que trabajan en capas, donde cada una se ocupa de una tarea específica y se apoya en la capa inferior.

- **IP (Internet Protocol):** Es el protocolo de la capa de red. Su función principal es el direccionamiento y el enrutamiento. Asigna una dirección única (dirección IP) a cada dispositivo conectado a una red y se encarga de dirigir los paquetes de datos (datagramas) desde un origen a un destino a través de las distintas redes que componen Internet. Es un protocolo "no orientado a conexión" y "no fiable", lo que significa que no garantiza que los paquetes lleguen, ni que lo hagan en orden. Es como el sistema de correos: sabe las direcciones, pero no garantiza la entrega certificada.
- **TCP (Transmission Control Protocol):** Es el protocolo de la capa de transporte que trabaja directamente sobre IP. Su misión es solucionar las carencias de IP, proporcionando una comunicación **fiable y ordenada**. Antes de enviar datos, establece una conexión formal entre el emisor y el receptor (mediante el *three-way handshake*). Luego, divide los datos en segmentos, los numera, y se asegura de que lleguen todos y en el orden correcto, solicitando la retransmisión de los que se pierdan. Es el "servicio de mensajería certificada" que verifica la entrega.
- **HTTP (Hypertext Transfer Protocol):** Es un protocolo de la capa de aplicación, el lenguaje que hablan los navegadores web y los servidores web. Funciona sobre TCP/IP y define un conjunto de métodos de petición (como GET para solicitar datos, POST para enviar datos, PUT para actualizar, DELETE para borrar) y códigos de respuesta (como 200 OK, 404 Not Found,

500 Internal Server Error). Es un protocolo "sin estado" (*stateless*), lo que significa que cada petición es independiente y el servidor no guarda información sobre peticiones anteriores del mismo cliente.

- **HTTPS (Hypertext Transfer Protocol Secure):** No es un protocolo nuevo en sí mismo, sino la combinación del protocolo HTTP con una capa de seguridad adicional: **SSL/TLS (Secure Sockets Layer/Transport Layer Security)**. Esta capa, que se sitúa entre HTTP y TCP, se encarga de dos cosas cruciales:
    1. **Cifrar la comunicación:** Todo el intercambio de datos entre el cliente y el servidor se vuelve ilegible para cualquiera que intercepte el tráfico.
    2. **Autenticar al servidor:** El cliente puede verificar, a través de un certificado digital, que se está conectando al servidor legítimo y no a un impostor.

¿Para qué sirve?

En el desarrollo de aplicaciones web, esta pila de protocolos es el cimiento sobre el que trabajamos cada día:

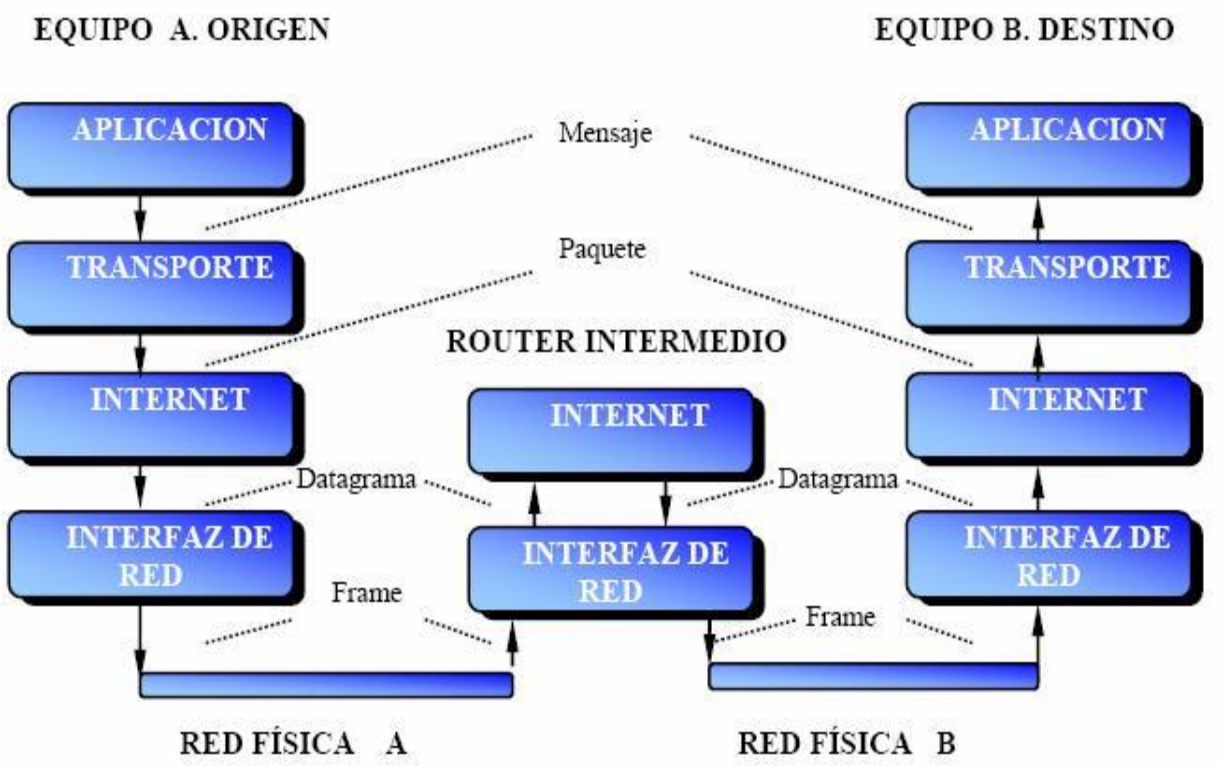
- **IP y TCP** son la infraestructura invisible pero esencial. Sin ellos, no habría una red global fiable sobre la cual desplegar nuestras aplicaciones. Garantizan que el HTML, CSS, JavaScript y los datos de nuestras APIs lleguen intactos desde el servidor al navegador del usuario.
- **HTTP** es nuestro lenguaje de trabajo en el backend. Cuando creamos una API REST, estamos definiendo *endpoints* (URLs) que responden a métodos HTTP (GET /usuarios, POST /usuarios). Entender HTTP es fundamental para diseñar servicios web eficientes, escalables y semánticamente correctos.
- **HTTPS** es el estándar de facto por seguridad y confianza. Cualquier aplicación que maneje datos sensibles (contraseñas, datos personales, tarjetas de crédito) **debe** usar HTTPS para proteger al usuario de ataques de intermediario (*Man-in-the-Middle*). Además, los navegadores modernos marcan los sitios HTTP como "No seguros" y los motores de búsqueda como Google penalizan su posicionamiento.

Relación con el módulo

- En **Entorno Servidor**, al crear un backend (con Node.js, PHP, Java, etc.), estamos implementando la lógica que gestiona las peticiones HTTP. Definimos rutas, procesamos los datos que llegan en un POST, decides qué código de estado devolver y qué cabeceras HTTP enviar. Entender la naturaleza sin estado de HTTP es clave para implementar mecanismos de sesión y autenticación (como cookies o tokens JWT).
- En **Despliegue**, te enfrentas a la configuración del servidor web (Apache, Nginx). Tendrás que configurar el servidor para que escuche en los puertos 80 (HTTP) y 443 (HTTPS), gestionar redirecciones de HTTP a HTTPS, y lo más importante, instalar y renovar certificados SSL/TLS para habilitar HTTPS. También configurarás reglas de firewall que operan a nivel de TCP/IP.

Ejemplos

1. Un usuario escribe https://google.com en su navegador.
2. El DNS traduce google.com a una dirección **IP** (ej. 142.250.200.110).
3. El navegador inicia una conexión **TCP** con esa IP en el puerto 443 (puerto por defecto para HTTPS). Se produce el *three-way handshake* (SYN, SYN-ACK, ACK) para establecer la conexión.
4. Se negocia una conexión segura **SSL/TLS** (el "S" de HTTPS). El servidor presenta su certificado, se intercambian claves y se establece un canal cifrado.
5. El navegador envía una petición **HTTP** GET / a través de ese canal seguro.
6. El servidor de Google responde con un código de estado HTTP 200 OK y el contenido HTML de la página, todo cifrado.



Recurso externo

¿Cómo funciona Internet? - Una explicación visual de los protocolos.

- https://www.youtube.com/watch?v=AYdF7b3nMto

2. Modelo de comunicaciones cliente – servidor y su relación con las aplicaciones web.

El diálogo que construye la web: el modelo cliente-servidor como pilar de las aplicaciones modernas. Esta arquitectura distribuye las tareas entre un "cliente" que solicita servicios y un "servidor" que los provee, permitiendo una separación de responsabilidades que es fundamental para el desarrollo, la escalabilidad y la seguridad de cualquier aplicación web.

¿Qué es?

El modelo cliente-servidor es un **paradigma de arquitectura de software** en el que las tareas y las cargas de trabajo se reparten entre dos tipos de participantes independientes pero interconectados: los **clientes** y los **servidores**. La comunicación se produce a través de una red, siguiendo un patrón de petición-respuesta.

- **El Cliente:** Es el actor que **inicia la comunicación**. Su principal función es solicitar recursos o servicios al servidor. Generalmente, es la parte con la que el usuario interactúa directamente. Un navegador web, una aplicación móvil, o incluso otro servicio backend, pueden actuar como clientes. El cliente se encarga de la capa de presentación (la interfaz de usuario) y de construir las peticiones adecuadas.
- **El Servidor:** Es el actor que **espera y responde a las peticiones** de los clientes. Su función es gestionar, procesar y proveer los recursos o servicios solicitados. Alberga la lógica de negocio, el acceso a las bases de datos y los recursos (imágenes, archivos, etc.). Es un sistema potente, siempre activo (*always on*), esperando conexiones entrantes en puertos de red específicos (ej. puerto 80 para HTTP, 443 para HTTPS).

La clave de este modelo es la **distinción clara de roles**. El servidor no inicia el contacto; simplemente escucha y sirve. El cliente no almacena la lógica de negocio principal; la consume. Esta separación es lo que lo diferencia de otras arquitecturas como la *peer-to-peer* (P2P), donde cada nodo puede actuar como cliente y servidor simultáneamente.

¿Para qué sirve?

Este modelo es la columna vertebral de la gran mayoría de las aplicaciones en red. Sus ventajas son la razón de su éxito:

1. **Centralización de la gestión:** La lógica de negocio y los datos residen en el servidor. Esto facilita enormemente las actualizaciones, el mantenimiento y el control de la seguridad. Si se necesita cambiar una regla de negocio, solo se modifica el código del servidor, y todos los clientes se benefician del cambio al instante.
2. **Separación de responsabilidades (*Separation of Concerns*):** Permite que los equipos de desarrollo se especialicen. El equipo de *frontend* (cliente) se

enfoca en la experiencia de usuario y la interfaz, mientras que el equipo de *backend* (servidor) se enfoca en la eficiencia, la seguridad y la gestión de datos.

3. **Escalabilidad:** Se puede escalar el servidor de forma independiente a los clientes. Si una aplicación web se vuelve muy popular, se pueden añadir más recursos al servidor (más CPU, más RAM, o incluso más máquinas detrás de un balanceador de carga) sin tener que modificar los millones de clientes (navegadores) que la consumen.
4. **Independencia de plataforma:** Un mismo servidor (backend) puede dar servicio a múltiples tipos de clientes: una aplicación web para navegadores, una aplicación nativa para iOS, otra para Android y una aplicación de escritorio. Todos ellos consumen la misma API, lo que reduce la duplicidad de la lógica de negocio.

Relación con el módulo

- **"Desarrollo Web en Entorno Cliente" (DWEC):** Este módulo se centra al 100% en la construcción del **cliente**. Cuando aprendes JavaScript, TypeScript y frameworks como Angular o Vue.js, estás aprendiendo a crear aplicaciones sofisticadas que se ejecutan en el navegador del usuario. El objetivo principal de este código es renderizar interfaces y, crucialmente, realizar peticiones HTTP (usando fetch o axios) al servidor para obtener o enviar datos.
- **"Desarrollo Web en Entorno Servidor" (DWES):** Este es el corazón del **servidor**. Con tecnologías como Node.js, PHP o Java, aprendes a crear la lógica que recibe esas peticiones HTTP. Aquí defines los *endpoints* de una API, validas los datos de entrada, te comunicas con la base de datos (módulo "Acceso a Datos") y formulas las respuestas que se enviarán de vuelta al cliente.
- **"Despliegue de Aplicaciones Web" (DAW):** Este módulo te enseña a poner el **servidor** en producción. Aprendes a configurar el servidor web (Nginx, Apache), la base de datos, el firewall y el sistema operativo para que tu aplicación backend esté disponible, segura y sea accesible en Internet para todos los clientes.

Ejemplos

1. **Cliente (Navegador):** El usuario introduce la URL https://miblog.com/articulo/123 y pulsa Enter. El navegador envía una petición GET a esa URL.
2. **Servidor (Backend con PHP/Laravel):** El servidor recibe la petición. El *router* de Laravel la dirige a un controlador específico.
3. **Lógica del Servidor:** El controlador pide al modelo que busque en la base de datos el artículo con ID 123.
4. **Respuesta del Servidor:** Una vez obtenido el artículo, el servidor renderiza una plantilla HTML, inyectando el título y el contenido del artículo en ella. Envía esta página HTML completa de vuelta al cliente.
5. **Cliente (Navegador):** Recibe el HTML y lo muestra al usuario.

Recurso externo

Client-Server Model Explained

https://www.geeksforgeeks.org/client-server-model/

3. Estudio sobre los métodos de petición HTTP /HTTPS más utilizados.

Los verbos de la web: cómo GET, POST, PUT y DELETE construyen las aplicaciones interactivas. Los métodos de petición HTTP son las acciones que un cliente solicita a un servidor sobre un recurso específico. Entender su propósito, semántica y diferencias es clave para diseñar y consumir APIs RESTful de manera correcta y eficiente.

¿Qué es?

Los métodos de petición HTTP (también conocidos como "verbos HTTP") son un conjunto de identificadores estandarizados que indican la acción deseada que debe ser realizada por el servidor sobre un recurso concreto. Un recurso es cualquier cosa que pueda ser identificada por una URI (Uniform Resource Identifier), como una página HTML, una imagen, un conjunto de datos sobre un usuario, etc.

Estos métodos son una parte fundamental de la línea de solicitud de un mensaje HTTP. Los más importantes y utilizados se asocian comúnmente con las operaciones CRUD (Create, Read, Update, Delete) de la gestión de datos:

- **GET:** Solicita una representación de un recurso específico. Es el método más común, usado por los navegadores para obtener páginas web, estilos CSS, imágenes y datos de APIs.
- **POST:** Envía una entidad a un recurso específico, causando a menudo un cambio en el estado o efectos secundarios en el servidor. Se usa típicamente para crear nuevos recursos (ej. registrar un nuevo usuario, publicar un comentario).
- **PUT:** Reemplaza **toda** la representación actual del recurso de destino con la carga útil de la petición. Se usa para actualizar un recurso existente en su totalidad.
- **DELETE:** Elimina un recurso específico.

Es crucial entender dos propiedades de estos métodos:

- **Seguridad (Safe Methods):** Un método es "seguro" si no modifica el estado del servidor. GET es un método seguro, ya que solo solicita datos. Hacer una petición GET no debería tener más efecto que la propia recuperación de la información.
- **Idempotencia (Idempotent Methods):** Un método es "idempotente" si realizar la misma petición múltiples veces produce el mismo resultado que realizarla una sola vez. GET, PUT y DELETE son idempotentes. Por ejemplo, borrar un usuario con DELETE /users/123 una vez tiene el mismo efecto en el estado del servidor que hacerlo diez veces (el usuario es borrado la primera vez, y las siguientes peticiones probablemente devuelvan un error 404, pero el estado final del sistema no cambia). POST **no es idempotente**: enviar el mismo formulario de registro dos veces creará dos usuarios diferentes.

¿Para qué sirve?

Utilizar el método HTTP correcto no es solo una convención, sino que aporta semántica y previsibilidad a las APIs. Permite a desarrolladores, frameworks y herramientas intermediarias (como cachés o proxies) entender la intención de una petición sin necesidad de analizar su contenido.

- **Diseño de APIs RESTful:** Son la base para crear APIs limpias y auto-descriptivas. Una API que usa GET /productos para obtener una lista, POST /productos para crear uno nuevo y DELETE /productos/45 para eliminarlo es intuitiva y sigue un estándar global.
- **Interacción con el Frontend:** El código del cliente (JavaScript) utiliza estos verbos para comunicarse con el backend. La lógica de la aplicación cambia drásticamente si se necesita obtener datos (GET) o enviar un nuevo formulario (POST).
- **Optimización y Caching:** Las peticiones GET, al ser seguras, pueden ser cacheadas por navegadores y CDNs (Content Delivery Networks), mejorando drásticamente el rendimiento de una aplicación. Las peticiones POST o PUT nunca deben ser cacheadas.

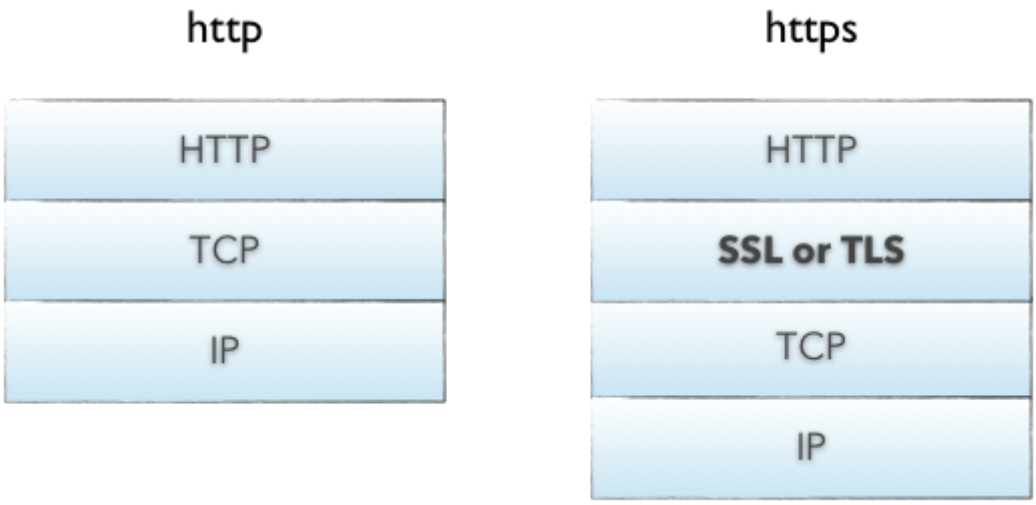
Relación con el módulo

Los métodos HTTP son el nexo de unión entre el cliente y el servidor, por lo que son fundamentales en ambos módulos de desarrollo:

- **"Desarrollo Web en Entorno Cliente" (DWEC):** Al usar la API fetch o librerías como axios en JavaScript, debes especificar explícitamente el método que quieres usar. Por ejemplo: fetch('/api/users', { method: 'POST', body: JSON.stringify(newUser) }). Entender qué método usar para cada operación es una habilidad básica del desarrollador frontend.
- **"Desarrollo Web en Entorno Servidor" (DWES):** En el backend, los frameworks modernos (Express, Laravel, Spring Boot) utilizan un sistema de *routing* que asocia una URL y un método HTTP a una función controladora específica.

Ejemplos

- **Un formulario de contacto.**
    1. **Cliente:** Cargas la página mipagina.com/contacto. El navegador realiza una petición **GET** para obtener el formulario HTML.
    2. **Servidor:** Responde con el HTML del formulario.
    3. **Cliente:** Rellenas el formulario y pulsas "Enviar". El navegador empaqueta los datos del formulario y realiza una petición **POST** a mipagina.com/enviar-mensaje.
    4. **Servidor:** Recibe los datos del **POST**, los procesa (ej. los guarda en la base de datos y envía un email) y responde con una página de "Gracias".



Recurso externo

HTTP request methods - MDN Web Docs

- https://developer.mozilla.org/es/docs/Web/HTTP/Methods

4. Estudio sobre el concepto de URI (Idendificador de Recursos Uniforme)/URL/URN, estructura, utilidad y relación con el protocolo HTTP/HTTPS.

El DNI, la dirección y el nombre de los recursos web: descifrando URI, URL y URN. Un URI es el identificador genérico de cualquier recurso, mientras que una URL especifica su ubicación (dónde encontrarlo) y una URN le da un nombre único (qué es), independientemente de dónde se encuentre. En la práctica diaria, usamos URLs para decirle al protocolo HTTP a qué recurso debe aplicar sus métodos.

¿Qué es?

Para entender la web, debemos saber cómo identificar y localizar sus componentes. Aquí es donde entran estos tres acrónimos, que forman una jerarquía conceptual.

- **URI (Uniform Resource Identifier - Identificador de Recursos Uniforme):** Es el concepto más amplio. Un URI es una cadena de caracteres que identifica de forma inequívoca un recurso, ya sea físico o abstracto. Es el "superconjunto" que engloba tanto a las URLs como a las URNs. Pensemos en un URI como el concepto general de "identificación de una persona".
- **URL (Uniform Resource Locator - Localizador de Recursos Uniforme):** Es el tipo de URI más común y conocido. Una URL no solo identifica un recurso, sino que, y esto es lo clave, también especifica **cómo y dónde localizarlo**. Proporciona el mecanismo de acceso (protocolo) y la ubicación en la red. Siguiendo la analogía, una URL sería la "dirección postal completa" de una persona: te dice exactamente dónde encontrarla. **Toda URL es un URI.**
- **URN (Uniform Resource Name - Nombre de Recurso Uniforme):** Es un tipo de URI que busca dar un **nombre único y persistente** a un recurso, independientemente de su ubicación o de cómo acceder a él. Una URN identifica al recurso por su "identidad", no por su "localización". En nuestra analogía, una URN sería el "número de DNI" de una persona: la identifica de forma única, pero no te dice dónde vive. Un ejemplo clásico es el ISBN de un libro (urn:isbn:0-486-27557-4). Aunque son importantes conceptualmente, las URNs son mucho menos comunes en el desarrollo web del día a día.

Estructura de una URL
La URL es la herramienta con la que trabajamos constantemente. Su estructura se descompone en varias partes:
https://www.ejemplo.com:443/productos/123?orden=popular#descripcion

- **Esquema (Scheme):** https:// - Define el protocolo a utilizar para acceder al recurso (HTTP, HTTPS, FTP, mailto, file, etc.).
- **Host (o Dominio):** www.ejemplo.com - Identifica el servidor que aloja el recurso.

- **Puerto:** :443 - El "canal" específico en el servidor al que se dirige la petición. A menudo se omite si se usa el puerto por defecto (80 para HTTP, 443 para HTTPS).
- **Ruta (Path):** /productos/123 - Especifica la ubicación del recurso dentro del servidor, a menudo siguiendo una estructura jerárquica.
- **Query String (o Parámetros):** ?orden=popular - Una serie de pares clave-valor que se usan para pasar datos adicionales al servidor, como filtros, paginación u opciones de ordenación. Comienza con ? y se separan con &.
- **Fragmento (o Ancla):** #descripcion - Es un identificador que apunta a una sección específica dentro del propio recurso. El fragmento es procesado exclusivamente por el cliente (el navegador) y no se envía al servidor.

¿Para qué sirve?

La URL es el pilar de la navegación y la interacción en la web. Sin ella, los protocolos como HTTP no sabrían a qué recurso aplicar sus métodos.

- **Navegación y Enlazado:** Es el mecanismo que permite a los usuarios saltar de una página a otra a través de hipervínculos (<a href="...">).
- **Identificación de Endpoints de API:** En el desarrollo de backend, la ruta de la URL es fundamental para definir los *endpoints* de una API REST. La URL api/users/1 identifica unívocamente al recurso "usuario con ID 1".
- **Paso de datos al servidor:** La Query String es una forma sencilla y estándar de que el cliente envíe datos al servidor en peticiones GET, permitiendo la creación de funcionalidades dinámicas como búsquedas (/buscar?q=desarrollo+web) o filtrados.
- **Control del cliente:** El fragmento permite a los desarrolladores crear enlaces que lleven al usuario a un punto exacto de una página larga, mejorando la experiencia de usuario.

Relación con el módulo

- **"Desarrollo Web en Entorno Servidor" (DWES):** La principal tarea de un framework de backend (como Express en Node.js o Laravel en PHP) es el **enrutamiento (routing)**. El framework analiza la ruta de la URL y los parámetros de la petición entrante para decidir qué bloque de código (controlador) debe ejecutarse. Diseñar una estructura de rutas lógica y semántica es una de las habilidades clave del desarrollador backend.
- **"Desarrollo Web en Entorno Cliente" (DWEC):** En el frontend, se trabaja constantemente con URLs. Desde crear enlaces, cargar recursos (imágenes, scripts), hasta construir las URLs correctas para realizar peticiones fetch a la API del backend. Además, en las Single Page Applications (SPAs), librerías como React Router o Vue Router manipulan la URL del navegador para simular la navegación entre páginas sin recargar el sitio.
- **"Despliegue de Aplicaciones Web" (DAW):** Al configurar un servidor web como Apache, se crean reglas basadas en la URL. Por ejemplo, se puede configurar un *reverse proxy* para que todas las peticiones que lleguen a /api/ se

redirijan internamente al servidor de la API, mientras que el resto se dirijan al servidor que sirve los archivos estáticos del frontend.

Ejemplos

- **Cargar una imagen en una página HTML.**
  El navegador lee la etiqueta <img src="/imagenes/logo.png">.
  1. El navegador construye la URL completa: https://www.miweb.com/imagenes/logo.png.
  2. Realiza una petición GET a esa URL.
  3. El servidor web busca el archivo logo.png en la carpeta /imagenes de su directorio raíz y lo devuelve.

Recurso externo

The Anatomy of a URL

- https://www.cloudflare.com/learning/dns/what-is-a-url/

5. Modelo de desarrollo de aplicaciones multicapa – comunicación entre capas – componentes – funcionalidad de cada capa.

El "Lego" del software moderno: cómo la arquitectura multicapa organiza el caos y potencia la escalabilidad. Este modelo de desarrollo descompone una aplicación en capas lógicas independientes, cada una con una responsabilidad específica (presentación, lógica y datos), permitiendo un desarrollo más ordenado, mantenible y escalable.

¿Qué es?

La arquitectura multicapa (o N-tier) es un **patrón de arquitectura de software** que separa una aplicación en distintas capas lógicas. Aunque el número de capas ("N") puede variar, el modelo más común y didáctico es la **arquitectura de 3 capas (3-tier)**, que divide la aplicación en:

1. **Capa de Presentación (UI - User Interface):** Es la capa más externa, con la que el usuario interactúa directamente. Su única responsabilidad es mostrar los datos al usuario y capturar sus entradas. No contiene lógica de negocio. En el desarrollo web, esta capa es el **frontend**: el HTML, CSS y JavaScript que se ejecuta en el navegador del cliente.
2. **Capa de Lógica de Negocio (o Capa de Aplicación):** Es el cerebro de la aplicación. Reside en el servidor y su función es procesar las peticiones del cliente, aplicar las reglas de negocio, tomar decisiones y coordinar la comunicación con la capa de datos. No se preocupa de cómo se muestran los datos (eso es trabajo de la presentación) ni de cómo se almacenan (eso es trabajo de los datos). Aquí es donde vive el **backend**.
3. **Capa de Datos (o Capa de Acceso a Datos):** Es la capa más interna y se encarga de la persistencia de la información. Su responsabilidad es almacenar, recuperar, actualizar y eliminar datos de una o más fuentes, como una base de datos (SQL o NoSQL), un sistema de archivos o servicios externos. Expone una interfaz sencilla para que la capa de lógica pueda solicitarle datos sin necesidad de conocer los detalles de su implementación.

La **comunicación entre capas** es fundamental: siempre es jerárquica y en una única dirección. La capa de presentación solo habla con la capa de lógica. La capa de lógica habla con la de presentación (para responderle) y con la de datos. La capa de presentación **nunca** se comunica directamente con la capa de datos. Esta estricta separación es el pilar del modelo.

¿Para qué sirve?

Adoptar una arquitectura multicapa proporciona beneficios inmensos, especialmente a medida que las aplicaciones crecen en complejidad:

- **Separación de Responsabilidades:** Cada capa tiene un trabajo claro y definido. Esto hace que el código sea más fácil de entender, depurar y mantener.
- **Desarrollo en Paralelo:** Permite que diferentes equipos trabajen simultáneamente. El equipo de frontend puede construir la interfaz de usuario mientras el equipo de backend desarrolla la lógica de negocio, siempre que acuerden la "interfaz" de comunicación entre ellos (la API).
- **Escalabilidad Independiente:** Cada capa puede ser escalada por separado según las necesidades. Si la base de datos es el cuello de botella, se pueden añadir más recursos a la capa de datos sin tocar las otras. Si hay muchas peticiones de usuarios, se pueden añadir más servidores de aplicación (capa de lógica).

- **Flexibilidad Tecnológica:** Se pueden usar diferentes tecnologías para cada capa. Puedes tener un frontend hecho con Vue.js, un backend con Java y una base de datos PostgreSQL. Mientras se respete el protocolo de comunicación (HTTP/JSON), las capas son intercambiables.
- **Reutilización:** La misma capa de lógica y datos (el backend) puede servir a múltiples capas de presentación: una aplicación web, una aplicación móvil nativa y una aplicación de escritorio, todas consumiendo la misma API.

Relación con el módulo

- **"Desarrollo Web en Entorno Cliente" (DWEC):** Se enfoca exclusivamente en la **Capa de Presentación**. Aquí aprendes a construir interfaces de usuario ricas e interactivas con JavaScript y frameworks (Angular, Vue, React) que se ejecutan en el navegador.
- **"Desarrollo Web en Entorno Servidor" (DWES):** Es el núcleo de la **Capa de Lógica de Negocio**. Aprendes a crear APIs, implementar las reglas de la aplicación, gestionar la autenticación y procesar los datos que vienen del cliente y van hacia la capa de datos.
- **"Despliegue de Aplicaciones Web" (DAW):** Te enseña a poner todas estas capas en producción, configurando los distintos servidores (servidor web para la presentación, servidor de aplicación para la lógica, servidor de base de datos para los datos) para que trabajen juntos.

Ejemplos

- **Un sistema de login.**
    1. **Capa de Presentación:** El usuario ve un formulario HTML (<form>) con campos para email y contraseña. Al pulsar "Entrar", el JavaScript del navegador envía los datos mediante una petición POST a /api/login.
    2. **Capa de Lógica:** El servidor recibe la petición. El controlador de login toma el email y la contraseña. Le pide a la capa de datos que busque un usuario con ese email.
    3. **Capa de Datos:** Ejecuta una consulta SQL: SELECT * FROM users WHERE email = ?. Devuelve el registro del usuario (si existe) a la capa de lógica.
    4. **Capa de Lógica:** Recibe los datos del usuario. Compara el hash de la contraseña almacenada con el hash de la contraseña enviada. Si coinciden, genera un token de sesión. Envía una respuesta 200 OK con el token al cliente.
    5. **Capa de Presentación:** Recibe la respuesta. Guarda el token en el almacenamiento local y redirige al usuario al panel de control.

Recurso externo

N-Tier Arquitectura explicada

- https://www.ibm.com/topics/n-tier-architecture

6. Modelo de división funcional front-end / back-end para aplicaciones web.

Las dos caras de la web: Front-end, la belleza visible, y Back-end, el motor invisible. Esta división funcional separa el desarrollo de una aplicación en dos grandes áreas: el Front-end, que se ocupa de todo lo que el usuario ve e interactúa en su navegador, y el Back-end, que gestiona la lógica, los datos y la infraestructura del servidor.

¿Qué es?

El modelo de división funcional Front-end / Back-end es una especialización de la arquitectura cliente-servidor y multicapa. Consiste en separar el proceso de desarrollo de una aplicación web en dos grandes dominios, cada uno con sus propias tecnologías, responsabilidades y perfiles profesionales.

- **Front-end (Lado del Cliente):** Se refiere a la parte de la aplicación que se ejecuta en el navegador del usuario. Es responsable de la **capa de presentación**. Su trabajo es renderizar la interfaz de usuario (UI), gestionar las interacciones del usuario (clics, formularios, etc.) y comunicarse con el Back-end para solicitar o enviar datos. No contiene lógica de negocio crítica.
  - **Tecnologías clave:** HTML (estructura), CSS (estilo) y JavaScript (interactividad).
  - **Frameworks y librerías:** React, Angular, Vue.js, Svelte.
  - **Preocupaciones:** Experiencia de Usuario (UX), diseño responsive, accesibilidad, rendimiento de la carga.
- **Back-end (Lado del Servidor):** Se refiere a la parte de la aplicación que se ejecuta en el servidor. Es el responsable de la **capa de lógica de negocio y la capa de datos**. Su trabajo es recibir peticiones del Front-end, procesarlas, interactuar con la base de datos, aplicar reglas de negocio, gestionar la autenticación y seguridad, y enviar una respuesta (generalmente en formato JSON) de vuelta al Front-end.
  - **Tecnologías clave:** Lenguajes como Node.js (JavaScript), PHP, Python, Java, C#, Go.
  - **Frameworks:** Express, Laravel, Django, Spring Boot.
  - **Bases de datos:** MySQL, PostgreSQL, MongoDB, Redis.
  - **Preocupaciones:** Seguridad, escalabilidad, rendimiento de la base de datos, integridad de los datos.

La comunicación entre ambos se realiza a través de una **API (Application Programming Interface)**, que actúa como un "contrato". El Back-end expone una serie de *endpoints* (URLs) y el Front-end los consume utilizando peticiones HTTP. Esta API es el pegamento que une las dos mitades.

¿Para qué sirve?

Esta separación no es arbitraria; es una respuesta a la creciente complejidad de las aplicaciones web y ofrece ventajas cruciales:

1. **Especialización y Eficiencia:** Permite a los desarrolladores especializarse. Un desarrollador Front-end se convierte en un experto en crear interfaces fluidas y atractivas, mientras que un desarrollador Back-end se enfoca en crear sistemas robustos, seguros y eficientes. Esto conduce a un producto final de mayor calidad.
2. **Desarrollo Desacoplado:** Mientras se respete el contrato de la API, el equipo de Front-end y el de Back-end pueden trabajar de forma independiente y en paralelo. El Front-end puede incluso trabajar con datos simulados (*mock data*) antes de que la API esté completamente terminada.
3. **Flexibilidad Tecnológica:** Se puede elegir la mejor tecnología para cada tarea. Un Front-end en React puede comunicarse sin problemas con un Back-end en Python/Django. Si en el futuro se decide rehacer el Front-end con una nueva tecnología (ej. pasar de Angular a Svelte), no es necesario tocar el Back-end.
4. **Reutilización del Back-end (Omnicanal):** Este es uno de los mayores beneficios. Un único Back-end bien diseñado puede servir datos no solo a una aplicación web (Front-end), sino también a una aplicación móvil nativa (iOS/Android), a una aplicación de escritorio o incluso a dispositivos IoT. La lógica de negocio se escribe una sola vez.

Relación con el módulo

Esta división es el **eje central que estructura todo el ciclo de DAW**. Los módulos están diseñados explícitamente para formar especialistas en estas áreas:

- **"Desarrollo Web en Entorno Cliente" (DWEC)** y **"Diseño de Interfaces Web" (DIW):** Estos módulos son 100% **Front-end**. En DIW aprendes los principios de usabilidad y diseño con HTML y CSS. En DWEC, te sumerges en JavaScript y frameworks para dar vida a esas interfaces, hacerlas dinámicas y comunicarlas con el Back-end.
- **"Desarrollo Web en Entorno Servidor" (DWES):** En este módulo aprendes a construir el servidor, la API y la lógica de negocio.
- **"Despliegue de Aplicaciones Web" (DAW):** Te enseña a desplegar ambas partes. Aprenderás a configurar un servidor web (como Apache) que sirva los archivos estáticos del Front-end y que, a la vez, redirija las peticiones a la API hacia el servidor de aplicación del Back-end.

Ejemplos

- **Ejemplo básico: Un conversor de divisas.**
    1. **Front-end:** Una página con dos campos de entrada (cantidad y divisa origen), un menú desplegable (divisa destino) y un botón "Convertir". Muestra el resultado en un párrafo.

2. **Interacción:** Cuando el usuario pulsa "Convertir", el JavaScript del Front-end recoge los valores y hace una petición GET a https://api.conversor.com/convert?from=EUR&to=USD&amount=100.
3. **Back-end:** El servidor recibe la petición. No almacena los tipos de cambio él mismo, sino que llama a otra API externa (ej. la del Banco Central Europeo) para obtener el tipo de cambio actual. Realiza el cálculo matemático (100 * tipo_de_cambio).
4. **Respuesta:** El Back-end responde con un JSON: {"result": 108.50}.
5. **Front-end:** Recibe el JSON y actualiza el párrafo del resultado para que muestre "108.50 USD".

Recurso externo

What is the Difference Between Frontend and Backend?

- https://aws.amazon.com/es/compare/the-difference-between-frontend-and-backend/

7. Página web estática – página web dinámica – aplicación web – mashup.

De la tarjeta de visita al software en la nube: la evolución de la web en cuatro conceptos. Las páginas web estáticas son como folletos digitales, las dinámicas personalizan su contenido, las aplicaciones web ofrecen herramientas interactivas, y los mashups crean nuevos servicios combinando APIs de terceros, mostrando una clara progresión en complejidad y funcionalidad.

¿Qué es?

Estos cuatro términos describen diferentes tipos de sitios y aplicaciones web, clasificados por su nivel de interactividad y cómo se genera su contenido. Representan una escala de complejidad creciente.

1. **Página Web Estática:** Es la forma más simple de contenido web. Consiste en archivos (HTML, CSS, JavaScript, imágenes) que se almacenan en el servidor y se entregan al navegador del cliente **exactamente como están guardados**. El contenido no cambia a menos que un desarrollador modifique manualmente los archivos fuente. No hay procesamiento en el servidor ni conexión a una base de datos para generar el contenido.
    - **Analogía:** Un folleto o una tarjeta de visita en formato digital.
2. **Página Web Dinámica:** A diferencia de la estática, el contenido de una página dinámica se genera "al vuelo" en el servidor en el momento de la solicitud. Cuando un usuario pide una URL, el servidor ejecuta un script (escrito en PHP, Node.js, Python, etc.) que puede consultar una base de datos, procesar datos

del usuario, comprobar la hora o realizar cualquier otra lógica para **ensamblar la página HTML final** antes de enviarla al cliente.
    - **Analogía:** Un periódico personalizado que se imprime justo para ti con las noticias del día.
3. **Aplicación Web (Web App):** Es un salto cualitativo desde una página dinámica. Una aplicación web no se centra solo en *mostrar* contenido, sino en *proporcionar una funcionalidad* compleja e interactiva al usuario. El usuario no es un mero consumidor, sino un creador o editor de datos. Técnicamente, las web apps suelen ser Single Page Applications (SPAs), donde una carga inicial de la página descarga una aplicación JavaScript robusta que se encarga de la interfaz y la comunicación con el servidor a través de APIs, sin necesidad de recargar la página entera.
    - **Analogía:** Un programa de software (como Word o Photoshop) que se ejecuta dentro del navegador.
4. **Mashup:** Es un tipo específico de aplicación web cuya característica principal es que **combina datos o funcionalidades de múltiples fuentes externas (APIs de terceros) para crear un nuevo servicio**. No genera todo su valor a partir de sus propios datos, sino integrando y presentando de forma novedosa los datos de otros.
    - **Analogía:** Un collage que crea una nueva imagen a partir de recortes de diferentes revistas.

¿Para qué sirve?

Cada tipo tiene su lugar y propósito en el ecosistema web:

- **Páginas Estáticas:** Ideales para sitios donde el contenido no cambia con frecuencia. Son extremadamente rápidas, seguras (menor superficie de ataque) y baratas de alojar. Perfectas para portfolios, páginas de aterrizaje (*landing pages*), sitios de documentación o webs corporativas sencillas.
- **Páginas Dinámicas:** Necesarias cuando el contenido debe ser personalizado o gestionado a gran escala. Son la base de blogs, foros, periódicos digitales y la mayoría de los sitios de e-commerce (donde se muestran productos desde una base de datos).
- **Aplicaciones Web:** Se utilizan para crear software como servicio (SaaS). Permiten a los usuarios realizar tareas complejas: gestionar proyectos (Trello), editar documentos (Google Docs), comunicarse (Gmail), o diseñar (Figma).
- **Mashups:** Permiten crear aplicaciones potentes rápidamente sin tener que construir toda la infraestructura desde cero. Son comunes en portales inmobiliarios (que muestran propiedades sobre un mapa de Google), agregadores de viajes (que combinan vuelos de múltiples aerolíneas) o dashboards de analítica.

Relación con el módulo

- **Página Estática:** Es el punto de partida. Lo que aprendes en **"Diseño de Interfaces Web" (DIW)** (HTML y CSS) y en los inicios de **"Desarrollo Web en**

Entorno Cliente" (DWEC) (JavaScript básico) te permite construir sitios estáticos.

- **Página Dinámica:** Es el modelo clásico que se estudia en **"Desarrollo Web en Entorno Servidor" (DWES)**. Aprender a usar un lenguaje como PHP o Node.js junto con una base de datos para generar HTML en el servidor es la esencia de este paradigma.
- **Aplicación Web:** Este es el enfoque moderno que domina el 2º año de DAW. Requiere una fuerte especialización en **DWEC** con frameworks como Angular o Vue.js para construir el front-end, y en **DWES** para crear una API RESTful que solo sirva datos (generalmente en JSON), separando completamente el front-end del back-end.
- **Mashup:** Es una aplicación práctica de las habilidades de una Web App. Tanto en **DWEC** (usando fetch para llamar a APIs de terceros desde el cliente) como en **DWES** (llamando a otras APIs desde el servidor) aprenderás a consumir servicios externos, que es la habilidad clave para construir un mashup.

Ejemplos

- **Página Estática:**
  - Una página personal de un desarrollador con su CV y enlaces a sus proyectos, escrita a mano en un único archivo HTML y CSS.
- **Página Dinámica:**
  - Una página de "Bienvenida" en PHP que muestra la fecha y hora actual del servidor cada vez que se recarga.
- **Aplicación Web:**
  - Una lista de tareas (To-Do list) donde puedes añadir, borrar y marcar tareas como completadas sin que la página se recargue. Cada acción es una llamada a una API en segundo plano.
- **Mashup:**
  - Una web que te pide tu código postal, utiliza una API gratuita para convertirlo en coordenadas (latitud/longitud) y luego utiliza la API de Google Maps para mostrarte un mapa de esa ubicación.

Recurso externo

Static vs. Dynamic Websites: What's the Difference?

- https://www.godaddy.com/resources/latam/stories/que-es-pagina-dinamica
-

8. Componentes de una aplicación web.

Los 4 fantásticos de la web: los componentes esenciales que dan vida a cualquier aplicación. Una aplicación web moderna se construye sobre cuatro pilares: el Front-end (la cara visible), el Servidor Web (el portero), el Back-end (el cerebro) y la

Base de Datos (la memoria). Entender el rol y la interacción de cada uno es fundamental para construir sistemas robustos y escalables.

¿Qué es?

Los componentes de una aplicación web son los distintos bloques de software, lógicamente separados, que trabajan en conjunto para entregar una funcionalidad completa al usuario. Aunque los detalles pueden variar, casi toda aplicación web moderna se compone de, al menos, estos cuatro elementos principales:

1. **Aplicación Front-end (Cliente):** Es el código que se descarga y ejecuta en el navegador del usuario (Chrome, Firefox, etc.). Este componente es responsable de todo lo que el usuario ve y con lo que interactúa: la interfaz gráfica, los botones, los formularios y las animaciones. Está construido principalmente con HTML, CSS y JavaScript.
2. **Servidor Web:** Es un software que se ejecuta en el servidor y actúa como el **punto de entrada** para todas las peticiones del cliente. Su trabajo principal es escuchar las peticiones HTTP/HTTPS que llegan desde Internet y decidir cómo gestionarlas. No es el "cerebro" de la aplicación, sino más bien el "recepcionista" o el "controlador de tráfico". Ejemplos comunes son **Nginx** y **Apache**.
3. **Aplicación Back-end (Servidor):** Este es el **núcleo lógico** de la aplicación. Es un programa (escrito en Node.js, PHP, Python, Java, etc.) que también se ejecuta en el servidor. A diferencia del Servidor Web, su trabajo no es gestionar conexiones, sino **implementar la lógica de negocio**. Recibe las peticiones que le reenvía el Servidor Web, procesa datos, aplica reglas, se comunica con la base de datos y genera una respuesta.
4. **Base de Datos (BBDD):** Es el sistema encargado de **almacenar y gestionar los datos de forma persistente**. La aplicación Back-end se comunica con la base de datos para guardar información nueva (un nuevo usuario, un producto), leer información existente (los datos de un perfil), actualizarla (cambiar una contraseña) o borrarla. Puede ser una base de datos relacional (SQL) como **MySQL** o **PostgreSQL**, o una no relacional (NoSQL) como **MongoDB**.

Es crucial entender la diferencia entre el **Servidor Web** y la **Aplicación Back-end**: el primero gestiona el tráfico y sirve archivos estáticos, mientras que el segundo ejecuta el código que da sentido a la aplicación. A menudo, el Servidor Web actúa como un *reverse proxy*, reenviando las peticiones dinámicas a la aplicación Back-end.

¿Para qué sirve?

Esta separación en componentes es la base de la arquitectura multicapa y del modelo Front-end/Back-end, y sus beneficios son enormes:

- **Organización y Mantenimiento:** Cada componente tiene una responsabilidad clara. Si hay un problema visual, se sabe que está en el Front-end. Si los datos

se guardan incorrectamente, el problema está en el Back-end o la BBDD. Esto simplifica la depuración y el mantenimiento.
- **Seguridad:** El Servidor Web puede actuar como una primera línea de defensa, filtrando peticiones maliciosas, gestionando los certificados SSL/TLS y ocultando la aplicación Back-end de la exposición directa a Internet.
- **Rendimiento y Escalabilidad:** El Servidor Web está altamente optimizado para manejar miles de conexiones simultáneas y servir archivos estáticos (imágenes, CSS, JS) muy rápidamente, liberando a la aplicación Back-end para que se concentre en la lógica de negocio, que es más costosa computacionalmente. Además, cada componente puede escalarse de forma independiente.

Relación con el módulo

El plan de estudios de DAW está diseñado para que te especialices en cada uno de estos componentes:

- **Front-end:** Los módulos de **"Diseño de Interfaces Web" (DIW)** y **"Desarrollo Web en Entorno Cliente" (DWEC)** se centran por completo en este componente.
- **Back-end:** El módulo de **"Desarrollo Web en Entorno Servidor" (DWES)** está dedicado a aprender a construir la lógica de la aplicación.
- **Servidor Web:** El módulo de **"Despliegue de Aplicaciones Web" (DAW)** es donde aprendes a instalar, configurar y gestionar este componente (Nginx, Apache) para poner en producción y conectar todas las demás piezas.

Ejemplos

- **Un blog en WordPress.**
    1. **Front-end:** El navegador del usuario solicita ver un artículo.
    2. **Servidor Web (Apache):** Recibe la petición GET /mi-articulo. Como la URL no es un archivo estático, Apache sabe que debe pasar la petición al motor de PHP.
    3. **Aplicación Back-end (WordPress/PHP):** El script de PHP analiza la URL, entiende que se está pidiendo "mi-articulo".
    4. **Base de Datos (MySQL):** El script PHP ejecuta una consulta SQL a la base de datos para obtener el contenido, autor y comentarios de "mi-articulo".
    5. **Back-end -> Servidor Web -> Front-end:** PHP usa los datos para generar una página HTML completa y se la devuelve a Apache. Apache la envía de vuelta al navegador del usuario para que la muestre.

Recurso externo

Cómo funciona un servidor web

- https://www.hostinger.com/es/tutoriales/que-es-un-servidor-web

9. Programas ejecutados en el lado del cliente y programas ejecutados en el lado del servidor - lenguajes de programación utilizados en cada caso.

El navegador y el servidor, los dos escenarios del código web. El código cliente se ejecuta en el navegador del usuario para crear la experiencia interactiva que se ve y se toca, mientras que el código servidor trabaja en la sombra para gestionar los datos, la seguridad y la lógica de negocio que da poder a la aplicación.

¿Qué es?

La distinción entre programas ejecutados en el lado del cliente y en el lado del servidor se refiere al **lugar físico y lógico donde se procesa el código** de una aplicación web.

- **Programas del Lado del Cliente (Client-Side):**
  Este código es descargado desde el servidor y se ejecuta íntegramente en la máquina del usuario, dentro de su navegador web (Google Chrome, Mozilla Firefox, etc.). El navegador actúa como un entorno de ejecución (una especie de "máquina virtual"). Este código tiene acceso directo al **DOM (Document Object Model)**, que es la representación en memoria de la página HTML, permitiéndole modificar la estructura, el estilo y el contenido de la página de forma dinámica.
    - **Lenguajes de programación:** El rey indiscutible es **JavaScript**. Hoy en día, se utiliza a menudo a través de **TypeScript** (un superconjunto de JavaScript que añade tipos estáticos) que se compila a JavaScript antes de ser enviado al navegador. **HTML** y **CSS**, aunque no son lenguajes de programación como tal, son las tecnologías que el código cliente manipula.
- **Programas del Lado del Servidor (Server-Side):**
  Este código reside y se ejecuta exclusivamente en el servidor web, una máquina remota controlada por los desarrolladores. El cliente (navegador) nunca ve este código; solo recibe el resultado de su ejecución (normalmente una página HTML, datos en formato JSON, una imagen, etc.). Este código tiene acceso a los recursos del servidor: el sistema de archivos, las bases de datos, otros servicios de red y puede ejecutar tareas computacionalmente intensivas.
    - **Lenguajes de programación:** El ecosistema es muy diverso. Los más populares en el contexto de DAW son:
      - **Node.js:** Permite ejecutar JavaScript/TypeScript en el servidor.
      - **PHP:** Uno de los lenguajes más veteranos y extendidos para el desarrollo web (la base de WordPress, Laravel).
      - **Python:** Muy popular por su simplicidad y potencia, con frameworks como Django y Flask.
      - **Java:** Robusto y escalable, usado en grandes aplicaciones empresariales con frameworks como Spring Boot.

    - Otros como **C# (.NET)**, **Go** o **Ruby** también son comunes.

¿Para qué sirve?

Cada lado tiene responsabilidades y capacidades distintas, y la magia de las aplicaciones web modernas reside en su colaboración:

- **El código cliente se especializa en:**
    - **Interactividad y Experiencia de Usuario (UX):** Crear interfaces ricas, animaciones, menús desplegables y responder a las acciones del usuario (clics, scroll, teclado) de forma instantánea, sin necesidad de recargar la página.
    - **Validación de entradas:** Comprobar que un formulario está bien rellenado (ej. que un email tiene formato de email) *antes* de enviarlo al servidor, proporcionando feedback inmediato al usuario.
    - **Manipulación del DOM:** Cambiar el contenido de la página dinámicamente basándose en las acciones del usuario o en los datos recibidos de una API.
    - **Gestionar el estado de la aplicación:** En las SPAs, es el cliente quien sabe qué "página" se está mostrando, si un menú está abierto, etc.
- **El código servidor se especializa en:**
    - **Seguridad y Autenticación:** Es el único lugar seguro para verificar contraseñas, gestionar sesiones de usuario y proteger los datos. **Nunca se debe confiar en el cliente para la seguridad.**
    - **Persistencia de Datos:** Es el único que puede conectarse de forma segura a la base de datos para leer, escribir, actualizar o borrar información.
    - **Lógica de Negocio Compleja:** Ejecutar las reglas clave de la aplicación (ej. calcular el precio final de un carrito de la compra con descuentos e impuestos, procesar un pago).
    - **Acceso a Servicios de Terceros:** Conectarse a otras APIs (ej. una pasarela de pago, un servicio de envío de emails) de forma segura, sin exponer claves secretas al cliente.

Relación con el módulo

- **"Desarrollo Web en Entorno Cliente" (DWEC):** Este módulo se centra al 100% en la programación **client-side**. Aquí es donde se aprende a dominar JavaScript/TypeScript, a manipular el DOM y a utilizar frameworks como Angular o Vue.js para construir las interfaces de usuario interactivas.
- **"Desarrollo Web en Entorno Servidor" (DWES):** Este módulo se dedica por completo a la programación **server-side**. Aquí se aprende a usar Node.js, PHP o Java para crear APIs, implementar la lógica de negocio, interactuar con el sistema de archivos y gestionar la comunicación con el cliente y la base de datos.

Ejemplos

- **Un formulario de registro.**
    1. **Lado del Cliente (JavaScript):** El usuario escribe su contraseña. El código JavaScript comprueba en tiempo real si la contraseña tiene más de 8 caracteres y muestra un mensaje "Contraseña segura" o "Contraseña demasiado corta" al lado del campo, sin recargar la página.
    2. **Lado del Servidor (PHP):** El usuario envía el formulario. El servidor recibe los datos. **Vuelve a validar** la longitud de la contraseña (principio de "nunca confíes en el cliente"). Procede a "hashear" la contraseña (un proceso criptográfico) y la guarda de forma segura en la base de datos. El código para hashear y guardar en la BBDD **solo puede existir en el servidor**.

Recurso externo

Lado cliente vs lado servidor

- https://www.cloudflare.com/es-es/learning/serverless/glossary/client-side-vs-server-side/

10. Lenguajes de programación utilizados en el lado servidor de una aplicación web (características y grado de implantación actual).

Los motores de la web: PHP domina el legado de la web, mientras JavaScript, Python y Java impulsan las nuevas aplicaciones. Los lenguajes del lado del servidor son la columna vertebral de la web dinámica. Las estadísticas de W3Techs confirman el liderazgo masivo de PHP (73.3%), pero el desarrollo de APIs y proyectos empresariales modernos se inclina hacia el ecosistema de Node.js, la versatilidad de Python y la robustez de Java.

¿Qué es?

Un lenguaje de programación del lado del servidor (server-side) es aquel cuyo código se ejecuta en el servidor web, no en el navegador del usuario. Su función principal es procesar las peticiones HTTP, aplicar la lógica de negocio y generar una respuesta dinámica (HTML, JSON, etc.) que se envía al cliente. A diferencia del código del lado del cliente, este permanece oculto al usuario final, protegiendo así la lógica interna y los datos sensibles.

A continuación, se analizan los lenguajes más utilizados según los datos de W3Techs, junto con sus características:

1. **PHP (73.3% - Líder histórico)**

- **Características:** Es un lenguaje interpretado con una sintaxis flexible, diseñado específicamente para la web. Tiene una integración nativa con servidores como Apache y un ecosistema gigantesco con frameworks (Laravel, Symfony) y CMS (WordPress, Drupal).
- **Implantación:** Domina la web, especialmente en CMS y e-commerce (WooCommerce, PrestaShop). Las versiones modernas como PHP 8.x han mejorado enormemente su rendimiento y características (JIT compiler, tipos estrictos).

2. **Ruby (6.4%)**
   - **Características:** Un lenguaje orientado a objetos, conocido por su sintaxis elegante y expresiva. Su framework Ruby on Rails popularizó el patrón MVC y el principio de "Convención sobre Configuración".
   - **Implantación:** Utilizado históricamente en startups de éxito como GitHub o Airbnb, aunque su cuota de mercado ha descendido frente a competidores más modernos.
3. **Java (5.4%)**
   - **Características:** Es un lenguaje compilado, fuertemente tipado y robusto, ideal para aplicaciones a gran escala. Frameworks como Spring Boot son el estándar de facto en el desarrollo empresarial.
   - **Implantación:** Es la opción preferida en sectores como la banca, telecomunicaciones y grandes corporaciones que requieren máxima escalabilidad, seguridad y fiabilidad.
4. **JavaScript / Node.js (5.0%)**
   - **Características:** Permite usar el mismo lenguaje en el front-end y el back-end. Su arquitectura asíncrona y orientada a eventos lo hace ideal para aplicaciones en tiempo real (WebSockets) y APIs de alto rendimiento. Sus frameworks más populares son Express.js y NestJS.
   - **Implantación:** Ha experimentado un crecimiento explosivo y es la tendencia dominante para nuevas APIs REST, microservicios y aplicaciones modernas.
5. **ASP.NET (4.8%)**
   - **Características:** Es el framework de Microsoft para construir aplicaciones web con C# o VB.NET. Su versión moderna, ASP.NET Core, es de código abierto y multiplataforma.
   - **Implantación:** Muy extendido en entornos corporativos que dependen del ecosistema de Microsoft (Azure, Windows Server).
6. **Python (1.2%)**
   - **Características:** Famoso por su sintaxis limpia y su versatilidad (desarrollo web, Machine Learning, ciencia de datos). Ofrece frameworks para todas las necesidades: Django (full-stack), Flask (microframework) y FastAPI (APIs de alto rendimiento).
   - **Implantación:** Aunque su porcentaje en webs tradicionales es bajo, Python es un líder indiscutible en la creación de APIs para aplicaciones de ciencia de datos y Machine Learning.

¿Para qué sirve?

El dominio de un lenguaje del lado del servidor es esencial para ser un desarrollador full-stack. Sus aplicaciones prácticas incluyen:

- **Generación de contenido dinámico:** Crear páginas personalizadas según el usuario (perfiles, historial).
- **Gestión de bases de datos:** Realizar operaciones CRUD (Create, Read, Update, Delete) con sistemas como MySQL, PostgreSQL o MongoDB.
- **Autenticación y autorización:** Validar credenciales, gestionar sesiones y tokens (JWT, OAuth).
- **Creación de APIs (REST/GraphQL):** Exponer servicios para ser consumidos por aplicaciones móviles, SPAs o microservicios.
- **Procesamiento de formularios:** Validar, sanear y almacenar los datos enviados por los usuarios.
- **Implementación de lógica de negocio:** Desde cálculos financieros y algoritmos de recomendación hasta el procesamiento de pagos.
- **Seguridad:** Proteger la aplicación contra ataques comunes (SQL Injection, XSS, CSRF) y cifrar datos sensibles.

Relación con el módulo

- **"Desarrollo Web en Entorno Servidor" (DWES):** Es la asignatura central donde se aprende a programar en PHP, Python (con Django/Flask) o Java (con Spring Boot), aplicando conceptos como la arquitectura MVC.
- **"Despliegue de Aplicaciones Web" (DAW):** Se aprende a configurar los servidores (Apache, Nginx) y entornos (Docker, Cloud) necesarios para ejecutar el código escrito en DWES.

Ejemplos

- **Procesar un formulario de contacto.**
    1. Un formulario HTML envía los datos (nombre, email, mensaje) a un script enviar.php mediante POST.
    2. El script PHP recibe los datos, los valida (ej. comprueba que el email es válido) y los sanea para prevenir ataques.
    3. Se conecta a una base de datos MySQL e inserta el mensaje en una tabla.
    4. Finalmente, redirige al usuario a una página de "Gracias".

Recurso externo

https://survey.stackoverflow.co/2023/#technology-web-frameworks-and-technologies

Referencias

W3Techs - Server-side Programming Languages Usage Statistics: (https://w3techs.com/technologies/overview/programming_language)

11. Características y posibilidades de desarrollo de una plataforma XAMPP.

Tu propio servidor web en un clic: XAMPP como el entorno de desarrollo local todo en uno. XAMPP es un paquete de software gratuito que instala y configura un servidor web completo (Apache, MariaDB, PHP, Perl) en tu ordenador personal, permitiendo desarrollar y probar aplicaciones web dinámicas de forma rápida y segura sin necesidad de un hosting externo.

¿Qué es?

XAMPP es un acrónimo que representa un conjunto de tecnologías de código abierto, empaquetadas en una distribución de software fácil de instalar. Su objetivo es simplificar la creación de un entorno de desarrollo web local. El nombre desglosa sus componentes principales:

- **X (Cross-platform):** Significa que funciona en múltiples sistemas operativos (Windows, macOS, Linux).
- **A (Apache):** Es el servidor web más popular y utilizado del mundo. Se encarga de recibir las peticiones HTTP de los navegadores y servir los archivos correspondientes.
- **M (MariaDB):** Es el sistema de gestión de bases de datos relacionales. Es un *fork* (derivación) de MySQL, creado por sus desarrolladores originales, y es totalmente compatible. Aquí es donde se almacenarán los datos de la aplicación.
- **P (PHP):** Es el lenguaje de programación del lado del servidor. Apache utiliza el intérprete de PHP para ejecutar los scripts que generan el contenido dinámico de las páginas.
- **P (Perl):** Otro lenguaje de programación del lado del servidor, aunque menos común hoy en día en el desarrollo web generalista que PHP.

En esencia, XAMPP es una herramienta que instala y pre-configura todos estos componentes para que trabajen juntos armónicamente, ahorrando al desarrollador el complejo proceso de instalar y vincular cada uno por separado. Proporciona un panel de control para iniciar y detener los servicios (Apache, MySQL) con un solo clic e incluye herramientas adicionales como **phpMyAdmin**, una interfaz web para gestionar las bases de datos MariaDB.

Es crucial entender que XAMPP está diseñado para **desarrollo y pruebas**, no para producción. Su configuración por defecto es abierta y permisiva para facilitar el desarrollo, lo que la haría insegura en un servidor real conectado a Internet.

¿Para qué sirve?

Las posibilidades y utilidades de XAMPP son fundamentales para el día a día de un desarrollador web:

- **Simular un Entorno de Producción:** Permite replicar el stack tecnológico de un servidor de hosting típico (LAMP: Linux, Apache, MySQL, PHP) en una máquina local.
- **Desarrollo Offline:** Puedes construir y probar una aplicación web completa sin necesidad de estar conectado a Internet.
- **Ciclo de Desarrollo Rápido:** Elimina la necesidad de subir archivos a un servidor remoto vía FTP cada vez que se realiza un cambio. Simplemente guardas el archivo en tu disco duro y recargas el navegador (http://localhost).
- **Entorno de Pruebas Seguro:** Es un "sandbox" perfecto para experimentar. Puedes probar nuevas versiones de PHP, instalar un CMS como WordPress para analizar su código, o ejecutar código que podría romper algo, todo ello sin afectar a ninguna aplicación en producción.
- **Aprender Tecnologías Server-Side:** Es la herramienta de entrada por excelencia para que los estudiantes aprendan PHP y la interacción con bases de datos SQL en un entorno controlado y sin coste.

Relación con el módulo

- **"Desarrollo Web en Entorno Servidor" (DWES):** Es el campo de juego principal. Todo el temario de PHP se desarrolla utilizando XAMPP. Los alumnos colocan sus scripts .php en la carpeta htdocs de XAMPP y los ejecutan a través de localhost en el navegador. Es aquí donde se aprende a procesar formularios, gestionar sesiones, subir archivos y crear la lógica de la aplicación.
- **"Acceso a Datos":** El componente MariaDB (MySQL) de XAMPP es la base de datos con la que se trabaja. Los alumnos utilizan la herramienta **phpMyAdmin** para diseñar y crear las tablas de sus bases de datos. Luego, desde los scripts PHP que crean en DWES, aprenden a conectar con esta base de datos, lanzar consultas SQL (SELECT, INSERT, UPDATE, DELETE) y gestionar los datos de forma persistente.
- **"Despliegue de Aplicaciones Web" (DAW):** Aunque XAMPP no se usa para el despliegue final, aprender a gestionar sus servicios a través del panel de control (iniciar, detener, ver logs, cambiar puertos) sirve como una introducción conceptual a la administración de servicios en un servidor real con Linux, que es un tema central en este módulo.

Ejemplos

- **Probar la instalación de PHP.**
    1. Instalar y ejecutar XAMPP, asegurándose de que el servicio Apache está corriendo.
    2. Navegar a la carpeta de instalación de XAMPP y abrir la subcarpeta htdocs.
    3. Crear un nuevo archivo llamado prueba.php.

4. Dentro del archivo, escribir una única línea de código: <?php phpinfo(); ?>.
5. Guardar el archivo, abrir un navegador web y escribir la URL: http://localhost/prueba.php.
6. Si todo está correcto, se mostrará una página detallada con toda la información de la configuración de PHP, confirmando que el servidor funciona.

Recurso externo

Cómo Instalar y Configurar XAMPP en Windows 10/11 - Guía Completa

- https://www.youtube.com/watch?v=IQ22Nme9t0M

12. En qué casos es necesaria la instalación de la máquina virtual Java (JVM) y el software JDK en el entorno de desarrollo y en el entorno de explotación.

Las dos caras de Java: el JDK para construir, la JVM para ejecutar. En el entorno de desarrollo, el JDK (Kit de Desarrollo de Java) es indispensable, ya que proporciona las herramientas para compilar y depurar el código. En producción, solo se necesita la JVM (Máquina Virtual de Java), el entorno de ejecución que interpreta el código ya compilado, garantizando eficiencia y seguridad.

¿Qué es?

Para entender cuándo se necesita cada componente, primero debemos definir con claridad la jerarquía del ecosistema Java:

1. **JVM (Java Virtual Machine - Máquina Virtual de Java):**
   Es el componente fundamental y el corazón de la portabilidad de Java ("Write Once, Run Anywhere"). La JVM es una **máquina abstracta**, una especificación que proporciona un entorno de ejecución en el que se puede ejecutar el bytecode de Java. El bytecode es el código intermedio, independiente de la plataforma, que se genera al compilar un archivo .java. Cada sistema operativo (Windows, Linux, macOS) tiene su propia implementación de la JVM, que traduce el bytecode a instrucciones nativas de la máquina. **La JVM es la que *ejecuta* el programa.**
2. **JRE (Java Runtime Environment - Entorno de Ejecución de Java):**
   Es la implementación concreta de la JVM. El JRE es un paquete de software que contiene todo lo necesario para *ejecutar* aplicaciones Java: la propia JVM, las bibliotecas de clases principales de Java (Java Class Libraries) y otros componentes de soporte. Si un usuario solo quiere ejecutar un programa hecho en Java, necesita instalar el JRE.
3. **JDK (Java Development Kit - Kit de Desarrollo de Java):**
   Es el "superconjunto". El JDK es un paquete de software destinado a los

desarrolladores. Incluye **todo lo que contiene el JRE** (es decir, la JVM y las librerías) y, además, añade un conjunto de **herramientas de desarrollo**. La más importante es el compilador (javac), que convierte el código fuente (.java) en bytecode (.class). También incluye otras herramientas como un depurador (jdb), un generador de documentación (javadoc) y un empaquetador de archivos (jar). **El JDK es para *construir* programas.**

La relación es: **JDK contiene al JRE, que a su vez contiene a la JVM.**

¿Para qué sirve? Casos de uso

La necesidad de instalar uno u otro depende exclusivamente de la tarea que se vaya a realizar:

- **Entorno de Desarrollo (ordenador de trabajo):**
  Es absolutamente necesaria la instalación del JDK.
  Un desarrollador necesita escribir código fuente (.java) y convertirlo en un programa ejecutable. Para ello, debe compilarlo. La única herramienta que proporciona el compilador javac es el JDK. Cuando trabajas con un IDE como IntelliJ IDEA o Eclipse, este utiliza internamente las herramientas del JDK para compilar, depurar y empaquetar tu aplicación (por ejemplo, en un archivo .jar). Sin el JDK, no puedes crear o modificar aplicaciones Java.
- **Entorno de Explotación o Producción (el servidor):**
  En teoría, solo es necesaria la instalación del JRE (o solo la JVM).
  El servidor donde se va a desplegar la aplicación no necesita compilar código; su única tarea es ejecutar el bytecode que el desarrollador ya ha compilado y empaquetado. Instalar el JDK completo en un servidor de producción es innecesario y, a menudo, desaconsejable por dos motivos:
  1. **Eficiencia:** El JRE es más ligero que el JDK, por lo que consume menos espacio en disco. En entornos de contenedores (Docker), donde cada megabyte cuenta, se utilizan imágenes con versiones mínimas del JRE.
  2. **Seguridad:** Instalar herramientas de desarrollo en un servidor de producción aumenta la "superficie de ataque". Un JRE mínimo expone menos binarios y librerías que podrían ser explotados.

Relación con el módulo

Esta distinción es crucial para conectar los conocimientos de varios módulos de DAW:

- **"Desarrollo Web en Entorno Servidor" (DWES):** Cuando aprendes a programar con Java y el framework Spring Boot en tu máquina local, estás en el **entorno de desarrollo**. Por tanto, la primera práctica es instalar el **JDK**.
- **"Acceso a Datos" (AD):** Para utilizar JDBC, JPA/Hibernate y conectar con bases de datos desde tu aplicación Java, necesitas el **JDK** para compilar tu código de acceso a datos.
- **"Despliegue de Aplicaciones Web" (DAW):** Este es el módulo donde la diferencia se hace más evidente. Aprenderás a empaquetar tu aplicación

Spring Boot en un archivo .jar ejecutable. Luego, configurarás un servidor Linux remoto (el **entorno de explotación**). En ese servidor, no instalarás el JDK completo, sino que instalarás un JRE (por ejemplo, sudo apt install default-jre) y luego ejecutarás tu aplicación con el comando java -jar mi-aplicacion.jar. Este proceso práctico consolida la teoría.

Ejemplos

- **"Hola Mundo" desde la consola.**
  1. **Desarrollo (necesita JDK):**
     - Escribes un archivo HolaMundo.java.
     - Abres una terminal y ejecutas javac HolaMundo.java. Este comando invoca al compilador del **JDK** y crea el archivo HolaMundo.class.
     - Luego ejecutas java HolaMundo. Este comando invoca a la **JVM** (incluida en el JDK) para ejecutar el bytecode.
  2. **Explotación (necesita JRE/JVM):**
     - Copias solo el archivo HolaMundo.class a otra máquina que únicamente tiene instalado el JRE.
     - En esa máquina, ejecutas java HolaMundo. El programa funciona perfectamente porque la JVM del JRE puede ejecutar el bytecode, aunque no tenga el compilador.

Recurso externo

Diferencias entre JDK, JRE y JVM

- https://codigojava.online/diferencias-entre-jdk-jre-y-jvm/

13. IDE más utilizados (características y grado de implantación actual).

14. Servidores HTTP /HTTPS más utilizados (características y grado de implantación actual).

Nginx lidera la era del alto rendimiento, mientras Apache consolida su legado. Los servidores web son el software esencial que procesa las peticiones y entrega el contenido de las aplicaciones. Nginx (33.4%) se impone por su eficiencia como proxy inverso, Apache (25.4%) se mantiene fuerte en el hosting tradicional, y Cloudflare Server (24.5%) redefine el perímetro de la web como un servicio de CDN y seguridad.

¿Qué es?

Un servidor web (o servidor HTTP/HTTPS) es un software que se ejecuta en un servidor y actúa como puerta de entrada a cualquier aplicación web. Su función principal es recibir peticiones HTTP/HTTPS desde los navegadores, procesarlas y devolver respuestas, ya sean archivos estáticos (HTML, CSS, imágenes) o contenido generado dinámicamente.

Los servidores web más utilizados son:

1. **Nginx (33.4% - Líder en crecimiento):**
   - **Características:** Su arquitectura es asíncrona y basada en eventos, lo que le permite manejar miles de conexiones concurrentes con un bajo consumo de memoria. Es excepcionalmente bueno como proxy inverso y balanceador de carga.
   - **Implantación:** Es la opción preferida para startups y sitios de alto rendimiento como Netflix. Es el estándar para servir como *frontend* a aplicaciones Node.js y Python.
2. **Apache HTTP Server (25.4% - El veterano robusto):**
   - **Características:** Utiliza un modelo basado en procesos o hilos. Su mayor fortaleza es su enorme ecosistema de módulos y la flexibilidad de su configuración a través de archivos .htaccess.
   - **Implantación:** Sigue siendo el pilar del stack LAMP (Linux, Apache, MySQL, PHP) y domina el sector del hosting compartido (cPanel, Plesk).
3. **Cloudflare Server (24.5% - El proxy en la nube):**
   - **Características:** No es un servidor web tradicional que se instala, sino una Red de Distribución de Contenido (CDN) que actúa como un proxy inverso global. Ofrece caché, protección DDoS y SSL/TLS gratuito.
   - **Implantación:** Se utiliza como una capa frontal de seguridad y rendimiento para cualquier otro servidor backend (sea Nginx, Apache, etc.).
4. **LiteSpeed (14.8% - La alternativa de alto rendimiento):**
   - **Características:** Ofrece compatibilidad con la configuración de Apache (.htaccess), facilitando la migración. Es conocido por su alto rendimiento, su caché avanzada (LSCache) y su soporte nativo de HTTP/3.
   - **Implantación:** Muy popular en proveedores de hosting optimizados para WordPress.
5. **Node.js (5.0% - El runtime que también es servidor):**
   - **Características:** Permite construir servidores web directamente en JavaScript. Su modelo de E/S no bloqueante es ideal para aplicaciones en tiempo real (chats, WebSockets).
   - **Implantación:** Es la base de innumerables APIs REST modernas y backends para SPAs. Generalmente se despliega detrás de Nginx, que actúa como proxy inverso.

¿Para qué sirve?

Dominar la configuración de un servidor web es crucial, ya que sus funciones van mucho más allá de simplemente "mostrar una página":

- **Servir contenido estático:** Entregar eficientemente los archivos del front-end (HTML, CSS, JS, imágenes).
- **Proxy inverso y balanceo de carga:** Distribuir el tráfico entre múltiples instancias de una aplicación back-end para garantizar la escalabilidad y la alta disponibilidad.
- **Gestión de SSL/TLS:** Centralizar el cifrado HTTPS, liberando a las aplicaciones back-end de esta tarea.
- **Cache y optimización:** Almacenar temporalmente respuestas para reducir la latencia y la carga del servidor.
- **Seguridad:** Actuar como primera línea de defensa, implementando firewalls de aplicaciones web (WAF) y limitando el número de peticiones (rate limiting).
- **Gateway de APIs:** Enrutar las peticiones a los microservicios correctos en arquitecturas complejas.

Relación con el módulo

- **Despliegue de Aplicaciones Web (DAW):** Aquí se aprende a instalar y configurar Apache y Nginx desde cero en un servidor Linux. Se trabajan competencias clave como la creación de *Virtual Hosts/Server Blocks*, la securización con certificados SSL (Let's Encrypt) y la automatización del despliegue con Docker.
- **Desarrollo Web en Entorno Servidor (DWES):** Los proyectos de este módulo necesitan un servidor web para ser accesibles.

Ejemplos

- **Servir un sitio estático con Apache.**
  Se configura un *Virtual Host* en un archivo de configuración de Apache:

```
<VirtualHost *:80>
ServerName mi-portfolio.com
DocumentRoot /var/www/html/portfolio
<Directory /var/www/html/portfolio>
   AllowOverride All
</Directory>
</VirtualHost>
```

  Cuando un usuario visita http://mi-portfolio.com, Apache recibe la petición, busca los archivos en el directorio /var/www/html/portfolio y los sirve.

Recurso externo

[GRÁFICO ESTADÍSTICO] Web Server Market Share - W3Techs
Este enlace lleva directamente a la fuente de datos utilizada. W3Techs ofrece gráficos

actualizados mensualmente sobre el uso de servidores web en todo Internet, permitiendo visualizar la cuota de mercado y las tendencias.

https://w3techs.com/technologies/overview/web_server

Referencias

1. **Nginx Official Documentation:** La fuente principal para aprender a configurar Nginx. (https://nginx.org/en/docs/)
2. **Apache HTTP Server Documentation:** La documentación oficial de Apache. (https://httpd.apache.org/docs/)

15. Apache HTTP vs Apache Tomcat

El Portero y el Motor: Apache HTTP, el servidor web universal, frente a Tomcat, el especialista en aplicaciones Java. Apache HTTP Server es un servidor web versátil diseñado para servir contenido estático y gestionar tráfico, mientras que Apache Tomcat es un contenedor de servlets (un servidor de aplicaciones) diseñado específicamente para ejecutar la lógica de aplicaciones web escritas en Java. A menudo, trabajan juntos para crear una arquitectura robusta y eficiente.

¿Qué es?

Aunque ambos son proyectos de la Apache Software Foundation y comparten la palabra "Apache" en su nombre, cumplen funciones fundamentalmente diferentes. Pensar en ellos como competidores es un error; son más bien colaboradores especializados.

- **Apache HTTP Server (a menudo llamado simplemente "Apache"):**
  Es un **servidor web**. Su principal propósito es escuchar peticiones a través del protocolo HTTP/HTTPS y servir archivos. Es extremadamente bueno y eficiente en su tarea principal: entregar contenido estático (HTML, CSS, JavaScript, imágenes, vídeos). Está escrito en C y es altamente configurable a través de un sistema de módulos que le permiten ampliar su funcionalidad. Por ejemplo, el módulo mod_php le permite interpretar y ejecutar scripts PHP. Es, por tanto, un software de propósito general para la web, agnóstico al lenguaje de back-end (siempre que exista un módulo para conectarse a él).
    - **Analogía:** Es el **recepcionista** de un gran edificio de oficinas. Atiende a todos los visitantes, responde preguntas sencillas directamente (sirve archivos estáticos) y dirige las consultas complejas al departamento especializado correcto.
- **Apache Tomcat:**
  Es un **contenedor de Servlets Java** y un **servidor de aplicaciones web Java**. No es un servidor web de propósito general. Su única misión es ejecutar aplicaciones web construidas sobre las tecnologías del ecosistema Java, como

Java Servlets, **JavaServer Pages (JSP)** y **Java Expression Language (EL)**. Proporciona el entorno de ejecución (la JVM y las APIs necesarias) que el código Java necesita para funcionar en un contexto web. Aunque Tomcat incluye un conector HTTP básico para poder funcionar de forma autónoma, no está tan optimizado ni es tan rico en características como Apache HTTP Server para gestionar conexiones de alto tráfico o servir archivos estáticos.

- **Analogía:** Es el **departamento de ingeniería Java** del edificio. No habla directamente con los visitantes. Recibe las tareas complejas que le pasa el recepcionista, las procesa utilizando sus herramientas y conocimientos especializados (ejecuta el código Java) y le devuelve el resultado al recepcionista para que este se lo entregue al visitante.

¿Para qué sirve?

Sus casos de uso derivan directamente de sus naturalezas distintas:

- **Usa Apache HTTP Server para:**
  - Servir el front-end de una aplicación (los archivos de React, Angular, Vue o HTML/CSS estáticos).
  - Actuar como **proxy inverso** y **balanceador de carga** para uno o varios servidores de aplicaciones (como Tomcat).
  - Gestionar la **terminación SSL/TLS** (el cifrado HTTPS) de forma centralizada.
  - Alojar sitios web basados en PHP (con mod_php) o Python (con mod_wsgi).
- **Usa Apache Tomcat para:**
  - **Ejecutar aplicaciones web escritas en Java.** Este es su único y principal propósito.
  - Servir como servidor embebido en frameworks modernos como **Spring Boot**, que a menudo empaquetan una versión de Tomcat dentro del propio archivo .jar de la aplicación para simplificar el despliegue.

La combinación es la clave: La arquitectura más común y robusta es usar **Apache HTTP Server como el punto de entrada** de cara a Internet (en el puerto 80/443) y **Tomcat ejecutándose en un puerto interno** (como el 8080). Apache recibe todas las peticiones, sirve los archivos estáticos directamente (imágenes, CSS) y, cuando una petición es para una ruta dinámica de la aplicación Java, la reenvía a Tomcat a través de un conector (como mod_proxy_ajp o mod_proxy_http).

Relación con el módulo

- En **DWES**, cuando desarrollas una aplicación con Spring Boot, por defecto estás utilizando un Tomcat embebido para probar tu aplicación en local. Accedes a ella a través de http://localhost:8080.
- En **DAW**, aprendes a llevar esa aplicación a un entorno de producción real. Esto implica:
  1. Instalar y configurar un servidor web como **Apache HTTP** en un servidor Linux.

2. Instalar una versión de **Tomcat** (o simplemente el JRE si usas el Tomcat embebido de Spring Boot).
3. Configurar Apache para que actúe como **proxy inverso**, redirigiendo las peticiones a la aplicación Java que se ejecuta en Tomcat. Esto enseña a crear una arquitectura segura, escalable y con buen rendimiento.

Ejemplos

- **Tomcat standalone:**
  1. Desarrollas un servlet "Hola Mundo" en Java y lo empaquetas en un archivo hola.war.
  2. Instalas Tomcat en tu máquina.
  3. Copias hola.war al directorio webapps de Tomcat.
  4. Tomcat automáticamente despliega la aplicación.
  5. Accedes a ella en tu navegador a través de http://localhost:8080/hola/mServlet. Todo el proceso es gestionado únicamente por Tomcat.

16. Navegadores HTTP /HTTPS más utilizados (características y grado de implantación actual).

Chrome domina el presente, obligando a los desarrolladores a pensar en todos los navegadores. Los navegadores son las aplicaciones cliente que interpretan y dan vida a nuestro código. El dominio absoluto de Chrome (71.77%) y la importancia de Safari en el ecosistema Apple (13.9%) hacen que la compatibilidad *cross-browser* sea una habilidad no negociable para cualquier desarrollador de DAW.

¿Qué es?

Un navegador web es una aplicación de software cliente que actúa como un intérprete y renderizador de las tecnologías web estándar (HTML, CSS, JavaScript), transformando el código fuente en las interfaces gráficas interactivas que los usuarios ven. Cada navegador se distingue principalmente por sus dos componentes internos clave: el motor de renderizado y el motor de JavaScript.

El panorama actual del mercado, según los datos proporcionados, está dominado por unos pocos actores principales:

- **Google Chrome (71.77%):** El líder absoluto del mercado, utiliza el motor de renderizado **Blink** y el motor de JavaScript **V8**.
- **Apple Safari (13.9%):** El navegador por defecto en el ecosistema de Apple, utiliza el motor de renderizado **WebKit** y el motor de JavaScript **JavaScriptCore (Nitro)**.

- **Microsoft Edge (4.67%):** En sus versiones modernas, está basado en Chromium, por lo que comparte los motores **Blink** y **V8** con Chrome.
- **Mozilla Firefox (2.17%):** El principal navegador independiente y de código abierto, utiliza sus propios motores: **Gecko** para el renderizado y **SpiderMonkey** para JavaScript.

Además de estos motores, un navegador incluye una interfaz de usuario, un subsistema de red para gestionar las peticiones HTTP/HTTPS, mecanismos de almacenamiento en el cliente (localStorage, cookies) y, fundamentalmente, las **Herramientas de Desarrollador (DevTools)**.

¿Para qué sirve?

Desde la perspectiva de un desarrollador, el navegador es la **plataforma de ejecución** de toda la lógica del lado del cliente. Su propósito es múltiple:

- **Ejecutar Aplicaciones Web:** Es el entorno donde viven y se ejecutan las Single Page Applications (SPAs) construidas con frameworks como React, Vue o Angular.
- **Desarrollo y Depuración:** Las DevTools son la navaja suiza del desarrollador front-end. Permiten inspeccionar el DOM en tiempo real, depurar código JavaScript paso a paso en la consola, monitorizar todas las peticiones de red y analizar el rendimiento de la aplicación.
- **Proporcionar APIs Modernas:** Los navegadores exponen un conjunto cada vez mayor de APIs que permiten a las aplicaciones acceder a funcionalidades del dispositivo, como la geolocalización, la comunicación en tiempo real (WebSockets) o el funcionamiento offline (Service Workers).
- **Garantizar la Seguridad:** Implementan políticas de seguridad cruciales como la Same-Origin Policy (SOP) y la Content Security Policy (CSP) para proteger a los usuarios de ataques como el Cross-Site Scripting (XSS).

Relación con el módulo

- **"Desarrollo Web en Entorno Cliente" (DWEC):** Este módulo se desarrolla íntegramente dentro del navegador. El objetivo es dominar JavaScript para manipular el DOM y utilizar las APIs del navegador.
- **"Diseño de Interfaces Web" (DIW):** El foco aquí es la compatibilidad *cross-browser*. Se aprende a escribir CSS que se renderice correctamente en los diferentes motores (Blink, WebKit, Gecko).
- **"Desarrollo Web en Entorno Servidor" (DWES):** Aunque el código se escribe para el servidor, es el navegador el que inicia las peticiones, gestiona las cookies de sesión y consume las APIs que se construyen.
- **"Despliegue de Aplicaciones Web" (DAW):** Se implementan estrategias de optimización (minificación, compresión, caché) que afectan directamente a cómo el navegador carga y renderiza la aplicación.

Ejemplos

- **Renderizado de una página.**
    1. Un usuario introduce una URL. El subsistema de **Networking** del navegador envía una petición HTTP GET.
    2. Al recibir el HTML, el **Motor de Renderizado** (ej. Blink) lo parsea para construir el árbol DOM.
    3. El motor solicita los recursos enlazados (CSS, JS). El CSS se usa para construir el CSSOM.
    4. El DOM y el CSSOM se combinan para crear el árbol de renderizado, que se "pinta" en la pantalla.
    5. Finalmente, el **Motor de JavaScript** (ej. V8) ejecuta el código JS para añadir interactividad.

Recurso externo

[GRÁFICO ESTADÍSTICO INTERACTIVO] Browser Market Share Worldwide - StatCounter

- https://gs.statcounter.com/browser-market-share

Referencias

1. **web.dev (by Google) - Browser Rendering Explained:** Una serie de artículos de Google que explican el "Critical Rendering Path", fundamental para entender el rendimiento. (https://web.dev/articles/critical-rendering-path/render-tree-construction)

17. Generadores de documentación HTML (PHPDoc): PHPDocumentor, ApiGen, …

El código que se explica solo: cómo PHPDoc y sus generadores convierten los comentarios en manuales de usuario. PHPDoc es un estándar para escribir comentarios en el código PHP que, al ser procesados por herramientas como PHPDocumentor, generan automáticamente una documentación HTML completa y navegable de una aplicación. Esta práctica es esencial para el mantenimiento, la colaboración en equipo y la integración con los IDEs modernos.

¿Qué es?

PHPDoc no es una herramienta, sino un **estándar de documentación** adaptado del Javadoc de Java para el lenguaje PHP. Consiste en un formato específico para escribir comentarios, llamados **bloques de documentación (DocBlocks)**, justo antes de un elemento del código (una clase, un método, una propiedad, una función, etc.). Estos bloques comienzan con /** y terminan con */.

Dentro de un DocBlock, se utiliza una sintaxis de etiquetas (tags), que empiezan con @, para describir el elemento de código de forma estructurada. Las etiquetas más comunes son:

- @param [tipo] $[nombre_variable] [descripción]: Describe un parámetro de una función o método.
- @return [tipo] [descripción]: Describe el valor que devuelve una función o método.
- @throws [tipo_excepcion] [descripción]: Indica que una función puede lanzar una excepción.
- @var [tipo]: Describe el tipo de una propiedad de una clase.
- @author [nombre] [email]: Indica el autor del código.
- @version [número]: Especifica la versión del elemento.

Los **generadores de documentación HTML** (como PHPDocumentor o ApiGen) son programas que **analizan (parsean)** todo el código fuente de un proyecto PHP, buscan estos DocBlocks y utilizan la información que contienen para construir un sitio web HTML completo. Este sitio web actúa como un manual de referencia de la API del proyecto, con una página para cada clase, un listado de sus métodos, los parámetros que aceptan, lo que devuelven, etc., todo interconectado con hipervínculos.

- **PHPDocumentor:** Es la herramienta más veterana, robusta y considerada el estándar de facto para generar documentación a partir de PHPDoc. Es altamente configurable.
- **ApiGen:** Fue otra alternativa popular, conocida por su diseño moderno y su soporte para características más nuevas de PHP. Su desarrollo ha sido menos activo en los últimos años en comparación con PHPDocumentor.

¿Para qué sirve?

La generación automática de documentación es una práctica con beneficios inmensos:

1. **Mantenimiento a largo plazo:** El código bien documentado es infinitamente más fácil de entender y modificar meses o años después de haber sido escrito, tanto por el autor original como por otros desarrolladores.
2. **Colaboración en equipo:** Actúa como un "contrato". La documentación de un método deja claro cómo debe ser utilizado por otros miembros del equipo, qué datos espera y qué devuelve, sin necesidad de leer su implementación interna.
3. **Integración con IDEs:** Este es un beneficio inmediato y masivo. Los IDEs modernos como PhpStorm o Visual Studio Code con extensiones de PHP **leen los DocBlocks en tiempo real**. Gracias a esto, pueden ofrecer:
    - **Autocompletado de código** mucho más preciso.
    - **Type Hinting** (sugerencias de tipo) incluso en versiones antiguas de PHP o cuando el tipado estricto no es posible.
    - **Documentación emergente:** Al pasar el ratón por encima de una llamada a una función, el IDE muestra una ventana con la descripción, los parámetros y el valor de retorno que se extrajeron del DocBlock.

4. **Creación de APIs públicas:** Si desarrollas una librería o un paquete para que otros lo usen, la documentación generada es su manual de usuario. Es un requisito indispensable para cualquier proyecto de código abierto serio.

Relación con el módulo

Esta práctica es una competencia clave del módulo de **"Desarrollo Web en Entorno Servidor" (DWES)**. Aunque el foco principal del módulo es aprender a programar en PHP, la forma *profesional* de hacerlo implica escribir código limpio, legible y mantenible.

- Se enseña junto con la **Programación Orientada a Objetos (POO)**. Documentar clases, propiedades, métodos y sus relaciones de herencia es fundamental.
- Es una habilidad que demuestra madurez como desarrollador. Un proyecto de fin de curso en DWES que incluya código bien documentado con PHPDoc y su correspondiente documentación generada recibirá una valoración mucho más alta.
- Se conecta con el módulo de **"Sistemas de Gestión Empresarial" (SGE)**, ya que la documentación del software es una parte crucial de los estándares de calidad y de los procesos de desarrollo en una empresa real.

Ejemplos

- **Una función simple.**

```
    /**
 * Calcula la suma de dos números enteros.
 *
 * Esta función toma dos enteros como entrada y devuelve su suma.
 *
 * @param int $a El primer sumando.
 * @param int $b El segundo sumando.
 * @return int La suma de $a y $b.
 */
function sumar(int $a, int $b): int {
    return $a + $b;
}
```

Al ejecutar PHPDocumentor sobre este código, generará una página HTML para la función sumar que mostrará su descripción, la lista de parámetros (con su tipo y descripción) y lo que devuelve.

Recurso externo

PHPDoc (PSR-5)

- https://github.com/php-fig/fig-standards/blob/master/proposed/phpdoc.md

18. Repositorios de software – sistemas de control de versiones : GIT , CVS, Subversion, ...

La máquina del tiempo del código: cómo los sistemas de control de versiones como Git salvan proyectos y potencian el trabajo en equipo. Un sistema de control de versiones (VCS) es una herramienta que rastrea cada cambio en el código, permitiendo a los desarrolladores colaborar sin caos, revertir errores y mantener un historial completo del proyecto. Git, con su modelo distribuido, se ha coronado como el estándar indiscutible de la industria.

¿Qué es?

Un **Sistema de Control de Versiones (VCS)** es un software que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, permitiendo recuperar versiones específicas más adelante. Un **repositorio** es la base de datos donde el VCS almacena todo este historial de cambios.

Existen dos arquitecturas principales para los VCS:

1. **Centralizados (CVCS - Centralized Version Control Systems):**
   - **Cómo funcionan:** Existe un único servidor central que contiene todo el historial del proyecto. Los desarrolladores "hacen check-out" de una copia de trabajo desde ese servidor, realizan cambios y luego "hacen check-in" (o *commit*) de esos cambios de vuelta al servidor central.
   - **Ejemplos: CVS (Concurrent Versions System)** y **Apache Subversion (SVN)**.
   - **Inconveniente:** Si el servidor central se cae, nadie puede colaborar ni guardar sus cambios. Si el disco duro del servidor se corrompe y no hay copias de seguridad, se pierde todo el historial.
2. **Distribuidos (DVCS - Distributed Version Control Systems):**
   - **Cómo funcionan:** En lugar de tener una única copia central, cada desarrollador "clona" el repositorio completo en su máquina local. Esto significa que cada desarrollador tiene una copia íntegra de todo el historial del proyecto. Los cambios se guardan (*commit*) primero en el repositorio local. Luego, se pueden "sincronizar" los cambios con otros repositorios, como uno centralizado en un servidor (ej. GitHub, GitLab).
   - **Ejemplo: Git**. (Otros como Mercurial existen, pero Git es el dominador absoluto).
   - **Ventaja:** Permite trabajar offline, es mucho más rápido para operaciones comunes (commit, diff) y es más seguro, ya que cada clon es una copia de seguridad completa del repositorio.

Hoy en día, **Git es el estándar de facto**. CVS y Subversion se consideran sistemas *legacy* (heredados), aunque es útil conocer su funcionamiento conceptualmente.

¿Para qué sirve?

El uso de un VCS es indispensable en el desarrollo de software por cuatro razones principales:

- **Historial y Auditoría:** Permite ver quién cambió qué, cuándo y por qué (a través de los mensajes de commit). Es una "máquina del tiempo" que permite volver a cualquier estado anterior del proyecto.
- **Trabajo en Equipo y Colaboración:** Es su función más potente. Permite que múltiples desarrolladores trabajen en el mismo proyecto de forma paralela. El VCS ayuda a fusionar (*merge*) los cambios de todos y a gestionar los conflictos cuando dos personas modifican la misma parte del código.
- **Ramificación (Branching):** Permite crear "ramas" o líneas de desarrollo independientes. Se puede crear una rama para desarrollar una nueva funcionalidad sin afectar la versión principal y estable del código (main o master). Una vez que la funcionalidad está terminada y probada, la rama se fusiona de nuevo con la principal.
- **Copia de Seguridad y Recuperación:** Si cometes un error catastrófico, puedes revertir fácilmente a una versión anterior que funcionaba. Si tu repositorio local se corrompe o pierdes tu ordenador, el código está a salvo en el repositorio remoto.

Relación con el módulo

- **"Entornos de Desarrollo":** Es el módulo donde se suelen introducir las herramientas de desarrollo, y Git es la más importante. Aprender el flujo de trabajo básico de Git es una competencia fundamental de este módulo.
- **"Desarrollo Web en Entorno Cliente" (DWEC) y "Servidor" (DWES):** Todo el código que se escribe en estos módulos debe ser gestionado con un VCS. Es una práctica obligatoria en cualquier proyecto que no sea trivial, especialmente en los proyectos en grupo de final de curso.
- **"Despliegue de Aplicaciones Web" (DAW):** El despliegue moderno se basa en la **Integración Continua y Despliegue Continuo (CI/CD)**. Estos sistemas automáticos se activan mediante eventos de Git. Por ejemplo, al hacer un git push a la rama main, se puede disparar un proceso automático que ejecuta tests, construye la aplicación y la despliega en el servidor de producción.

Ejemplos

- **Flujo de trabajo personal.**
  Creando tu portfolio personal.
  1. En la carpeta de tu proyecto, ejecutas git init para crear un nuevo repositorio.
  2. Creas tu index.html y style.css.

3. Ejecutas git add . para añadir los nuevos archivos al área de "preparación" (*staging area*).
4. Ejecutas git commit -m "Versión inicial del portfolio con HTML y CSS básicos". Has guardado la primera "fotografía" de tu proyecto.
5. Modificas el CSS para añadir un diseño responsive. Repites los pasos 3 y 4 con un mensaje de commit descriptivo: git commit -m "Añadido diseño responsive para móviles".
6. Si el nuevo diseño no te gusta, puedes volver a la versión anterior con un solo comando.

Recurso externo

Git explicado en 100 segundos

- https://www.youtube.com/watch?v=hwP7WQkmECE

19. Propuesta de configuración del entorno de desarrollo para la asignatura de Desarrollo web del lado servidor en este curso (incluyendo las versiones): xxx-USED y xxx-WXED.

20. Propuesta de configuración del entorno de explotación para la asignatura de Desarrollo web del lado servidor en este curso (incluyendo las versiones): xxx-USEE.

21. Realizar un estudio sobre los siguientes conceptos y su relación con el desarrollo de aplicaciones web:

22. CMS – Sistema de gestión de contenidos

23. ERP – Sistema de planificación de los recursos empresariales

24. Elegir y realizar un estudio y una presentación para la exposición del trabajo sobre una de las siguientes arquitecturas de desarrollo de Aplicaciones Web:

- MEAN (con MongoDB y con MySQL)
- Java EE vs Spring
- Microsoft .NET
- Angular 7
- Symfony
- Laravel
- CakePHP
- CodeIgniter

GLOSARIO DE TÉRMINOS RELACIONADOS CON DWES

➢ Contenidos y la diferencia entre los módulos que tienes en este curso.

➢ Protocolos TCP/IP. Socket.

Conjunto de protocolos de red en los que se basa Internet.

Son las "leyes de la física y la logística" de Internet. No son una sola cosa, sino una **pila de reglas jerárquicas** que resuelven el problema de enviar información entre dos puntos lejanos de forma fiable. Cada capa de la pila resuelve un problema, confiando en que la capa inferior ha resuelto el suyo.

- **IP:** Se encarga de una sola cosa: poner una etiqueta con una dirección de origen y una de destino a cada paquete de datos. Es el **servicio de correos universal**; su trabajo es el enrutamiento, no la fiabilidad.
- **TCP:** Trabaja encima de IP. Su misión es transformar el servicio de correos poco fiable de IP en una **conversación telefónica garantizada**. Corta el mensaje en trozos numerados, se asegura de que todos lleguen, los reordena y pide que se reenvíen los perdidos. Es el **control de calidad** de la comunicación.

Socket

Un punto final en una comunicación bidireccional.

Es el "enchufe" o "puerta de embarque" específico para una conversación en un ordenador. Un ordenador puede tener cientos de conversaciones de red a la vez (navegador, email, Spotify...). ¿Cómo sabe qué datos son para qué programa? El **socket** es la respuesta. Es la combinación única de una **dirección IP (el aeropuerto) y un número de Puerto (la puerta de embarque)**. Cuando un servidor "escucha" en un puerto (ej. 443), está esperando en una puerta de embarque específica a que lleguen los pasajeros (datos) de un vuelo concreto.

➢ Protocolo HTTP / HTTPS

Son protocolos para transferir información en la web, pero la diferencia clave es la seguridad: HTTP transmite datos en texto plano, mientras que HTTPS (HTTP Secure) los cifra usando SSL/TLS para proteger la información confidencial entre el navegador del usuario y el servidor web

➢ HTML

Hypertext Markup Language o el lenguaje de marcas hipertexto. Es un lenguaje de marcación que sirve para definir el contenido de las páginas web. Se compone en base a etiquetas, también llamadas marcas o tags, con las cuales conseguimos expresar las partes de un documento, cabecera, cuerpo, encabezados, párrafos, etc. En definitiva, el contenido de una página web.

Lenguaje de marcado para crear páginas web. Es el "esqueleto" o la "estructura semántica" de un documento web. No es un lenguaje de programación; no tiene lógica. Es un lenguaje descriptivo. Su única misión es decirle al navegador qué es cada pieza de contenido: "Esto es un título principal", "esto es un párrafo", "esto es una imagen", "esto es un enlace a otro documento". CSS luego se encargará de cómo se ve y JavaScript de cómo se comporta, pero HTML define la estructura y el significado puro.

➢ XML

Lenguaje de marcado que define un conjunto de reglas para codificar documentos. Es una "gramática para crear tus propios lenguajes de marcado". Mientras que HTML tiene un vocabulario fijo (<h1>, <p>, etc.), XML te da la libertad de inventar tus propias etiquetas para describir tus datos. Su propósito no es mostrar información a humanos, sino estructurar y transportar datos de forma que sea legible tanto para humanos como para máquinas. Es como si te dieran las reglas para crear un diccionario, en lugar de darte un diccionario ya hecho.
Ejemplo: <factura><cliente>Juan</cliente><total>100</total></factura>

➢ JSON

Formato de texto ligero para el intercambio de datos.
Es el "post-it" o la "nota rápida" para intercambiar datos entre sistemas. Surgió como una alternativa mucho más simple y ligera a XML. Su filosofía es describir datos usando una sintaxis mínima, directamente inspirada en los objetos de JavaScript, que consiste en pares de clave-valor. Es el formato de datos de facto en las APIs web modernas por su simplicidad y eficiencia. No tiene la rigidez de XML, pero es perfecto para enviar "listas de la compra" o "tarjetas de visita" entre el cliente y el servidor.
Ejemplo: {"cliente": "Juan", "total": 100}

➢ Lenguajes de programación embebidos en HTML

Código de un lenguaje de scripting insertado dentro de un archivo HTML.
Es la técnica que permite transformar un "documento" estático en una "plantilla" dinámica. La idea fundamental es que un servidor no envía un archivo HTML pre-escrito. En su lugar, lee un archivo que es mayormente HTML, pero que contiene "islas" de código de programación (como PHP, <?php ... ?>). El servidor ejecuta solo esas islas de código, y el resultado de esa ejecución (por ejemplo, el nombre de un usuario sacado de una base de datos) reemplaza al código en el documento final. El navegador del cliente nunca ve el código de programación, solo recibe el HTML puro resultante. Es el mecanismo original para la generación dinámica de páginas.

➢ Arquitecturas de desarrollo web

El conjunto de patrones y técnicas utilizadas para diseñar y construir una aplicación web.
Son los "planos maestros" o la "filosofía de diseño" de una aplicación. No se refieren al código en sí, sino a cómo se organizan las grandes piezas del sistema y cómo se comunican entre ellas. La arquitectura responde a la pregunta: "¿Cómo evitamos que nuestro proyecto se convierta en un caos inmanejable a medida que crece?". Es la aplicación de la Ley de la Abstracción Jerárquica para dividir el problema en partes más pequeñas y con responsabilidades claras (ej. Modelo-Vista-Controlador, Microservicios).

➢ Framework de desarrollo Web

Una plataforma de software para desarrollar aplicaciones web.
Es un "kit de construcción" o un "chasis" pre-fabricado para tu aplicación. Un framework es una arquitectura opinada que viene con un montón de herramientas y código reutilizable para las tareas más comunes (gestionar rutas, hablar con la base de datos, manejar la seguridad, etc.). Su principal característica es la Inversión de

Control: no llamas al framework, el framework llama al código. Te da una estructura y un camino a seguir, permitiéndote centrarte en la lógica específica de tu aplicación en lugar de reinventar la rueda una y otra vez. Ejemplos: Laravel, Symfony, Spring.

➢ ERP

Sistema de planificación de recursos empresariales.
Es el "sistema nervioso central" digital de una empresa. Su objetivo es integrar y centralizar todos los procesos de negocio clave en una única plataforma de software. En lugar de tener un programa para contabilidad, otro para inventario, otro para recursos humanos y otro para ventas, un ERP los une a todos. Su núcleo es una base de datos unificada. Cuando Ventas registra un nuevo pedido, Inventario ve automáticamente que debe reducir el stock y Contabilidad ya tiene la factura lista para generar. Su propósito es crear una única fuente de verdad para toda la organización, rompiendo los "silos" de información entre departamentos.

➢ CMS

Sistema de gestión de contenidos.
Es una "fábrica de páginas web" diseñada para ser operada por no-programadores. Su innovación fundamental es la separación radical entre el contenido y la presentación. Un periodista (usuario) puede escribir un artículo en un formulario simple, sin saber nada de HTML o CSS. El CMS guarda ese texto en una base de datos. Cuando un visitante llega a la web, el CMS recupera el texto y lo "viste" con una plantilla de diseño predefinida para generar la página final. Permite que la gestión del qué se dice (contenido) sea independiente del cómo se ve (diseño), democratizando la publicación en la web. Ejemplos: WordPress, Prestashop.

➢ PHP

Un popular lenguaje de scripting de código abierto especialmente adecuado para el desarrollo web.

Es el "pegamento" dinámico del lado del servidor. Su razón de ser es recibir una petición de un navegador, realizar lógica (hablar con una base de datos, hacer cálculos, comprobar si un usuario ha iniciado sesión) y, finalmente, **devolver texto como respuesta**, que casi siempre es HTML. Nació para ser embebido directamente en HTML, lo que lo hizo increíblemente fácil de aprender y usar para añadir dinamismo a páginas estáticas. Aunque ha evolucionado enormemente, su esencia sigue siendo esa: es un lenguaje diseñado desde su origen para vivir en un servidor y responder a las peticiones web.

➢ IDE

Un entorno de desarrollo integrado.

Es el "taller" o la "cabina de piloto" del programador. Un IDE va mucho más allá de ser un simple editor de texto (como el Bloc de Notas). Integra en una sola ventana *todas* las herramientas que un desarrollador necesita para ser productivo:

- **Editor de código inteligente:** Que autocompleta, colorea la sintaxis y señala errores al instante.
- **Depurador (Debugger):** Una "máquina del tiempo" que permite pausar la ejecución del programa, inspeccionar variables y avanzar línea por línea para encontrar fallos.
- **Integración con herramientas:** Como sistemas de control de versiones (Git), gestores de bases de datos y terminales de comandos.
  Su objetivo es reducir la "fricción" del desarrollo, manteniendo al programador en un estado de flujo y concentración. Ejemplos: Visual Studio Code, PhpStorm, NetBeans.

➢ Navegador

Un software para acceder a la World Wide Web.

Es un "intérprete" y "renderizador" de lenguajes web. Su misión principal es triple:

- **Hacer de Cliente:** Iniciar la conversación, enviando peticiones HTTP a los servidores.
- **Interpretar:** Recibir la respuesta (HTML, CSS, JavaScript) y entenderla. Lee el HTML para saber la estructura, el CSS para saber el estilo y ejecuta el código JavaScript para añadir interactividad.
- **Renderizar:** Dibujar el resultado final en la pantalla del usuario en forma de una página web visual y funcional.
  Es la pieza de software más crítica del lado del cliente, convirtiendo código abstracto en una experiencia de usuario tangible.

➢ Repositorio

Un lugar centralizado donde se almacenan y gestionan datos.

Es el "historial médico" completo y el "laboratorio" de un proyecto de software. Un repositorio de control de versiones (como Git) no solo almacena la versión actual del código. Almacena **cada cambio individual que se ha hecho a lo largo de toda la historia del proyecto**, quién lo hizo, cuándo y por qué. Esto le otorga superpoderes al equipo:

- **Máquina del tiempo:** Permite volver a cualquier punto anterior del proyecto.
- **Trabajo en paralelo seguro:** Varios desarrolladores pueden trabajar en diferentes "ramas" (versiones alternativas) sin estorbarse, y luego fusionar sus cambios de forma controlada.

- **Documentación viva:** El historial de cambios (commits) sirve como un diario de desarrollo.

➢ Entorno de Desarrollo

La configuración de software y hardware en la que un desarrollador crea una aplicación.

Es el "simulador de vuelo" o el "laboratorio de pruebas" de un programador. Es una réplica del sistema real (el servidor de producción) que se ejecuta en la máquina local del desarrollador o en un servidor de pruebas. Su propósito es crear un **espacio seguro y controlado para construir y experimentar**. Está optimizado para la depuración, la velocidad de desarrollo y la experimentación, no para el rendimiento o la seguridad. Es el lugar donde los errores no cuestan dinero y donde se puede "romper" todo sin afectar a los usuarios reales.

➢ Entorno de Explotación o Producción

El entorno donde se ejecuta el software y es utilizado por los usuarios finales.

Es el "escenario" o la "tienda abierta al público". Este es el entorno real, el que ven los usuarios. A diferencia del entorno de desarrollo, está **optimizado para la fiabilidad, el rendimiento y la seguridad**. El código se comprime, los mensajes de error detallados se ocultan, se activan todas las cachés y se monitoriza constantemente. Cualquier cambio en este entorno es crítico y se hace de forma muy controlada (a través de un despliegue o *deploy*). Es el sistema "de verdad", donde los errores tienen consecuencias reales.

➢ Gestión de la configuración. Control de cambios. Mantenimiento de la aplicación.

Disciplinas de la ingeniería de software para gestionar la evolución de un sistema.

Son las **"reglas de tráfico, el manual de mantenimiento y el registro de obras" de un proyecto de software.** No se refieren al código en sí, sino a los procesos para evitar que un proyecto exitoso se autodestruya por el caos del cambio a lo largo del tiempo.

- **Gestión de la Configuración:** Es el **inventario y control de todos los componentes del sistema**. No solo el código, sino también las versiones de las librerías, los archivos de configuración del servidor, la documentación, etc. Su objetivo es poder **reproducir el entorno** exacto en cualquier momento.
- **Control de Cambios:** Es la **burocracia (necesaria) para modificar el sistema**. Define el proceso formal para proponer, evaluar, aprobar e

implementar un cambio. Evita que se introduzcan modificaciones impulsivas que puedan desestabilizar el sistema. Es el "permiso de obra" para el software.

- **Mantenimiento de la Aplicación:** Es todo el trabajo que se realiza *después* del lanzamiento inicial. No es solo corregir errores (*mantenimiento correctivo*), sino también adaptar el software a nuevos entornos (ej. una nueva versión de PHP - *adaptativo*) y añadir pequeñas mejoras (*perfectivo*). Es el **mantenimiento a largo plazo del edificio** para que no se deteriore.

➢ Web Services

Una tecnología que permite la comunicación entre aplicaciones a través de Internet.

Son los "enchufes" o "APIs" que permiten que los programas hablen entre sí, usando el lenguaje universal de la web (HTTP). Un servicio web no tiene interfaz gráfica; no es para humanos. Es una URL que, cuando es llamada por otro programa, devuelve datos puros y estructurados (normalmente JSON), en lugar de una página HTML. Su propósito es **exponer una funcionalidad específica de tu aplicación para que otros sistemas puedan consumirla**, sin necesidad de saber cómo está implementada por dentro. Es la base de la comunicación máquina-a-máquina en la web moderna (ej. una app móvil que habla con su backend).

➢ AJAX

Una técnica de desarrollo web para crear aplicaciones interactivas.

Es la técnica que le dio a las páginas web la capacidad de "susurrar" al servidor sin tener que gritar (recargar la página entera). Antes de AJAX, cada interacción con el servidor (enviar un formulario, cargar nuevos datos) implicaba un parpadeo y una recarga completa de la página. AJAX permite que el JavaScript que se ejecuta en el navegador del usuario envíe pequeñas peticiones HTTP al servidor **en segundo plano**. El servidor responde con datos (JSON, normalmente), y JavaScript usa esos datos para actualizar solo una pequeña porción de la página, sin interrumpir al usuario. Es la tecnología que hizo posible las aplicaciones web fluidas y dinámicas que conocemos hoy (como Google Maps o Gmail).

➢ Desarrollo de aplicaciones multicapa. Estrategias de diseño de aplicaciones Web.

Un modelo de arquitectura de software.

Es el principio de "dividir y vencer" aplicado a la estructura de una aplicación. La idea es separar el código en **capas horizontales**, cada una con una única responsabilidad, como si fueran los pisos de un edificio.

- **Capa de Presentación (el escaparate):** Solo se preocupa de mostrar cosas al usuario y recoger sus interacciones.
- **Capa de Lógica de Negocio (la trastienda):** Contiene las reglas y el "cerebro" de la aplicación.
- **Capa de Acceso a Datos (el almacén):** Su único trabajo es hablar con la base de datos.
  El beneficio es el **desacoplamiento**: puedes cambiar completamente el "almacén" (la base de datos) sin que el "escaparate" (la interfaz de usuario) se vea afectado, siempre que la "trastienda" sepa cómo manejar el cambio. Es la estrategia de diseño fundamental para crear software mantenible y escalable.

➢ Aplicaciones basadas en microservicios.

Un estilo arquitectónico que estructura una aplicación como una colección de servicios pequeños y autónomos.

Es la filosofía de "construir con LEGO en lugar de con una sola pieza de mármol". En lugar de una aplicación monolítica gigante donde todo está interconectado, se descompone la funcionalidad en **servicios independientes y especializados**. Cada microservicio es una mini-aplicación en sí misma (con su propia base de datos si es necesario) que se ocupa de una única tarea (ej. servicio de usuarios, servicio de pagos, servicio de inventario). Se comunican entre sí a través de APIs (Web Services).

- **Ventajas:** Permite que diferentes equipos trabajen de forma independiente, que cada servicio se pueda escalar por separado, y que un fallo en un servicio no tumbe toda la aplicación. Es una arquitectura compleja pero muy potente para sistemas grandes.

➢ SaaS: Sofware as a Service.

Un modelo de distribución de software en el que las aplicaciones son alojadas por un proveedor y puestas a disposición de los clientes a través de una red, típicamente Internet.

Es el modelo de "alquilar el uso de un programa en lugar de comprar el programa". En lugar de instalar un software en tu propio ordenador (como el antiguo Microsoft Office en CD-ROM), accedes a él a través de tu navegador web. El software, los servidores, el mantenimiento y las actualizaciones son responsabilidad del proveedor. Tú solo pagas una suscripción por el servicio. Su esencia es la **transformación de un producto de software en un servicio continuo**. Ejemplos: Google Docs, Salesforce, Office 365, Spotify.

➢ Control de acceso a la aplicación web o los Web Services.

Mecanismos que restringen el acceso a recursos basándose en la identidad del usuario.

Es el "portero" y el "juego de llaves" de tu aplicación. Su misión es responder a dos preguntas fundamentales, y siempre en este orden:

- **Autenticación (¿Quién eres?):** Es el acto de **verificar la identidad** de un usuario. El usuario presenta unas credenciales (como usuario/contraseña o un token) y el sistema las comprueba. Si son correctas, el usuario está "autenticado". Es como enseñar el DNI para entrar a un edificio.
- **Autorización (¿Qué tienes permiso para hacer?):** Una vez que sabemos *quién* es el usuario, este mecanismo decide *a qué* tiene acceso. Se basa en **roles y permisos**. Un usuario normal (rol "lector") puede ver artículos, pero no puede editarlos. Un administrador (rol "editor") sí puede. Es el juego de llaves que te dan una vez dentro del edificio: la llave del conserje abre todas las puertas, la tuya solo la de tu apartamento.
  En Web Services, esto se implementa comúnmente con **API Keys** o **Tokens (OAuth 2.0, JWT)** que se envían en cada petición.

➢ Validación de entrada de datos a una aplicación Web

El proceso de asegurar que los datos introducidos en una aplicación cumplen con ciertos criterios.

Es el "control de calidad" y la "aduana" para cualquier dato que intente entrar en tu sistema. Su principio fundamental es **"nunca confíes en la entrada del usuario"**. La validación tiene dos propósitos vitales:

- **Garantizar la Integridad de los Datos:** Asegurarse de que los datos tienen el formato y el sentido correctos antes de guardarlos. Un email debe parecer un email, una edad debe ser un número positivo, un campo obligatorio no puede estar vacío. Esto evita que tu base de datos se llene de "basura".
- **Prevenir Ataques de Seguridad:** Es la primera línea de defensa contra ataques como la **Inyección SQL** o el **Cross-Site Scripting (XSS)**. Al "sanitizar" la entrada (limpiar o rechazar cualquier dato que parezca código malicioso), se evita que un atacante pueda engañar a tu aplicación para que ejecute comandos no deseados.
  Crucial: La validación *siempre* debe hacerse en el **lado del servidor**, aunque también se haga en el cliente (frontend) para mejorar la experiencia de usuario.

➢ Posicionamiento de una aplicación Web

Search Engine Optimization (SEO), el proceso de mejorar la visibilidad de un sitio web en los resultados de los motores de búsqueda.

Es el arte y la ciencia de "hablar el idioma de los motores de búsqueda" para que entiendan de qué trata tu web y la consideren relevante. El objetivo es aparecer lo más arriba posible en los resultados de Google (y otros) para ciertas palabras clave. No es un truco, sino un conjunto de buenas prácticas que se dividen en dos áreas principales:

- **SEO On-Page:** Todo lo que puedes hacer *dentro* de tu propia página: usar HTML semánticamente correcto (`<h1>` para títulos, no para estilo), tener contenido de

calidad y original, URLs limpias, buena velocidad de carga, y ser adaptable a móviles.

- **SEO Off-Page:** Todo lo que ocurre *fuera* de tu página y que le da autoridad. El factor más importante son los **backlinks**: que otras webs de prestigio enlacen a la tuya. Para Google, un enlace es un "voto de confianza".

➢ Historia, situación actual y evolución del diseño de aplicaciones Web

Es la crónica de la transición de la web desde un "libro digital" a una "plataforma de computación" global.

- **Web 1.0 (La Web Estática - Años 90):** La era del "solo lectura". Las páginas eran documentos HTML estáticos, como folletos digitales. La interacción era mínima (hacer clic en enlaces). El desarrollo se centraba en el servidor, que simplemente servía archivos.
- **Web 2.0 (La Web Social y Dinámica - Años 2000):** La era de la "lectura/escritura". Nacen los blogs, las wikis, las redes sociales. Las aplicaciones se vuelven dinámicas gracias a tecnologías de servidor (PHP, ASP) y la llegada de **AJAX**, que permite la interactividad sin recargar la página. El usuario pasa de ser un consumidor a ser un **creador de contenido**.
- **Web 3.0 / Web Moderna (La Web Semántica, Descentralizada y de Aplicaciones - Años 2010 en adelante):** La era de las **Aplicaciones Ricas de Internet (RIA)**. La lógica se traslada masivamente al cliente con frameworks de JavaScript potentes (Angular, React, Vue), que hablan con el servidor a través de APIs (Microservicios). La experiencia es similar a la de una aplicación de escritorio. Surgen conceptos como la web semántica (datos enlazados), la inteligencia artificial y la descentralización (blockchain). El móvil se convierte en el principal punto de acceso.

➢ Filosofías de desarrollo del software

Un enfoque o metodología para estructurar, planificar y controlar el proceso de desarrollo de sistemas de información.

Son los "sistemas operativos" para el trabajo en equipo humano en la creación de software. No son sobre código, son sobre personas, procesos y valores. Responden a la pregunta: "¿Cuál es la forma más efectiva de que un grupo de personas colabore para construir algo complejo y que cambia constantemente?".

- **Cascada (Waterfall):** La filosofía del "planifícalo todo por adelantado". Es un proceso lineal y rígido: análisis, diseño, construcción, pruebas. Funciona bien para proyectos con requisitos muy claros y estables (como construir un puente), pero es desastroso para el software, donde el cambio es constante.
- **Ágil (Agile):** La filosofía de "abrazar el cambio y entregar valor a menudo". En lugar de un gran plan, se trabaja en ciclos cortos (sprints) de 2-4 semanas, al final

de los cuales se entrega una pequeña pieza de software funcional. Valora la colaboración con el cliente, la adaptación continua y la comunicación cara a cara. **Scrum** y **Kanban** son los "sabores" o implementaciones más populares de esta filosofía.

- **DevOps:** La filosofía de "romper los muros". Es una evolución de Agile que busca integrar a los equipos de desarrollo (Dev) y operaciones (Ops). Su objetivo es **automatizar** todo el proceso de entrega de software (integración, pruebas, despliegue) para poder lanzar nuevas versiones de forma muy rápida y fiable (incluso varias veces al día).