Introducción a las preguntas teóricas:

Como sabemos, en la actualidad existen grandes modelos de inteligencia artificial con la capacidad de responder la mayoría de preguntas que podamos realizar como reclutadores. Si bien las respuestas van a ser evaluadas, la finalidad de la parte teórica es ser una guía para que el postulante pueda entender la orientación de la posición y afianzarse con los conocimientos necesarios para su desarrollo laboral.

Posteriormente a la evaluación del trabajo práctico, podrá existir una instancia de conversación donde validemos los conocimientos y el entendimiento de los conceptos.

Link al Repositorio de Github con la Resolución

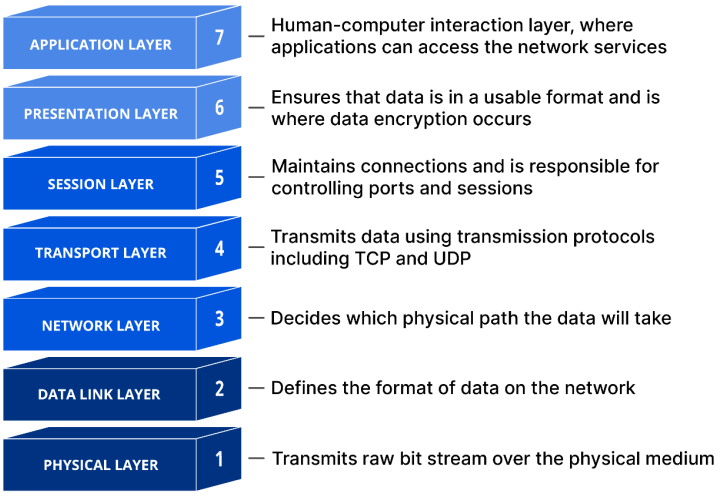
https://github.com/LuisDiNicco/TP-PHP-con-Laravel---Di-Nicco-Luis-Demetrio

Teoría:

Preguntas generales sobre HTTP/HTTPS:

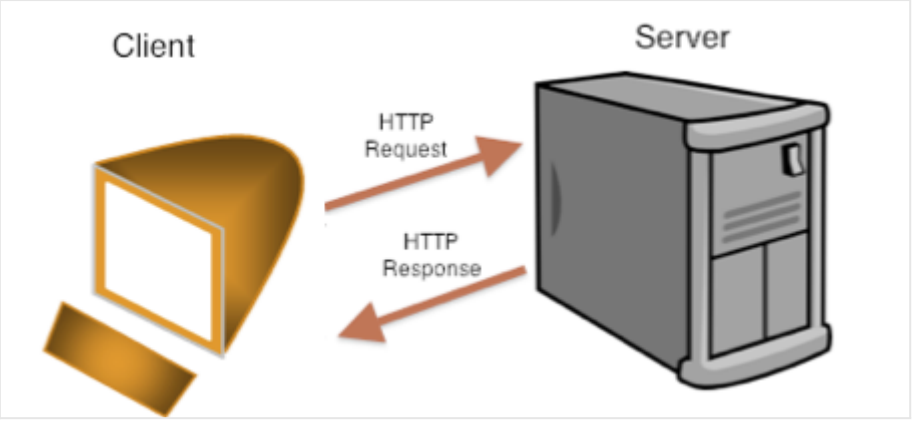
1. ¿Qué es HTTP y cuál es su función principal?

HTTP (HyperText Transfer Protocol o Protocolo de Transferencia de Hipertexto) es un protocolo de comunicación que se usa en la web para el intercambio de información entre clientes (normalmente navegadores) y servidores. Pertenece a la capa de aplicación dentro del modelo OSI y se apoya en protocolos de transporte como TCP/IP para asegurar la entrega confiable de los datos.



En redes, un protocolo es un conjunto de reglas para formatear y procesar datos. Los protocolos de red son como un lenguaje común para las computadoras. Las computadoras dentro de una red pueden usar software y hardware muy diferentes; sin embargo, el uso de protocolos les permite comunicarse entre sí.

Su funcionamiento sigue un modelo de solicitud-respuesta: el cliente envía una petición HTTP al servidor, y este responde con un mensaje HTTP que puede incluir documentos HTML, imágenes, archivos JSON, videos u otros tipos de contenido. Por eso también se habla de una relación cliente-servidor, donde el navegador solo envía solicitudes y el servidor se encarga de responder.



Una característica central de HTTP es que es un protocolo sin estado (stateless): cada petición es independiente y el servidor no guarda información de interacciones anteriores. Si se necesita mantener cierta “memoria” entre las peticiones (por ejemplo, una sesión de usuario), se recurre a mecanismos como cookies, sesiones o tokens.

HTTP es un protocolo basado en texto, lo que significa que tanto las solicitudes como las respuestas se estructuran de manera legible para humanos. Es además extensible, ya que no está limitado a transmitir únicamente páginas HTML: también se usa para APIs REST (con JSON), para la transferencia de archivos multimedia y para múltiples aplicaciones dentro de Internet.

Para organizar la comunicación, HTTP define distintos métodos (o verbos) como:

- **GET**: obtener información de un recurso.
- **POST**: enviar datos al servidor, por ejemplo al completar un formulario.
- **PUT**: actualizar información existente.
- **DELETE**: eliminar un recurso.

A su vez, las respuestas del servidor incluyen códigos de estado que indican el resultado de la petición. Algunos ejemplos comunes son:

- **200 (OK)**: la solicitud se procesó correctamente.
- **400 (Bad Request)**: hubo un error en la solicitud del cliente.
- **404 (Not Found)**: el recurso solicitado no existe en el servidor.

La función principal de HTTP es definir cómo se envían y reciben los datos en la web. Gracias a este mecanismo estandarizado, la web puede funcionar de manera uniforme entre distintos navegadores, servidores y sistemas, sin importar las diferencias de software o hardware que haya en cada extremo. Permite la comunicación entre cliente y servidor usando un formato estándar (texto plano estructurado).

2. ¿Cuál es la diferencia entre HTTP y HTTPS?

HTTP (HyperText Transfer Protocol) y HTTPS (HyperText Transfer Protocol Secure) son protocolos de comunicación que definen cómo se intercambian datos entre un cliente (por ejemplo, un navegador) y un servidor web. La diferencia fundamental está en la seguridad.

HTTP transmite la información en texto plano, lo que significa que los datos enviados y recibidos pueden ser interceptados o modificados por terceros durante el tránsito. Es un protocolo sin mecanismos de protección propios, por lo que no garantiza ni confidencialidad ni autenticidad de la comunicación.

HTTPS, en cambio, utiliza el mismo protocolo HTTP pero le agrega una capa de seguridad mediante TLS (Transport Layer Security) o su antecesor SSL. Gracias a este cifrado, se asegura que:

- **Confidencialidad**: los datos sólo puedan ser leídos por el cliente y el servidor.
- **Integridad**: la información no pueda ser alterada sin que se detecte.
- **Autenticación**: el servidor se identifica a través de un certificado digital válido.

Además de la seguridad, HTTPS ofrece ventajas prácticas. Permite usar cookies seguras (con atributos como *Secure*, *HttpOnly* y *SameSite*), es requisito para la mayoría de APIs modernas, protege credenciales y datos sensibles como tokens o información personal, y es favorecido por los motores de búsqueda: Google penaliza los sitios que siguen usando HTTP y mejora el posicionamiento de los que usan HTTPS. También brinda mayor confianza al usuario al mostrar el ícono de candado en el navegador.

En cuanto a lo técnico, HTTP usa por defecto el puerto 80, mientras que HTTPS utiliza el 443. Hoy en día prácticamente todos los sitios modernos implementan HTTPS, mientras que HTTP quedó relegado a usos muy puntuales. Incluso, con el soporte de HTTP/2 y HTTP/3 sobre TLS, HTTPS no solo es más seguro sino que también ofrece mejor rendimiento y tiempos de carga más rápidos en comparación con HTTP.

Cuadro Comparativo entre HTTP y HTTPS:

	HTTP	**HTTPS**
Significado	Protocolo de transferencia de hipertexto	Protocolo seguro de transferencia de hipertexto
Protocolos subyacentes	HTTP/1 y HTTP/2 utilizan TCP/IP. HTTP/3 usa el protocolo QUIC.	Utiliza HTTP/2 con SSL/TLS para cifrar aún más las solicitudes y respuestas HTTP
Puerto	Puerto predeterminado 80	Puerto predeterminado 443

Se utiliza para lo siguiente:	Sitios web antiguos basados en texto	Todos los sitios web modernos
Seguridad	Sin funciones de seguridad adicionales. No cifra los datos, viajan en texto plano.	Utiliza certificados SSL para el cifrado de clave pública. Cifra los datos usando TLS/SSL.
Beneficios	Hizo posible la comunicación a través de Internet	Mejora la autoridad del sitio web, la confianza y el posicionamiento en los motores de búsqueda
Privacidad	Cualquiera en la red puede leer la información (contraseñas, cookies, etc.).	La información viaja cifrada, sólo el cliente y servidor pueden leerla.
Integridad	Los datos pueden ser alterados en tránsito.	TLS garantiza que los datos no se modifiquen sin que se detecte.
Autenticidad	No hay validación de la identidad del servidor.	El servidor presenta un certificado digital para probar su identidad.
SEO y Confianza	Google y navegadores penalizan sitios HTTP.	Mejor ranking SEO y el famoso “candadito verde” en el navegador.

3. ¿Cómo funciona el proceso de cifrado en HTTPS?

Cuando un cliente (por ejemplo, un navegador) se conecta a un servidor mediante HTTPS, se establece un canal seguro a través del protocolo TLS (Transport Layer Security), que añade una capa de seguridad sobre HTTP. El proceso que garantiza la seguridad se llama handshake TLS (apretón de manos), y su función es asegurar tres cosas: cifrado, integridad y autenticación de la comunicación.

El handshake TLS se puede simplificar en los siguientes pasos:

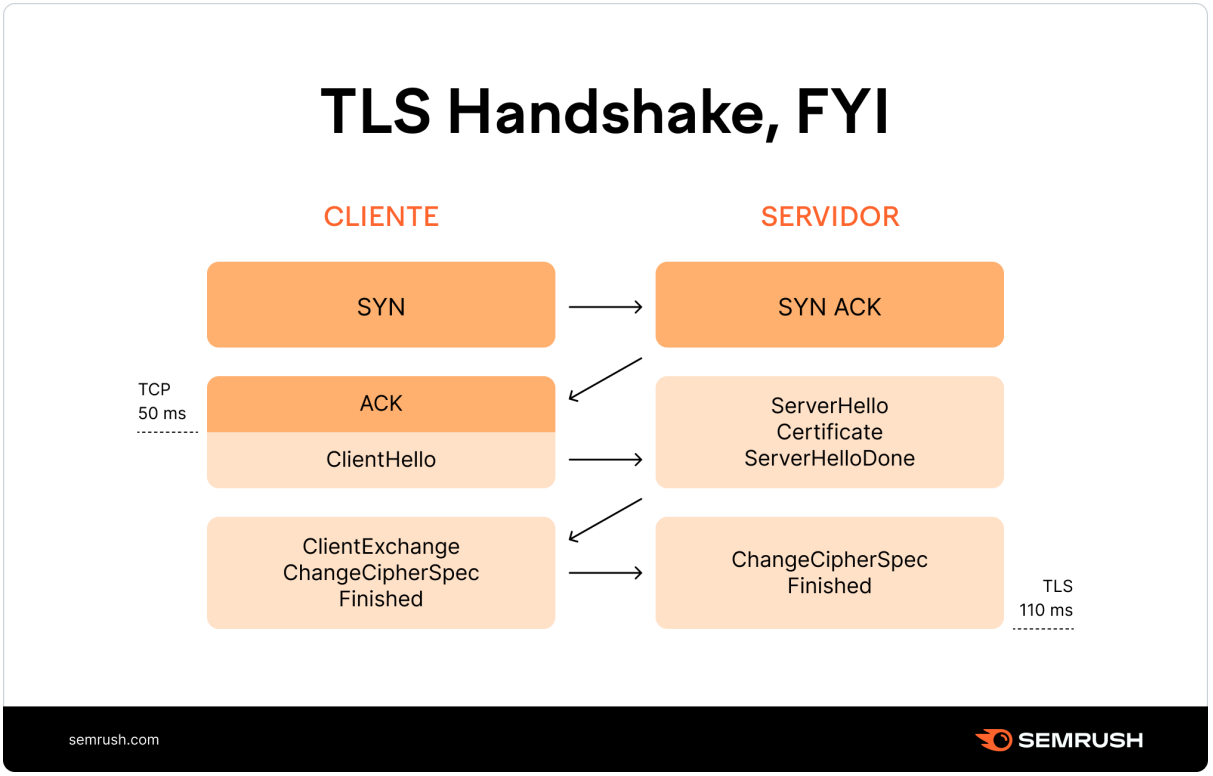
1. **Cliente → Servidor: ClientHello**
   El cliente inicia la conexión enviando:
   - Las versiones de TLS que soporta.
   - Una lista de algoritmos de cifrado posibles.
   - Un número aleatorio (*nonce*).

2. **Servidor → Cliente: ServerHello**
   El servidor responde:
   - Seleccionando la versión de TLS y el algoritmo de cifrado que se usará.
   - Enviando otro número aleatorio.
   - Presentando su **certificado digital**, que contiene su clave pública y está firmado por una Autoridad Certificadora confiable (como DigiCert o Let's Encrypt).

3. **Verificación del certificado**
   El cliente valida que:
   - El certificado haya sido emitido por una CA confiable.
   - No esté expirado ni revocado.
   - El dominio coincida con la URL solicitada.
     Si la verificación falla, el navegador muestra una advertencia de seguridad.

4. **Intercambio de claves**
   Usando criptografía asimétrica (RSA o Diffie-Hellman), el cliente y el servidor acuerdan una clave de sesión simétrica, que luego se usará para cifrar toda la comunicación de la sesión. La criptografía asimétrica es más lenta y se usa solo en este paso inicial, mientras que la simétrica (AES, ChaCha20, etc.) es más rápida y eficiente para cifrar los datos reales.

5. **Comunicación segura**
   A partir de este momento, todos los mensajes viajan cifrados con la clave de sesión, garantizando que la información transmitida:
   - No pueda ser leída por terceros (**confidencialidad**).
   - No pueda ser modificada sin que se detecte (**integridad**).
   - Sea enviada entre el cliente y el servidor legítimo (**autenticación**).



En la práctica, esto significa que cuando, por ejemplo, un usuario envía su contraseña o datos personales, estos viajan cifrados y solo el servidor puede descifrarlos. HTTPS combina de manera eficiente la seguridad de la criptografía asimétrica para el intercambio de claves con la velocidad de la criptografía simétrica para la transmisión de datos, asegurando así una comunicación confiable y segura entre el cliente y el servidor.

La criptografía asimétrica utiliza un par de claves diferentes: una pública y una privada. La clave pública se comparte abiertamente y sirve para cifrar la información, mientras que la clave privada se mantiene secreta y es la única capaz de descifrar lo que fue cifrado con la pública. De esta manera, aunque un atacante intercepte el mensaje cifrado, no podrá leerlo sin la clave privada. Este tipo de cifrado en https se emplea principalmente durante el handshake TLS para autenticar al servidor mediante su certificado digital y para intercambiar de forma segura la clave de sesión que luego usará la criptografía simétrica.

La criptografía simétrica, en cambio, se basa en una sola clave que tanto el cliente como el servidor deben conocer y mantener en secreto. La misma clave sirve para cifrar y descifrar los mensajes, lo que la hace muy rápida y eficiente en comparación con la asimétrica. En HTTPS, una vez que el cliente y el servidor acuerdan esta clave durante el handshake, se utiliza para proteger toda la comunicación posterior (por ejemplo, al enviar contraseñas, mensajes o datos bancarios). Gracias a su velocidad, la criptografía simétrica es ideal para manejar grandes volúmenes de información de forma segura.

4. ¿Qué es un certificado SSL/TLS y cuál es su importancia en HTTPS?

Un certificado SSL/TLS es un archivo digital que autentica la identidad de un sitio web y permite establecer una comunicación cifrada entre un servidor y un cliente (por ejemplo, un

navegador). Estos certificados son emitidos por entidades de confianza conocidas como Autoridades Certificadoras (CA), que validan la identidad del propietario del dominio antes de otorgar el certificado. En la práctica, funcionan como una “cédula de identidad digital” para el sitio web.

Los certificados SSL/TLS cumplen dos funciones principales en HTTPS:

1. **Autenticación:** Garantizan que el sitio web al que se accede es legítimo y no un impostor. Esto se verifica a través del certificado digital, la firma de la autoridad certificadora y la coincidencia del dominio con el certificado.

2. **Cifrado:** Permiten el establecimiento de una conexión segura mediante TLS, asegurando que los datos transmitidos entre el cliente y el servidor no puedan ser leídos ni modificados por terceros.

Los certificados SSL/TLS contienen información clave como:

- Nombre del dominio.
- Autoridad certificadora que emitió el certificado.
- Firma digital de la CA.
- Clave pública del servidor, que se usa en el handshake TLS para generar la clave de sesión simétrica.
- Fechas de emisión y vencimiento.
- Versión de SSL/TLS que se soporta.

El cifrado funciona mediante criptografía de clave pública: el servidor proporciona la clave pública en el certificado, que el cliente utiliza para cifrar información sensible antes de enviarla. Solo el servidor, con su clave privada, puede descifrarla. Esto asegura que los datos viajen confidenciales y autenticados. Además, los certificados incluyen una firma digital, que permite detectar cualquier alteración del certificado mientras se transmite por la red.

La importancia de los certificados SSL/TLS es central en la web moderna:

- Protegen datos sensibles de usuarios, como credenciales o información financiera.
- Refuerzan la confianza del usuario mediante señales visuales como el candado en la barra de direcciones y el prefijo HTTPS.
- Son requeridos para cumplir normativas de seguridad, como PCI DSS en transacciones de tarjetas de pago.
- Mejoran el SEO, ya que los motores de búsqueda priorizan los sitios que implementan HTTPS correctamente.

Aunque SSL es el protocolo original, los certificados modernos utilizan principalmente TLS, que es su sucesor, más seguro y eficiente. Sin un certificado SSL/TLS válido, HTTPS no puede garantizar ni seguridad ni autenticidad, y los navegadores suelen advertir a los usuarios sobre conexiones no seguras.

5. ¿Qué es un método HTTP? ¿Podrías enumerar algunos de los más utilizados?

Un método HTTP, también llamado verbo HTTP, es la acción que un cliente (como un navegador o una aplicación) le solicita a un servidor que realice sobre un recurso. Cada método indica la intención de la solicitud, por ejemplo, si se quiere obtener información, enviar datos, modificar o eliminar un recurso. Son la base de la comunicación en la web y en el desarrollo de APIs RESTful, ya que permiten implementar operaciones equivalentes a las del patrón CRUD (Create, Read, Update, Delete).

Los métodos HTTP son fundamentales para la arquitectura de la web y la implementación de APIs, ya que permiten comunicar de forma clara y estandarizada la intención de cada solicitud entre cliente y servidor.

Algunos de los métodos HTTP más utilizados son:

- **GET**: Solicita un recurso del servidor, como una página web o datos de una API. Es seguro y no modifica el estado del servidor; puede repetirse sin efectos secundarios.

- **POST**: Envía datos al servidor para crear un nuevo recurso, como al completar un formulario. Puede modificar el estado del servidor y no es idempotente, es decir, repetirlo varias veces puede crear múltiples recursos.

- **PUT**: Actualiza completamente un recurso existente o lo crea si no existe en la ubicación indicada. Es idempotente, así que repetir la solicitud produce el mismo resultado final.

- **PATCH**: Modifica parcialmente un recurso, enviando solo los cambios necesarios en lugar de reemplazar todo el contenido. Esto optimiza el ancho de banda y es útil para actualizaciones parciales.

- **DELETE**: Elimina un recurso específico del servidor. También es idempotente.

- **HEAD**: Similar a GET, pero solo solicita las cabeceras de la respuesta, sin el cuerpo. Se usa para verificar información sobre un recurso sin descargarlo.

- **OPTIONS**: Permite consultar qué métodos y opciones de comunicación soporta un recurso o servidor.

- **CONNECT**: Establece un túnel hacia el servidor indicado, por ejemplo para conexiones seguras a través de un proxy.

- **TRACE**: Realiza una prueba de bucle de retorno, mostrando cómo se ha procesado la solicitud a lo largo del camino hacia el recurso.

Estos métodos no solo definen la operación a realizar, sino que también pueden clasificarse según ciertas propiedades:

- **Safe (seguro)**: No modifica recursos, como GET y HEAD.
- **Idempotent (idempotente)**: Repetir la misma solicitud produce el mismo resultado, como PUT y DELETE.

- **Cacheable (cacheable)**: Permite almacenar respuestas en caché, como GET y HEAD.

6. Explica las diferencias entre los métodos HTTP GET y POST.

Los métodos HTTP GET y POST se diferencian principalmente en su propósito y en cómo transmiten los datos entre cliente y servidor. GET se emplea principalmente para consultar recursos y transmitir parámetros sencillos, mientras que POST se utiliza para enviar información más compleja y generar cambios en el servidor, como la creación de nuevos recursos o la actualización de datos.

- GET se utiliza para solicitar información de un recurso en el servidor, funcionando como una operación de lectura. Los datos opcionales se envían en la URL, a través de la cadena de consulta (query string), por ejemplo: www.ejemplo.com/productos?id=10. Esto hace que los parámetros sean visibles en la barra de direcciones, en el historial del navegador y en los registros del servidor, por lo que no se recomienda para información sensible. GET es idempotente, lo que significa que repetir la misma solicitud no cambia el estado del recurso en el servidor, y sus respuestas pueden ser cacheadas, permitiendo que navegadores o proxys guarden la información para mejorar el rendimiento. El tamaño de los datos está limitado por la longitud máxima de la URL.

- POST, en cambio, se utiliza para enviar datos al servidor, generalmente con el objetivo de crear un recurso o procesar información. Los datos viajan en el cuerpo de la solicitud (request body), lo que permite enviar información más extensa y estructurada. No es idempotente, ya que repetir la misma solicitud puede generar múltiples registros o efectos distintos en el servidor. Las solicitudes POST normalmente no se cachean, y aunque los datos no son visibles en la URL, requieren HTTPS para garantizar la seguridad de la información transmitida.

Cuadro comparativo entre GET y POST:

Aspecto	GET	POST
Propósito	Obtener datos	Enviar datos (crear o procesar)
Dónde viajan los datos	En la URL (?param=valor)	En el cuerpo de la petición

Tamaño de datos	Limitado por la longitud de la URL	Mucho más grande (dependiendo del servidor)
Idempotencia	Sí (repetirlo no cambia nada)	No (repetirlo puede generar efectos distintos)
Seguridad	Menos seguro (los parámetros son visibles en la URL y logs)	Más seguro (datos no visibles en la URL, pero igualmente requieren HTTPS)
Cache	Puede ser cacheado por el navegador y proxys	Generalmente no se cachea

7. ¿Qué es un código de estado HTTP? ¿Podrías mencionar algunos de los más comunes y lo que significan?

Un código de estado HTTP es un número de tres dígitos que el servidor devuelve como parte de la respuesta a una solicitud HTTP, indicando el resultado de la operación. Estos códigos permiten que el cliente (como un navegador o una aplicación) entienda si la solicitud fue exitosa, si hubo un error, o si se requiere alguna acción adicional. Son esenciales para la comunicación cliente-servidor, ya que estandarizan la forma en que se informa el resultado de cada solicitud y facilitan el diagnóstico de problemas en aplicaciones web y APIs.

Los códigos de estado se agrupan en cinco categorías principales según el primer dígito:

- **1xx (Informativos):** Indican que la solicitud se recibió y que el proceso continúa.
  2xx (Éxito): Señalan que la operación se completó correctamente.
- **3xx (Redirecciones):** Indican que el cliente debe realizar otra acción para completar la solicitud, como seguir una nueva URL.
- **4xx (Errores del cliente):** Señalan que hubo un problema en la solicitud enviada por el cliente, como errores de sintaxis o falta de permisos.
- **5xx (Errores del servidor):** Indican que el servidor falló al procesar una solicitud válida.

Algunos códigos de estado HTTP más comunes son:

- **200 OK:** La solicitud fue exitosa y se devolvió el recurso solicitado.
- **201 Created:** La solicitud se completó correctamente y se creó un nuevo recurso (por ejemplo, después de un POST).
- **301 Moved Permanently:** El recurso solicitado se movió a otra URL de forma permanente.

- **302 Found:** Redirección temporal a otra URL.
- **400 Bad Request:** La solicitud tiene errores de sintaxis o parámetros inválidos.
- **401 Unauthorized:** Se requiere autenticación o las credenciales son inválidas.
- **403 Forbidden:** El cliente no tiene permisos para acceder al recurso.
- **404 Not Found:** El recurso solicitado no existe en el servidor.
- **500 Internal Server Error:** Error genérico del servidor que impidió completar la solicitud.
- **503 Service Unavailable:** El servidor no está disponible temporalmente, normalmente por mantenimiento o sobrecarga.

8. ¿Qué es una cabecera HTTP? Da ejemplos de cabeceras comunes.

Una cabecera HTTP (HTTP header) es un campo dentro de una solicitud o respuesta HTTP que transmite información adicional sobre la comunicación entre cliente y servidor. No forma parte del cuerpo del mensaje (los datos reales), sino que proporciona metadatos que pueden incluir instrucciones, información técnica, preferencias del cliente o del servidor, y directivas de control como caché, autenticación o tipo de contenido.

Las cabeceras se pueden clasificar en cuatro tipos principales:

- **Cabeceras de solicitud (Request Headers):** Enviadas por el cliente al servidor para proporcionar información adicional sobre la petición.
  Ejemplos comunes:
  - **Host:** indica el dominio del servidor al que se envía la solicitud.
  - **User-Agent:** identifica el navegador o aplicación cliente.
  - **Accept:** especifica los tipos de contenido que el cliente puede procesar (por ejemplo, application/json).
  - **Authorization:** contiene credenciales de autenticación.
  - **Cookie:** envía información de sesión previamente guardada por el servidor.

- **Cabeceras de respuesta (Response Headers):** Enviadas por el servidor al cliente para informar sobre la respuesta y el contenido.
  Ejemplos:
  - **Content-Type:** define el tipo de contenido devuelto (por ejemplo, text/html o application/json).
  - **Content-Length:** indica el tamaño del cuerpo de la respuesta.
  - **Set-Cookie:** envía cookies para que el cliente las almacene.
  - **Cache-Control:** establece directivas de almacenamiento en caché.
  - **Location:** indica la nueva URL en caso de redirección.
  - **Server:** informa sobre el software del servidor que procesa la solicitud.

- **Cabeceras generales (General Headers):** Aplican tanto a solicitudes como a respuestas y suelen incluir información de control de la comunicación, como Connection o Date.

- **Cabeceras de entidad (Entity Headers):** Describen el cuerpo del mensaje, incluyendo detalles como tipo, tamaño o codificación del contenido (Content-Type,

Content-Encoding, Content-Length).

Las cabeceras HTTP son esenciales para enriquecer la comunicación cliente-servidor con información contextual y técnica, permitiendo que navegadores, servidores y APIs manejen correctamente la transferencia de datos, la autenticación, la caché y otros aspectos fundamentales de la web moderna.

9. ¿En qué consiste el concepto de "idempotencia" en los métodos HTTP? ¿Qué métodos cumplen con esta característica?

En HTTP, la idempotencia es la propiedad de un método según el cual ejecutar la misma operación varias veces produce siempre el mismo resultado en el servidor, sin generar efectos secundarios adicionales después de la primera ejecución. Esto significa que repetir una solicitud idempotente no altera el estado del recurso más allá de lo que ocurrió en la primera llamada, lo cual es fundamental para la confiabilidad y estabilidad de las aplicaciones web, ya que permite reintentos seguros ante fallos de red o errores temporales.

Los métodos HTTP idempotentes son:

- **GET**: Solicitar un recurso no modifica su estado; pedirlo varias veces devuelve la misma información sin efectos adicionales.
- **HEAD**: Similar a GET, pero sólo devuelve las cabeceras de la respuesta; no altera el recurso.
- **PUT**: Reemplaza un recurso existente; enviar la misma solicitud varias veces deja el recurso en el mismo estado que tras la primera actualización.
- **DELETE**: Eliminar un recurso es idempotente porque, después de la primera eliminación, intentarlo nuevamente no cambia nada adicional: el recurso ya no existe.
- **OPTIONS**: Consultar las opciones o métodos soportados por un recurso no modifica su estado.

En contraste, **POST** no es idempotente, ya que cada ejecución puede generar efectos diferentes, como crear múltiples recursos o enviar datos que alteran el estado del servidor. Por ejemplo, enviar dos veces un formulario de registro podría crear dos cuentas de usuario en lugar de una.

La idempotencia garantiza que operaciones de lectura, actualización o eliminación puedan repetirse de forma segura sin riesgos de duplicar cambios o alterar el estado de los recursos de manera inesperada.

10. ¿Qué es un redirect (redirección) HTTP y cuándo es utilizado?

Un redirect (redirección) HTTP es una respuesta que un servidor envía al cliente para indicarle que el recurso solicitado se encuentra en otra URL y que debe realizar una nueva solicitud en esa dirección. La redirección se comunica mediante un código de estado HTTP

de la familia 3xx y la cabecera Location, que especifica la nueva URL a la que el cliente debe dirigirse. Las redirecciones son una herramienta vital en el desarrollo web y el SEO para asegurar una buena experiencia de usuario y mantener la integridad de un sitio.

Se utiliza en varias situaciones:

- **Cambio permanente de URL:** cuando una página o recurso se traslada definitivamente a otra dirección, se emplea un 301 Moved Permanently. Los navegadores y motores de búsqueda actualizan sus índices y la autoridad SEO se transfiere a la nueva URL.
- **Redirección temporal:** si el recurso está temporalmente en otra URL, se usan códigos como 302 Found o 307 Temporary Redirect, sin afectar el SEO ni actualizar los enlaces antiguos en caché.
- **Migración o reestructuración de un sitio web:** asegura que los usuarios y motores de búsqueda lleguen a la nueva ubicación de los recursos después de cambios en dominios o rutas de URLs.
- **Redirección de HTTP a HTTPS:** obliga a que todo el tráfico utilice conexiones seguras cifradas mediante TLS.
- **Balanceo de carga o mantenimiento:** dirige a los usuarios hacia servidores alternativos o páginas de mantenimiento temporal.
- **Flujos de autenticación:** después de iniciar sesión, redirigir al usuario a la página principal o dashboard.
- **Redirección de www a sin www (o viceversa):** para evitar contenido duplicado en buscadores y mantener coherencia de dominio.
- **Corrección de enlaces rotos:** si una página ha sido eliminada o movida, se puede redirigir su URL a una página relevante para evitar el error 404 "Not Found".
- **Consolidación de contenido:** cuando se fusionan varias páginas de un sitio en una sola, se usan redirecciones para dirigir el tráfico de las páginas antiguas a la nueva.

Cuando el navegador recibe la redirección, automáticamente solicita la URL indicada en la cabecera Location, permitiendo que el usuario acceda al recurso correcto sin intervención manual.

Las redirecciones se clasifican principalmente en dos tipos, según el código de estado HTTP que se utiliza:

- **Redirección Permanente (301 Moved Permanently):** Indica que el recurso ha sido movido a una nueva URL de forma permanente. Los navegadores y los motores de búsqueda, como Google, almacenan en caché esta redirección y actualizan sus índices. Es la opción preferida para cambios definitivos, ya que transfiere la autoridad de SEO de la URL antigua a la nueva.
- **Redirección Temporal (302 Found):** Indica que el recurso ha sido movido temporalmente. El navegador no almacena en caché la redirección y debe seguir solicitando la URL original en el futuro. Es útil para situaciones como mantenimiento del sitio, pruebas A/B o redirecciones por tiempo limitado, ya que no afecta al SEO de la URL de origen.

<u>Códigos de Redirección:</u>

Código	Nombre	Uso principal
301 Moved Permanently	Redirección permanente	El recurso se trasladó definitivamente a otra URL. Los motores de búsqueda actualizan sus índices.
302 Found	Redirección temporal	El recurso se encuentra temporalmente en otra URL. Los motores de búsqueda no cambian el índice.
303 See Other	Ver otro recurso	Después de un POST, se redirige a otra URL usando GET (ej: página de confirmación).
307 Temporary Redirect	Redirección temporal	Similar a 302, pero obliga a usar el mismo método HTTP en la nueva URL.
308 Permanent Redirect	Redirección permanente	Similar a 301, pero asegura que se mantenga el método HTTP original.

Preguntas técnicas y de seguridad en HTTP/HTTPS:

11. ¿Cómo se asegura la integridad de los datos en una conexión HTTPS?

La integridad de los datos en una conexión HTTPS se garantiza gracias al protocolo TLS (Transport Layer Security), que combina cifrado y verificación de mensajes para asegurar que la información transmitida no sea alterada durante la comunicación. Esto significa que cualquier modificación, ya sea accidental o maliciosa, puede ser detectada.

Cuando se establece una conexión HTTPS, el cliente y el servidor realizan un *handshake* para acordar una clave de sesión simétrica. A partir de esa clave, cada mensaje que se envía incluye un código de autenticación de mensajes (MAC), que se genera usando una función hash criptográfica, como SHA-256, aplicada a los datos originales junto con la clave de sesión.

El proceso de verificación funciona de la siguiente manera:

- El mensaje se cifra usando la clave de sesión y se envía junto con su MAC.
- El receptor descifra el mensaje con la misma clave.
- Calcula nuevamente el MAC de los datos recibidos.
- Compara el MAC calculado con el que venía en el mensaje. Si coinciden, se confirma que los datos no fueron alterados. Si no coinciden, se detecta manipulación y la conexión puede cerrarse por seguridad.

Este mecanismo se aplica a cualquier tipo de dato enviado por HTTPS, ya sean páginas web, formularios o respuestas de APIs, garantizando que la información llegue exactamente como fue enviada. De esta forma, HTTPS protege la integridad de los datos junto con la confidencialidad que ofrece el cifrado simétrico de la sesión.

12. ¿Qué diferencia hay entre un ataque de "man-in-the-middle" y un ataque de "replay" en un contexto HTTPs?

En un contexto HTTPS, los ataques de *man-in-the-middle* (MITM) y *replay* son amenazas distintas que buscan explotar la comunicación entre cliente y servidor, pero operan de formas diferentes. MITM busca interceptar o modificar la comunicación, mientras que un ataque de replay busca reutilizar mensajes válidos para generar efectos no deseados. HTTPS, con TLS, protege la comunicación contra ambos mediante cifrado, autenticación y mecanismos de integridad.

- Un ataque de *man-in-the-middle* ocurre cuando un atacante se interpone secretamente entre el cliente y el servidor, actuando como un intermediario que puede interceptar, leer e incluso modificar la información en tiempo real. En HTTP sin cifrado, esto es relativamente fácil porque los datos viajan en texto plano. Con HTTPS, la comunicación está cifrada mediante TLS, lo que dificulta que el atacante lea o altere los datos. Sin embargo, si el atacante logra engañar al cliente con un certificado falso, podría interceptar la información. La mitigación principal en HTTPS es la correcta validación del certificado digital del servidor: si el navegador detecta que el certificado no es válido o no proviene de una autoridad de confianza, emite una advertencia que previene el ataque.

- Por otro lado, un ataque de *replay* consiste en capturar mensajes legítimos enviados por el cliente y volver a enviarlos posteriormente para generar efectos no deseados, como repetir transacciones. En este caso, el atacante no necesita descifrar el contenido, sino simplemente reutilizar los datos. HTTPS protege contra este tipo de ataque utilizando mecanismos como *nonces* (números aleatorios de un solo uso) o marcas de tiempo durante el handshake TLS y en cada mensaje de la sesión, asegurando que cada solicitud sea única y detectando intentos de repetición.

La diferencia principal entre ambos ataques es que el MITM implica una intervención activa y en tiempo real para leer o modificar los datos, mientras que el ataque de replay es pasivo y busca reutilizar mensajes válidos para provocar acciones no autorizadas.

13. Explica el concepto de "handshake" en HTTPS.

En HTTPS, el *handshake* es el proceso inicial mediante el cual el cliente y el servidor establecen una conexión segura usando TLS (Transport Layer Security). Su objetivo es garantizar la confidencialidad, integridad y autenticación de los datos que se van a transmitir. Durante este proceso se cumplen tres funciones principales: autenticar al servidor, negociar los algoritmos de cifrado y generar una clave de sesión simétrica para cifrar la comunicación posterior.

El *handshake* se puede describir en los siguientes pasos, considerando la versión moderna TLS 1.3:

1. **Hola del cliente (*Client Hello*)**: El cliente envía un mensaje al servidor con la versión de TLS que soporta, la lista de suites de cifrado disponibles y un número aleatorio (nonce) que servirá luego para generar la clave de sesión. También incluye los parámetros necesarios para el intercambio de claves.

2. **Hola del servidor y envío de certificado (*Server Hello*)**: El servidor responde seleccionando una suite de cifrado de la lista del cliente y enviando su propio

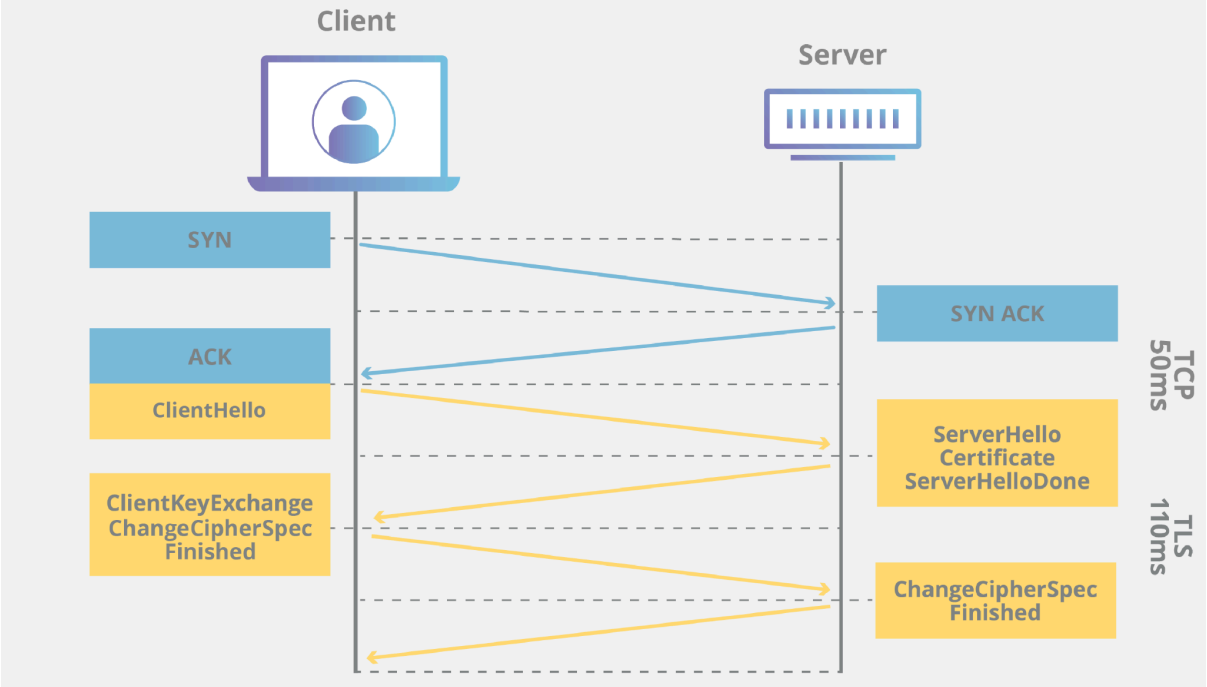
número aleatorio. Además, envía su certificado digital, que contiene la clave pública y está firmado por una Autoridad de Certificación (CA) de confianza.

3. **Verificación del certificado**: El cliente valida la autenticidad del certificado, comprobando la firma de la CA y que el certificado no haya caducado. Si la verificación falla, el navegador muestra una advertencia de seguridad.

4. **Generación del secreto maestro y clave de sesión**: El cliente utiliza la clave pública del servidor para cifrar un secreto aleatorio (*pre-master secret*). Este secreto, junto con los números aleatorios de cliente y servidor, se usa para generar la clave de sesión simétrica. El servidor, con su clave privada, descifra el secreto y ambos extremos ya comparten la misma clave de sesión.

5. **Mensajes de "Finalizado"**: Tanto cliente como servidor envían un mensaje de "Finalizado" cifrado con la clave de sesión para confirmar que la conexión segura está establecida.

A partir de este momento, toda la comunicación entre cliente y servidor se realiza mediante cifrado simétrico, lo que asegura confidencialidad, integridad y autenticación de los datos, y permite que la comunicación sea rápida y segura.



14. ¿Qué es HSTS (HTTP Strict Transport Security) y cómo mejora la seguridad de una aplicación web?

HSTS (HTTP Strict Transport Security) es un mecanismo de seguridad que permite a un servidor web indicarle al navegador que todas las conexiones futuras a su dominio deben realizarse exclusivamente mediante HTTPS, evitando cualquier comunicación por HTTP no seguro. Esto se logra mediante la cabecera Strict-Transport-Security, que puede incluir

parámetros como la duración de la política, la aplicación a subdominios y la inclusión en listas de precarga de los navegadores.

El funcionamiento básico de HSTS es el siguiente: cuando un navegador recibe la cabecera HSTS en una conexión segura, almacena la política y la aplica a todas las futuras solicitudes al dominio durante el período definido. Así, incluso si un usuario escribe http:// en la barra de direcciones, el navegador fuerza automáticamente el uso de HTTPS, evitando la vulnerabilidad de la primera solicitud HTTP.

Entre los beneficios de HSTS se encuentran:

- **Prevención de ataques de degradación o downgrade**: impide que un atacante obligue al navegador a usar HTTP en lugar de HTTPS.
- **Protección contra ataques MITM**: garantiza que todo el tráfico siempre esté cifrado, evitando que un atacante pueda interceptar o modificar los datos.
- **Reducción de errores por enlaces inseguros**: el navegador redirige automáticamente cualquier intento de conexión por HTTP a HTTPS, incluso sin intervención del usuario.
- **Mejora de la confianza del usuario:** reduce advertencias de seguridad y asegura que los datos sensibles estén protegidos.

Un ejemplo de cabecera HSTS sería:
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload

- **max-age**=31536000: la política se recuerda por un año.
- **includeSubDomains**: aplica la política a todos los subdominios.
- **preload**: indica que el dominio puede incluirse en la lista de precarga de los navegadores, forzando HTTPS incluso en la primera visita.

HSTS refuerza de manera transparente la seguridad de una aplicación web, asegurando que toda la comunicación entre cliente y servidor sea cifrada y confiable.

15. ¿Qué es un ataque "downgrade" y cómo HTTPS lo previene?

Un ataque “downgrade” es un intento de un atacante para obligar a un cliente y a un servidor a comunicarse utilizando versiones antiguas o menos seguras de un protocolo o algoritmos de cifrado. En el contexto de HTTPS, esto puede significar forzar la conexión a usar HTTP no cifrado o protocolos TLS obsoletos como SSLv3 o TLS 1.0. El objetivo del atacante es aprovechar vulnerabilidades de estas versiones antiguas para interceptar, modificar o espiar la comunicación, y muchas veces es un paso previo a un ataque man-in-the-middle (MITM).

El mecanismo típico de un ataque downgrade se basa en la vulnerabilidad de la primera solicitud HTTP. Por ejemplo, cuando un usuario escribe ejemplo.com, el navegador normalmente hace una solicitud inicial por HTTP y el servidor responde con una redirección a HTTPS. Un atacante que se interpone en la comunicación puede bloquear esa redirección o responder con HTTP, manteniendo la conexión en un canal inseguro y permitiendo leer o modificar los datos transmitidos.

HTTPS moderno previene los ataques *downgrade* mediante varias medidas:

- **Negociación segura de versión y cifrado**: durante el *handshake* TLS, cliente y servidor acuerdan usar la versión más moderna y segura del protocolo y algoritmos de cifrado robustos. Las versiones inseguras se descartan automáticamente.
- **Integridad y autenticación**: el certificado digital del servidor garantiza que el cliente se conecta al servidor legítimo y detecta manipulaciones en la negociación del protocolo.
- **HSTS (HTTP Strict Transport Security)**: fuerza al navegador a usar HTTPS siempre, evitando que un atacante pueda redirigir al cliente a HTTP vulnerable. Si el dominio está incluido en la lista de precarga de HSTS de los navegadores (*HSTS Preload List*), incluso la primera visita se realiza directamente por HTTPS, eliminando la vulnerabilidad inicial.

En conjunto, estas medidas aseguran que la comunicación entre cliente y servidor utilice siempre protocolos modernos y cifrados robustos, bloqueando intentos de degradación que podrían comprometer la seguridad de los datos.

16. ¿Qué es el CORS (Cross-Origin Resource Sharing) y cómo se implementa en una aplicación web?

CORS (Cross-Origin Resource Sharing) es un mecanismo de seguridad que permite a un servidor web controlar qué orígenes externos pueden acceder a sus recursos mediante solicitudes HTTP desde un navegador. Esto surge porque, por defecto, los navegadores aplican la política de mismo origen (Same-Origin Policy), que restringe las solicitudes a recursos del mismo dominio, protocolo y puerto, evitando que scripts maliciosos accedan a datos de otros sitios. CORS se implementa mediante cabeceras HTTP que el servidor envía en la respuesta, indicando los permisos para los orígenes externos.

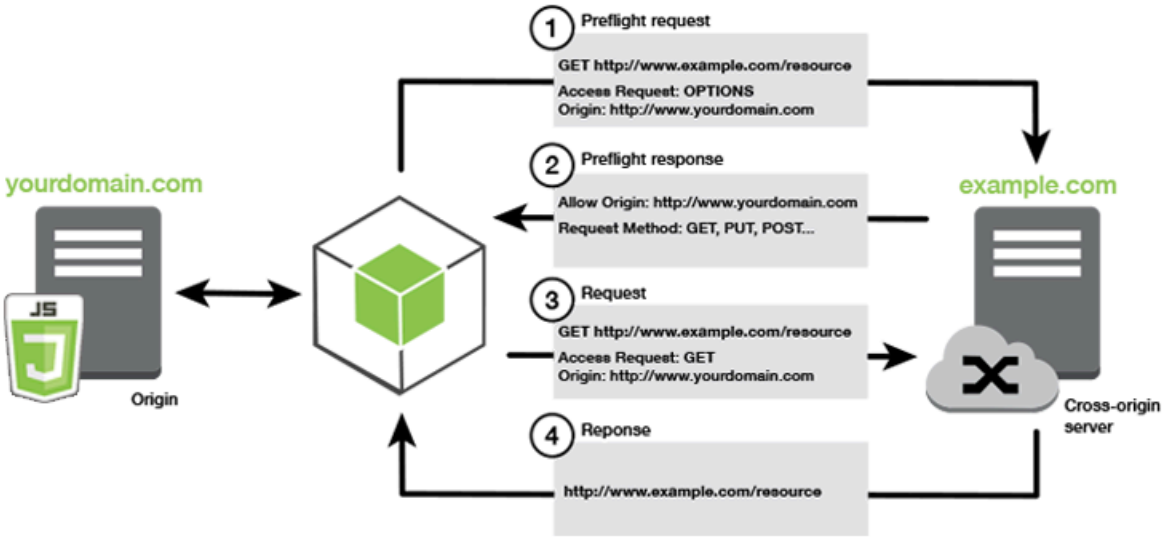
CORS es un mecanismo para integrar aplicaciones. Define una forma con la que las aplicaciones web clientes cargadas en un dominio pueden interactuar con los recursos de un dominio distinto. Esto resulta útil porque las aplicaciones complejas suelen hacer referencia a API y recursos de terceros en el código del cliente. Es una ampliación de la política del mismo origen. Lo necesita para compartir recursos autorizados con terceros externos.

Funcionamiento de CORS en una aplicación web

1. **Solicitud del cliente:** Un navegador desde un origen A (por ejemplo, https://mi-tienda.com) intenta acceder a recursos de un servidor en un origen B (https://api.mi-tienda.com). El navegador agrega automáticamente la cabecera Origin con el dominio de la página que realiza la solicitud.
2. **Preflight request (solicitud previa):** Si la solicitud es “compleja” (por ejemplo, métodos PUT o DELETE, o cabeceras personalizadas), el navegador primero envía una solicitud OPTIONS al servidor para verificar si está permitido.
3. **Respuesta del servidor:** El servidor decide si autoriza la solicitud y responde con cabeceras CORS que indican los permisos:

- *Access-Control-Allow-Origin:* dominios permitidos (puede ser un dominio específico o * para todos).
- *Access-Control-Allow-Methods:* métodos HTTP permitidos (GET, POST, PUT, etc.).
- *Access-Control-Allow-Headers:* cabeceras personalizadas permitidas.
- *Access-Control-Allow-Credentials:* si se permiten cookies y credenciales.

4. **Procesamiento por el navegador:** Si el origen y los métodos están permitidos, el navegador completa la solicitud; si no, bloquea la respuesta y muestra un error de CORS.



Buenas prácticas en la implementación de CORS

- Restringir los orígenes permitidos a dominios específicos y evitar el uso de * si los datos son sensibles.
- Configurar correctamente los métodos y cabeceras permitidos.
- Evitar incluir null como origen permitido, ya que algunos navegadores pueden enviar este valor en solicitudes desde archivos locales u otros contextos, lo que podría generar riesgos de seguridad.

Ejemplo práctico

Si https://noticias.ejemplo.com quiere acceder a https://api-socio.com, el servidor de la API configuraría:

- Access-Control-Allow-Origin: https://noticias.ejemplo.com
- Access-Control-Allow-Methods: GET, POST
- Access-Control-Allow-Credentials: true

Con esta configuración, el navegador permite que la aplicación cliente acceda a los recursos de la API de manera segura.

CORS permite así integrar aplicaciones complejas que consumen APIs externas o recursos de terceros, asegurando que solo los orígenes autorizados puedan acceder a datos sensibles y respetando la política de mismo origen del navegador.

17. ¿Qué diferencia hay entre una cabecera Authorization y una cabecera Cookie?

Ambas cabeceras, **Authorization** y **Cookie**, son utilizadas para enviar información de autenticación y estado del cliente a un servidor, pero tienen propósitos, mecanismos y contextos de uso distintos.

- La cabecera **Authorization** se emplea para enviar credenciales directamente en cada solicitud al servidor, con el objetivo de que este pueda verificar la identidad del cliente y decidir si tiene acceso a un recurso protegido. Puede adoptar distintos mecanismos de autenticación, como Basic, Digest o el más común en la actualidad, Bearer, que suele usarse para transmitir tokens JWT u OAuth 2.0. En este caso, la aplicación cliente es responsable de gestionar el token, almacenarlo (por ejemplo, en memoria o en localStorage) y adjuntarlo manualmente en cada petición. Por eso es la opción más común en APIs REST y en sistemas modernos que funcionan de manera stateless.

  Por ejemplo, un token de acceso en una API REST se envía típicamente en esta cabecera:

  Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI...

- En cambio, la cabecera **Cookie** permite que el cliente envíe automáticamente información que el servidor le asignó previamente mediante la instrucción Set-Cookie. Estos datos suelen incluir identificadores de sesión, configuraciones de usuario o incluso tokens, y se envían en cada solicitud posterior al mismo dominio mientras la cookie sea válida. A diferencia de Authorization, el manejo acá es automático, ya que el navegador se encarga de almacenar la cookie, enviarla al servidor y eliminarla cuando caduca. Además, se pueden configurar atributos como HttpOnly o Secure para mejorar la seguridad, evitando que sean accedidas desde JavaScript o que se transmitan en conexiones no seguras.

  Por ejemplo, la cookie podría contener un ID de sesión que el servidor usa para identificar al usuario.

  Cookie: sessionId=abc1234; preferences=light-theme

De esta manera, Authorization funciona como un mecanismo explícito y controlado por el cliente para enviar credenciales, mientras que Cookie se utiliza como un sistema automatizado de persistencia de estado que facilita al servidor mantener sesiones activas entre múltiples solicitudes. Ambas se pueden usar para autenticación, pero Authorization suele ser preferida en APIs modernas sin estado (stateless), mientras que Cookie es más común en aplicaciones web tradicionales con sesiones.

Cuadro comparativo entre las cabeceras Authorization y Cookie

Característica	Authorization	Cookie
Propósito	Autenticación explícita	Mantener estado/sesión
Manejo	Manual (la aplicación cliente la genera y adjunta).	Automático (el navegador la gestiona por sí mismo).
Contenido	Típicamente un token (JWT, OAuth).	Un par clave-valor, a menudo un ID de sesión.
Contexto	Común en APIs y aplicaciones de una sola página (SPA).	Común en aplicaciones web tradicionales basadas en sesiones.
Forma de envío	Cabecera en cada solicitud	Cabecera que contiene datos previamente asignados
Usos comunes	APIs REST con tokens (JWT, OAuth)	Aplicaciones web con sesiones tradicionales
Persistencia	Temporal (por la duración de la solicitud)	Persistente (almacenada en el navegador)
Seguridad	Generalmente se almacena en localStorage o sessionStorage, lo que la hace susceptible a XSS si no se protege adecuadamente.	Puede configurarse como HttpOnly para evitar que JavaScript acceda a ella, lo que la protege de ataques XSS.

18. ¿Qué son las cabeceras de seguridad como Content-Security-Policy o X-Frame-Options? ¿Cómo ayudan a mitigar ataques comunes?

Las cabeceras de seguridad HTTP son instrucciones que el servidor envía al navegador para indicarle cómo manejar distintos aspectos de la carga de recursos y del comportamiento de la página. Su función principal es reforzar la seguridad del lado del cliente, reduciendo la superficie de ataque frente a técnicas como cross-site scripting (XSS), clickjacking o inyecciones de contenido.

- Una de las más importantes es **Content-Security-Policy (CSP)**, que establece qué orígenes de contenido están autorizados para la página. Esto incluye scripts, hojas de estilo, imágenes, fuentes o iframes, entre otros recursos. Gracias a esta política,

el navegador puede bloquear la ejecución de código proveniente de dominios no confiables, lo que resulta clave para prevenir ataques XSS.

Por ejemplo, una configuración como:
Content-Security-Policy: default-src 'self'; script-src 'self' https://apis.ejemplo.com

Permite cargar recursos solo desde el propio dominio y scripts únicamente desde ese dominio o desde apis.ejemplo.com.
De esta forma, si un atacante intenta inyectar un script malicioso desde otro origen, el navegador lo va a bloquear automáticamente porque no está en la lista de fuentes permitidas.

- Otra cabecera muy relevante es **X-Frame-Options**, que controla si una página puede ser incrustada dentro de un <iframe>. Su propósito es evitar ataques de clickjacking, en los que un atacante superpone un sitio legítimo dentro de un iframe invisible y engaña al usuario para que haga clic en botones o enlaces sin darse cuenta. Los valores más comunes son DENY, que impide por completo la incrustación, y SAMEORIGIN, que solo la permite si el iframe proviene del mismo dominio.

  Con esta cabecera activada, aunque un atacante intente embeber la página en un sitio fraudulento, el navegador directamente no permitirá que se cargue en el iframe. De esa manera se evita que el usuario interactúe sin querer con elementos críticos de la aplicación, como botones de compra o formularios sensibles.

Existen además otras cabeceras de seguridad que refuerzan la protección.

- **X-Content-Type-Options: nosniff** evita que el navegador interprete archivos con un tipo MIME incorrecto, reduciendo el riesgo de ejecución de código no autorizado.

  Esto mitiga ataques en los que un archivo aparentemente inofensivo, como una imagen, podría ser interpretado como un script ejecutable por el navegador y correr código malicioso.

- **Strict-Transport-Security (HSTS)** obliga al navegador a utilizar siempre HTTPS, protegiendo contra ataques de *downgrade* o intentos de intercepción en conexiones inseguras.

  Si un atacante trata de forzar una conexión por HTTP, el navegador va a rechazarla automáticamente, asegurando que la comunicación siempre esté cifrada y sea mucho más difícil de manipular.

- También se encuentra **Referrer-Policy**, que limita la información del encabezado *Referer* compartida con otros sitios, lo que ayuda a proteger la privacidad y a reducir filtraciones de datos sensibles.

  Esto significa que si un usuario navega desde una página con datos internos a un sitio externo, la cabecera puede configurarse para que no se transmitan esos datos

en la URL, evitando así que información confidencial termine en manos equivocadas.

19. ¿Cuáles son las diferencias entre HTTP/1.1, HTTP/2 y HTTP/3?

HTTP/1.1, HTTP/2 y HTTP/3 representan distintas etapas en la evolución del protocolo HTTP, buscando mejorar velocidad, eficiencia y seguridad en la comunicación entre cliente y servidor. HTTP/1.1 es más simple pero menos eficiente, HTTP/2 introduce multiplexación y compresión de cabeceras para acelerar la carga de páginas, y HTTP/3 optimiza la latencia y la robustez en la comunicación al nivel del transporte, especialmente en entornos con pérdidas de paquetes o cambios de red.

- **HTTP/1.1** es la versión más antigua ampliamente adoptada. Funciona sobre TCP y mantiene conexiones persistentes, pero procesa las solicitudes de manera secuencial, generando retrasos si un recurso tarda demasiado (problema conocido como head-of-line blocking). Cada recurso requiere su propia solicitud, y los encabezados se envían completos en cada petición, aumentando el overhead y la latencia.
- **HTTP/2** mejora estas limitaciones al mantener el transporte sobre TCP, pero introduciendo multiplexación, lo que permite enviar múltiples solicitudes y recibir múltiples respuestas simultáneamente por una sola conexión. Además, comprime los encabezados con HPACK y permite priorizar recursos, mientras que el server push puede enviar archivos al cliente antes de que los solicite, optimizando la velocidad de carga. El uso de un formato binario reduce errores y mejora la eficiencia general.
- **HTTP/3** da un salto importante al usar QUIC sobre UDP como protocolo de transporte, incorporando TLS 1.3 directamente en el handshake. Esto elimina los bloqueos de TCP a nivel de transporte, permitiendo que cada stream funcione de forma independiente y evitando retrasos por pérdida de paquetes. También ofrece un handshake más rápido y permite cambiar de red sin interrumpir la conexión, lo que mejora significativamente la experiencia en redes móviles o inestables.

Cuadro comparativo entre entre HTTP/1.1, HTTP/2 y HTTP/3

Característica	HTTP/1.1	HTTP/2	HTTP/3
Año de estándar	1997	2015	2020
Protocolo de transporte	TCP	TCP	QUIC (sobre UDP)
Multiplexación	Limitada / Secuencial	Sí, múltiples solicitudes/respuestas en la misma conexión	Sí, flujos independientes sin head-of-line blocking
Compresión de cabeceras	No	HPACK	QPACK

Seguridad	Opcional (HTTPS separado)	TLS sobre TCP	TLS 1.3 integrado en QUIC
Latencia	Alta en páginas con muchos recursos	Menor que HTTP/1.1 gracias a multiplexación	Más baja, especialmente en redes con pérdida de paquetes
Head-of-line blocking	Sí, un paquete perdido bloquea los demás	Sí, sobre TCP aún puede haber bloqueo	No, cada flujo es independiente
Compatibilidad con HTTP/1.1	Sí	Sí, misma semántica	Sí, misma semántica de métodos y URLs
Ventaja principal	Simple y ampliamente soportado	Eficiencia, menor latencia, compresión de cabeceras	Máxima eficiencia y menor latencia en redes inestables, cifrado integrado

20. ¿Qué es un "keep-alive" en HTTP y cómo mejora el rendimiento de las aplicaciones?

En HTTP, keep-alive es una característica que permite mantener abierta la misma conexión TCP entre el cliente y el servidor para múltiples solicitudes/respuestas, en lugar de abrir y cerrar una nueva conexión para cada recurso. Esto se especifica mediante la cabecera *Connection: keep-alive.*

Este comportamiento está activado por defecto en HTTP/1.1 y en todas las versiones posteriores. HTTP/2 y HTTP/3 integran el concepto de conexión persistente de manera nativa mediante multiplexación sobre una sola conexión, lo que optimiza aún más el rendimiento y la eficiencia de la comunicación cliente-servidor. Esto hace que la experiencia de usuario sea más rápida y estable, especialmente en páginas con numerosos recursos o en redes con alta latencia.

El uso de keep-alive mejora significativamente el rendimiento de las aplicaciones web al reducir la sobrecarga de la creación de conexiones. La mejora se debe a los siguientes factores:

1. **Reduce la latencia:** Abrir una conexión TCP es un proceso costoso y lento que requiere varios intercambios de mensajes. Si cada recurso de una página (HTML, CSS, JavaScript, imágenes) necesitara una nueva conexión, el tiempo total de carga aumentaría drásticamente. El keep-alive evita el tiempo adicional de establecer nuevas conexiones TCP y realizar handshakes TLS repetidos.

2. **Disminuye el uso de recursos del servidor:** Mantener una conexión abierta es más eficiente que abrir y cerrar repetidamente conexiones, ya que consume menos recursos de la CPU y la memoria del servidor.
3. **Facilita la reutilización de la conexión:** El navegador puede enviar múltiples solicitudes de forma secuencial a través de la misma conexión, lo que es especialmente útil para cargar todos los recursos de una página web de manera más rápida.

El keep-alive transforma la forma en que los navegadores interactúan con los servidores. En lugar de un modelo ineficiente de "una conexión por recurso", permite un modelo de "una conexión para múltiples recursos", lo que se traduce en una carga de página mucho más rápida y una mejor experiencia para el usuario.

Preguntas de implementación práctica:

21. ¿Cómo manejarías la autenticación en una API basada en HTTP/HTTPS? ¿Qué métodos conoces (Basic, OAuth, JWT, etc.)?

La autenticación en una API HTTP/HTTPS es el proceso mediante el cual se verifica la identidad de un cliente antes de permitirle acceder a recursos protegidos, y su correcta implementación es fundamental para proteger los datos y evitar accesos no autorizados.

Dependiendo del tipo de API y del contexto de uso, se pueden emplear distintos métodos de autenticación, cada uno con sus ventajas y limitaciones.

- **HTTP Basic Authentication** envía usuario y contraseña codificados en Base64 en la cabecera Authorization. Aunque es muy simple de implementar, no es seguro por sí solo, ya que las credenciales pueden ser interceptadas si no se usa HTTPS.
- **OAuth 2.0** es un marco de autorización que permite a aplicaciones de terceros acceder a recursos limitados en nombre de un usuario sin compartir sus credenciales. Mediante este flujo, la API emite un token de acceso (Bearer token) que se envía en la cabecera Authorization, facilitando la revocación y control de permisos sin exponer contraseñas.
- **JWT (JSON Web Token)** es un token firmado digitalmente que contiene información del usuario y sus permisos. Se envía en la cabecera Authorization: Bearer <token> y permite que la API sea stateless, ya que el servidor valida la firma sin necesidad de mantener sesiones activas, aumentando la escalabilidad y eficiencia.
- **API Key** consiste en una clave única que se envía en la cabecera o en la URL para identificar al cliente. Es sencilla de usar, pero menos segura que OAuth o JWT si no se protege mediante HTTPS.
- **Cookies y sesiones** se usan para mantener el estado entre el cliente y el servidor mediante un identificador de sesión enviado en la cabecera Cookie. Este método es más común en aplicaciones web tradicionales y menos recomendado en APIs modernas stateless.

Para APIs modernas se recomienda usar OAuth 2.0 combinado con JWT, asegurando que todas las solicitudes se realicen sobre HTTPS. OAuth 2.0 gestiona la autorización inicial y emite un token JWT que el cliente envía en cada solicitud a la API, proporcionando un

balance entre seguridad, eficiencia y escalabilidad. Además, es recomendable implementar expiración y revocación de tokens, evitando el uso de contraseñas en cada solicitud y protegiendo los datos frente a ataques tipo man-in-the-middle.

22. ¿Qué es un proxy inverso (reverse proxy) y cómo se utiliza en entornos HTTP/HTTPS?

Un proxy inverso (o *reverse proxy*) es un servidor que se sitúa entre los clientes (por ejemplo, navegadores web) y uno o más servidores web de origen. En lugar de que los clientes se conecten directamente a un servidor, todas las solicitudes son enviadas al proxy inverso. Este servidor las recibe y las reenvía a los servidores de destino apropiados. Al implementar un reverse proxy, las aplicaciones pueden ser más escalables, seguras y fáciles de mantener, mientras que los clientes siempre interactúan con una única dirección pública.

Esencialmente, actúa como un "punto de entrada" centralizado para las solicitudes a una aplicación o grupo de servidores, lo que lo diferencia de un proxy normal (o *forward proxy*), que se sitúa entre los clientes y el Internet para proteger la privacidad o el acceso a los sitios.

El proxy inverso tiene múltiples usos para mejorar la seguridad, el rendimiento y la gestión de aplicaciones en entornos HTTP/HTTPS:

- **Distribución de carga (load balancing):** Permite distribuir las solicitudes entrantes entre varios servidores de origen, garantizando que ningún servidor se sobrecargue. Esto mejora la disponibilidad de la aplicación y mantiene tiempos de respuesta más consistentes para los usuarios.
- **Caché de contenido:** almacena respuestas frecuentes para reducir la carga de los servidores backend y mejorar la velocidad de respuesta.
- **Seguridad:** Oculta la dirección IP y la estructura interna de los servidores de backend, protegiéndolos de accesos directos. Además, puede actuar como un firewall de aplicaciones web (WAF), filtrando tráfico malicioso o ataques como inyección SQL y XSS antes de que lleguen a los servidores.
- **Terminación de TLS/SSL:** el proxy puede manejar el cifrado HTTPS, liberando a los servidores backend de la carga de cifrado y simplificando la gestión de certificados.
- **Compresión y optimización de tráfico:** mejora el rendimiento reduciendo el tamaño de las respuestas enviadas al cliente.

23. ¿Cómo implementarías una redirección automática de HTTP a HTTPS en un servidor?

Para implementar una redirección automática de HTTP a HTTPS, primero hay que asegurarse de contar con un certificado SSL/TLS válido en el servidor.

La idea es que todas las solicitudes que lleguen por HTTP (puerto 80) se redirijan a la versión segura HTTPS (puerto 443) mediante un código de estado HTTP 301 (permanente) o 302 (temporal) y la cabecera Location. Esto garantiza que los usuarios siempre naveguen de manera cifrada, protegiendo la confidencialidad e integridad de los datos.

Pasos generales:

- **Configurar SSL/TLS:** Instalar y habilitar un certificado válido en el servidor para que pueda manejar conexiones HTTPS correctamente.
- **Capturar solicitudes HTTP:** Configurar el servidor para que todas las solicitudes que lleguen por HTTP sean detectadas.
- **Redirigir a HTTPS:** Enviar una redirección a la misma URL pero con protocolo HTTPS, utilizando el código de estado correspondiente.
- **Opcional: HSTS (Strict-Transport-Security):** Esta cabecera indica a los navegadores que deben conectarse siempre por HTTPS, evitando futuros intentos por HTTP.

Ejemplo práctico según el servidor web:

Apache:
Se puede usar el módulo mod_rewrite agregando estas reglas en el archivo *.htaccess*:

- *RewriteEngine On*
  - Activa el motor de reescritura.
- *RewriteCond %{HTTPS} off*
  - Esta es la condición. Le dice a Apache que la siguiente regla solo se aplique si la conexión no es HTTPS (es decir, es HTTP)
- *RewriteRule ^(.*)$ https://%{HTTP_HOST}%{REQUEST_URI} [L,R=301]*
  - *^(.*)$:* Captura toda la URL de la solicitud.
  - *https://%{HTTP_HOST}%{REQUEST_URI}*: Reemplaza el protocolo por https:// y mantiene el mismo dominio y la misma ruta.
  - *[L,R=301]*: Son las banderas. R=301 le dice al navegador que es una redirección permanente (Moved Permanently), lo cual es ideal para SEO, y L (Last) le indica a Apache que no procese más reglas.

Esto verifica si la conexión no es HTTPS y redirige al mismo host y ruta pero con el protocolo seguro.

Nginx:
Se configura un bloque server en el archivo de configuración del servidor que escuche en el puerto 80 y redirija al HTTPS:

```
 server {
   listen 80;
   server_name ejemplo.com www.ejemplo.com;
   return 301 https://$host$request_uri;
}
```

- *listen 80*: Nginx escuchará las solicitudes entrantes en el puerto 80.
- *server_name*: Se pone el nombre del dominio, con y sin www.
- *return 301 https://$host$request_uri*: Esta es la directiva que realiza la redirección permanente.
  - 301: Es el código de estado para la redirección permanente.

    - https://$host$request_uri: Reconstruye la URL con HTTPS. $host es el nombre de host de la solicitud y $request_uri es la ruta completa.

24. ¿Cómo mitigarías un ataque de denegación de servicio (DDoS) en un servidor HTTP?

Un ataque de denegación de servicio distribuido (DDoS) ocurre cuando un atacante inunda un servidor HTTP con un volumen masivo de solicitudes, saturando sus recursos y haciendo que los usuarios legítimos no puedan acceder al servicio. Mitigar un DDoS requiere combinar estrategias de red, servidor y aplicación para reducir el impacto del ataque y mantener la disponibilidad del sistema.

Principales medidas de mitigación:

- **Detección y monitoreo:** Es fundamental identificar el ataque lo antes posible. Se pueden usar herramientas de monitoreo de red para detectar picos inusuales en solicitudes, volumen de datos o patrones de IP. Además, sistemas de análisis de comportamiento ayudan a reconocer tráfico anómalo comparado con el tráfico normal de la aplicación.
- **Uso de un proveedor de mitigación DDoS o CDN:** Plataformas como Cloudflare, Akamai, AWS Shield o Google Cloud Armor actúan como un filtro entre Internet y tu servidor. Estas redes tienen gran capacidad de ancho de banda y usan técnicas avanzadas para bloquear tráfico malicioso, dejando pasar solo el tráfico legítimo. Además, al usar una CDN se distribuye el tráfico globalmente, absorbiendo picos de carga.
- **Limitación de solicitudes y filtrado a nivel de servidor:** Configurar rate limiting en servidores web como Nginx o Apache permite limitar la cantidad de solicitudes por IP en un período de tiempo. También se pueden bloquear IPs identificadas como maliciosas y usar un Web Application Firewall (WAF) para filtrar solicitudes sospechosas que puedan consumir recursos de la aplicación.
- **Arquitectura escalable y balanceo de carga:** Distribuir el tráfico entre varios servidores mediante un load balancer evita que un único servidor se sature. Combinado con caching de contenido estático, se reduce la carga sobre los servidores backend y se mejora la respuesta ante picos de tráfico.
- **Optimización del servidor:** Configurar correctamente conexiones persistentes (keep-alive) y habilitar protocolos modernos como HTTP/2 o HTTP/3 permite manejar mejor múltiples conexiones concurrentes, aumentando la resiliencia frente a ataques de alto volumen.

Estas medidas deben implementarse en conjunto para lograr una defensa en capas: monitoreo interno, reglas de servidor y firewall, uso de CDNs o servicios especializados y optimización de la infraestructura. Esta combinación permite que el tráfico legítimo llegue al servidor mientras se mitiga la efectividad del ataque DDoS.

25. ¿Qué problemas podrías enfrentar al trabajar con APIs que dependen de HTTP, y cómo los resolverías?

Al trabajar con APIs que dependen de HTTP/HTTPS, se pueden enfrentar diversos problemas que afectan la seguridad, el rendimiento, la gestión de estado y la escalabilidad. A continuación se detallan algunos problemas que podrían surgir y sus posibles soluciones.

- Problemas de autenticación y autorización
    - Una API insegura o mal configurada puede permitir accesos no autorizados, exponiendo información sensible y recursos críticos. Esto ocurre principalmente si no se implementan protocolos seguros o si las credenciales se transmiten en texto plano.
    - Para resolverlo, se debe obligatoriamente utilizar HTTPS para cifrar la comunicación y evitar que terceros intercepten las credenciales. Además, se recomienda emplear OAuth 2.0 o JWT para la autenticación, asegurando que cada solicitud se valide correctamente y que los permisos se verifiquen según el usuario o la aplicación que realiza la petición.
- Problemas de latencia y rendimiento
    - En APIs que reciben muchas solicitudes o manejan recursos pesados, las conexiones HTTP secuenciales pueden generar demoras importantes, afectando la experiencia del usuario. La repetición de encabezados y la apertura de múltiples conexiones TCP incrementa la carga en el servidor y aumenta la latencia.
    - La solución consiste en optimizar la infraestructura y el protocolo. Se puede activar keep-alive para reutilizar conexiones existentes, migrar a HTTP/2 o HTTP/3 para aprovechar la multiplexación y la compresión de cabeceras, y utilizar caché y optimización de payloads. En entornos de alto tráfico, también conviene implementar balanceo de carga para distribuir las solicitudes entre varios servidores.
- Errores de comunicación y fallos de red
    - Las solicitudes pueden fallar debido a timeouts, pérdida de paquetes o servidores caídos, generando interrupciones en el servicio y errores en las aplicaciones que consumen la API.
    - Para mitigarlo, se recomienda implementar reintentos con backoff exponencial, manejar correctamente los códigos de estado HTTP para que la aplicación pueda reaccionar ante errores y establecer sistemas de monitoreo y alertas que detecten fallas o picos anómalos de tráfico.
- Problemas de CORS (Cross-Origin Resource Sharing)
    - Cuando una aplicación web intenta acceder a una API desde un dominio diferente, los navegadores pueden bloquear la solicitud por políticas de seguridad, impidiendo la comunicación entre cliente y servidor.
    - La solución es configurar correctamente las cabeceras CORS, como *Access-Control-Allow-Origin* y *Access-Control-Allow-Methods*, permitiendo solo los orígenes confiables y los métodos HTTP necesarios para que la aplicación funcione sin comprometer la seguridad.
- Seguridad y ataques web
    - Las APIs pueden ser vulnerables a ataques de inyección, XSS, CSRF o DDoS si no se implementan medidas de protección adecuadas, lo que puede comprometer datos de usuarios o la disponibilidad del servicio.
    - Para prevenir estos riesgos, se deben implementar cabeceras de seguridad como CSP, X-Frame-Options y HSTS, establecer rate limiting para limitar solicitudes por cliente, validar todos los datos que ingresan a la API y monitorear el tráfico para detectar patrones sospechosos.
- Versionado de la API

    - Los cambios en la API pueden romper aplicaciones cliente si no se maneja correctamente la compatibilidad entre versiones.
    - La solución es usar versionamiento explícito, ya sea en la URL (/v1/usuarios) o en las cabeceras, y mantener compatibilidad hacia atrás cuando sea posible. Esto permite actualizar la API sin afectar a los clientes que todavía dependen de versiones anteriores.
- Problemas de documentación y entendimiento de endpoints
    - La falta de documentación clara puede llevar a un uso incorrecto de la API, provocando errores y retrabajo en las aplicaciones cliente.
    - Se recomienda emplear herramientas como OpenAPI o Swagger, brindar ejemplos claros de cada endpoint y establecer validaciones de contrato entre cliente y servidor para asegurar que la API se use correctamente y que los cambios futuros sean previsibles.
- Problemas de gestión de estado
    - HTTP es un protocolo sin estado (stateless), lo que significa que cada solicitud se procesa de manera independiente, sin que el servidor recuerde interacciones anteriores. Esto dificulta mantener la sesión de un usuario y puede complicar la personalización de la experiencia o la autenticación persistente en aplicaciones que consumen la API.

    - Para solucionarlo, se utilizan mecanismos que permiten mantener la sesión de manera segura. Una opción es usar cookies, donde el servidor envía un identificador de sesión que el navegador almacena y devuelve en cada solicitud, permitiendo reconocer al usuario. Otra alternativa es emplear tokens de autenticación JWT, que contienen información del usuario y sus permisos. El cliente envía el token en cada solicitud y el servidor verifica su validez sin necesidad de guardar el estado de la sesión, siendo una solución ideal para APIs RESTful y aplicaciones stateless.

26. ¿Qué es un cliente HTTP? ¿Mencionar la diferencia entre los clientes POSTMAN y CURL?

Un cliente HTTP es un programa o herramienta que permite enviar solicitudes HTTP a un servidor y recibir sus respuestas. Funciona como la parte que inicia la comunicación, solicitando al servidor recursos específicos o enviando datos, y es esencial para interactuar con APIs, probar endpoints y depurar la comunicación entre cliente y servidor. Los navegadores web son un ejemplo común de clientes HTTP, aunque existen herramientas especializadas para desarrolladores.

En cuanto a Postman y cURL, ambos son clientes HTTP, pero con enfoques distintos:

- **cURL**: es una herramienta de línea de comandos que permite enviar solicitudes HTTP/HTTPS especificando métodos, cabeceras y datos mediante comandos. Es ideal para automatización, scripting y entornos donde no hay interfaz gráfica. Su ventaja principal es la potencia y ligereza; la desventaja es que requiere aprender su sintaxis, lo que puede ser menos intuitivo para principiantes.

- **Postman**: es una aplicación con interfaz gráfica que facilita la creación, prueba y documentación de APIs. Permite guardar colecciones de solicitudes, configurar cabeceras y cuerpos de forma visual y colaborar en equipos. Su ventaja es la facilidad de uso y la claridad al probar manualmente APIs, pero su desventaja es que no es tan conveniente para automatización en scripts o entornos de servidor.

Ambos son clientes HTTP que permiten probar y consumir APIs, pero POSTMAN es más visual e intuitivo para pruebas y documentación, haciendo más amigable el trabajo manual con endpoints, mientras que cURL es más liviano y potente para automatización y scripting en entornos de línea de comandos.

Diferencias entre POSTMAN y cURL:

Característica	POSTMAN	CURL
Tipo de herramienta	Aplicación gráfica (GUI) y multiplataforma	Herramienta de línea de comandos (CLI)
Facilidad de uso	Muy fácil de usar, ideal para pruebas visuales y manejo de colecciones	Requiere conocer sintaxis y comandos, más técnico
Funciones avanzadas	Permite guardar colecciones, variables, tests automatizados, documentación	Orientado a solicitudes rápidas y scripts automatizados
Interacción con APIs	Visualización de solicitudes y respuestas con JSON, cabeceras, cookies, autenticación	Devuelve la respuesta en la consola; útil para scripts y automatización
Uso en automatización	Permite pruebas automáticas pero principalmente GUI	Ideal para integración en scripts, CI/CD y automatización de tareas
Licencia y acceso	Gratis con versión de pago para funciones avanzadas en la nube	Open source y gratuito, disponible en la mayoría de sistemas

Preguntas de GIT

27. ¿Qué es GIT y para qué se utiliza en desarrollo de software?

Git es un sistema de control de versiones distribuido que permite a los desarrolladores registrar, gestionar y colaborar sobre los cambios en el código fuente de un proyecto a lo largo del tiempo. Es la herramienta estándar en el desarrollo de software para gestionar proyectos, tanto grandes como pequeños. Git organiza y protege el historial del proyecto, facilita el trabajo en equipo y mejora la eficiencia del desarrollo de software.

Git se utiliza en desarrollo de software para:

- Seguimiento de cambios: Git registra cada modificación en los archivos del proyecto, permitiendo consultar el historial, revertir cambios o comparar versiones anteriores. Esto facilita detectar errores y entender la evolución del código.
- Colaboración: Permite que varios desarrolladores trabajen sobre el mismo proyecto simultáneamente sin sobrescribir el trabajo de otros. Cada uno puede trabajar en su propia copia y luego integrar los cambios mediante fusiones (merges).
- Gestión de ramas y entornos: Con Git se pueden crear ramas independientes para desarrollar nuevas funciones, corregir errores o experimentar, sin afectar la rama principal. Luego, las ramas se pueden fusionar de manera controlada.
- Integración con repositorios remotos: Git se conecta con plataformas como GitHub, GitLab o Bitbucket, lo que facilita almacenar código en la nube, colaborar en equipo y automatizar despliegues.
- Auditoría y trazabilidad: Se puede identificar quién hizo cada cambio y cuándo, lo que ayuda a revisar errores, asignar responsabilidades y mantener un historial claro del proyecto.
- Reversión de errores: Si se introduce un error en el código, Git permite regresar fácilmente a una versión anterior y estable del proyecto.

28. ¿Cuál es la diferencia entre un repositorio local y un repositorio remoto en GIT?

La principal diferencia entre un repositorio local y un repositorio remoto en Git es su ubicación y su propósito en el flujo de trabajo. El local es tu copia personal de trabajo en tu máquina, mientras que el remoto es una versión compartida del proyecto en un servidor.

- **Repositorio Local:** Un repositorio local es la copia completa de un proyecto que se encuentra en la computadora del desarrollador. Permite editar archivos, preparar cambios (staging) y guardar versiones mediante commits. Todos los cambios permanecen en la máquina del desarrollador y no afectan a otros hasta que se sincronizan con un repositorio remoto. Esto permite trabajar de forma aislada y probar nuevas funcionalidades sin interferir con el proyecto compartido.
- **Repositorio Remoto:** Un repositorio remoto es la versión del proyecto alojada en un servidor o en la nube, por ejemplo en GitHub, GitLab o Bitbucket. Su función es servir como punto de encuentro para la colaboración entre desarrolladores. Permite subir cambios (push), descargar los cambios de otros (pull) y mantener la sincronización entre todos los miembros del equipo. Además, funciona como respaldo del proyecto, asegurando que el código no se pierda si falla alguna máquina local, y facilita la distribución del trabajo entre los colaboradores.

El repositorio local es para desarrollo individual, mientras que el remoto es para colaboración y respaldo, y Git permite sincronizar ambos para mantener el proyecto actualizado y consistente.

29. ¿Cómo se crea un nuevo repositorio en GIT y cuál es el comando para inicializarlo? Explica la diferencia entre los comandos git commit y git push.

En Git, un nuevo repositorio puede crearse de dos formas principales: inicializar uno desde cero en tu máquina o clonar un repositorio remoto existente.

1. Inicializar un repositorio local
    - Primero, se crea una carpeta para el proyecto y se navega hasta ella desde la terminal.
    - Luego se ejecuta el comando: *git init*
    - Este comando genera una subcarpeta oculta .git que contiene todo el historial de versiones y la configuración del repositorio. A partir de este momento, Git empezará a rastrear los cambios que realices en esa carpeta.
2. Clonar un repositorio remoto
    - Si el proyecto ya existe en un servicio como GitHub, GitLab o Bitbucket, se puede clonar usando: *git clone <url-del-repositorio>*
    - Esto crea una copia local del repositorio remoto, incluyendo todo su historial de commits y ramas.

Diferencia entre git commit y git push

git commit se utiliza para guardar los cambios preparados (con git add) en el repositorio local. Cada commit actúa como una instantánea del proyecto en ese momento y permanece únicamente en tu máquina hasta que se sincroniza. Por ejemplo:

git commit -m "Agrega función de login"

- git commit guarda los cambios en el historial local.

git push, en cambio, envía los commits locales al repositorio remoto, compartiendo tus cambios con el resto del equipo. Hasta que no se ejecuta este comando, los demás colaboradores no verán tus cambios. Por ejemplo:

git push origin main

- git push sube esos cambios al repositorio remoto para que otros los vean.

30. ¿Qué es un "branch" en GIT y para qué se utilizan las ramas en el desarrollo de software?

En Git, un branch (rama) es un puntero móvil a un commit específico dentro del historial del repositorio. Cada rama representa una línea de desarrollo independiente que permite trabajar sobre nuevas funcionalidades, correcciones o experimentos sin afectar la rama principal del proyecto (generalmente llamada main o master).

Usos de las ramas en el desarrollo de software:

- **Aislar cambios:** Permiten trabajar en nuevas funcionalidades o correcciones sin alterar el código estable de la rama principal. Si algo falla en la rama de desarrollo, el proyecto principal permanece intacto.
- **Colaboración en equipo:** Cada desarrollador puede trabajar en su propia rama de manera simultánea y luego fusionar (merge) los cambios al repositorio principal, evitando conflictos directos sobre el mismo código.

- **Gestión de entornos:** Se pueden crear ramas específicas para desarrollo, pruebas y producción, lo que facilita el control del ciclo de vida del software y mantiene separadas las distintas etapas del proyecto.
- **Pruebas seguras:** Se pueden crear ramas experimentales para probar nuevas ideas o funcionalidades sin comprometer la versión oficial del proyecto, reduciendo el riesgo de introducir errores en el código estable.

Una vez que el trabajo en una rama está completo y se ha verificado, se puede fusionar (con el comando git merge) de vuelta a la rama principal, integrando los cambios en el proyecto estable.

31. ¿Qué significa hacer un "merge" en GIT y cuáles son los posibles conflictos que pueden surgir durante un merge?

Hacer un "merge" en Git significa combinar los cambios de una rama en otra, integrando los historiales de commits de ambas. Esto se utiliza normalmente para llevar el trabajo realizado en una rama de desarrollo, como *feature/login*, a la rama principal (*main o master*).

Cuando se ejecuta un merge, Git intenta fusionar automáticamente los cambios realizados en ambas ramas. Si los cambios afectan a diferentes partes del código, la fusión ocurre sin problemas.

Un conflicto de merge ocurre cuando Git no puede fusionar los cambios de forma automática porque las ramas que se están combinando modificaron la misma línea o la misma parte de un archivo de manera diferente. En estos casos, Git detiene el proceso y requiere que un humano resuelva el conflicto de forma manual.

Posibles conflictos durante un merge:

- **Diferentes cambios en la misma línea:** Si dos ramas modifican la misma línea de un archivo de manera distinta, Git no puede decidir cuál conservar y marca un conflicto.
- **Eliminación y modificación simultáneas:** Ocurre cuando un archivo o línea es eliminada en una rama mientras que en la otra se realiza una modificación sobre ese mismo contenido.
- **Renombrado distinto de un mismo archivo:** Si un archivo es renombrado de manera diferente en cada rama, Git no puede fusionar automáticamente los cambios.

Cuando surge un conflicto, Git inserta delimitadores (<<<<<<<, =======, >>>>>>>) en el archivo afectado para que el desarrollador resuelva manualmente cuál versión conservar o cómo combinar los cambios. Una vez resuelto, se realiza un commit para completar el merge.

32. Describe el concepto de "branching model" en GIT y menciona algunos modelos comunes (por ejemplo, Git Flow, GitHub Flow).

Un Branching Model en Git es una estrategia definida para crear, organizar, fusionar y gestionar las ramas dentro de un proyecto. Permite que los desarrolladores trabajen en paralelo, integren nuevas funcionalidades, corrijan errores y desplieguen software de manera organizada, asegurando que la rama principal permanezca estable.

Los modelos de branching son importantes porque:

- Facilitan la **colaboración entre múltiples desarrolladores**.
- Mantienen una **línea principal estable** para producción.
- Ayudan a definir **flujos de trabajo consistentes** en un equipo.

Modelos comunes de branching

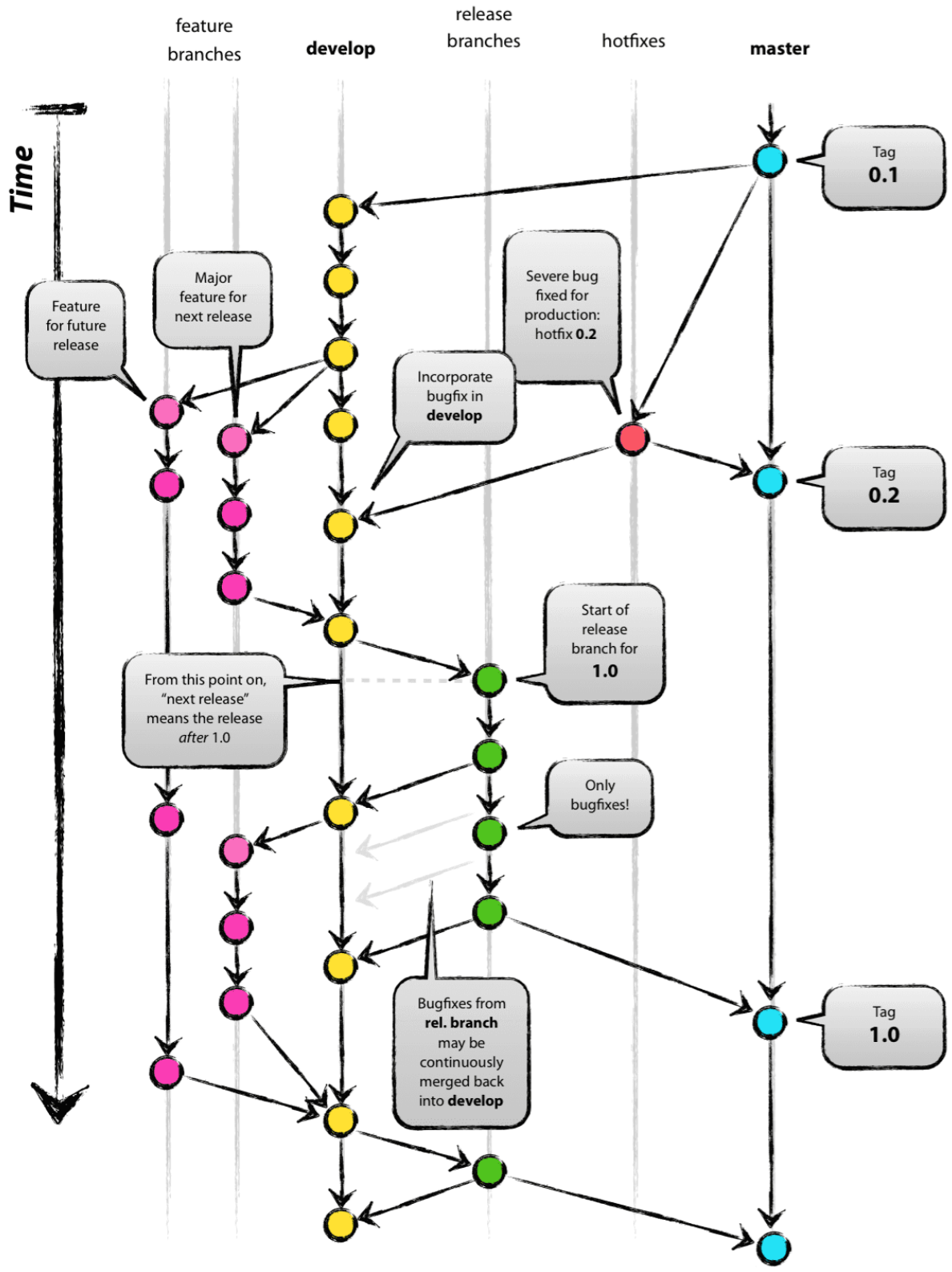
1. Git Flow

Git Flow es un modelo complejo y estructurado, ideal para proyectos con ciclos de lanzamiento planificados y versiones bien definidas. Utiliza varias ramas a largo plazo para organizar el desarrollo.

- Ramas principales:
    - **main**: Contiene el código en producción. Todo lo que está aquí es estable.
    - **develop**: La rama de desarrollo principal. Las nuevas características se fusionan aquí.
- Ramas de soporte:
    - **feature/nombre-de-la-funcionalidad:** Se crean a partir de develop para trabajar en nuevas características.
    - **release/version-x.x:** Se crean a partir de develop cuando se va a preparar una nueva versión. Se usan para correcciones de errores finales antes del lanzamiento.
    - **hotfix/nombre-del-bug:** Se crean a partir de main para corregir rápidamente errores críticos en producción.

Ventajas: Es robusto y proporciona un control estricto sobre las versiones, lo que lo hace perfecto para equipos grandes o proyectos con lanzamientos programados.

Desventajas: Puede ser demasiado complejo y rígido para proyectos pequeños o de desarrollo ágil.



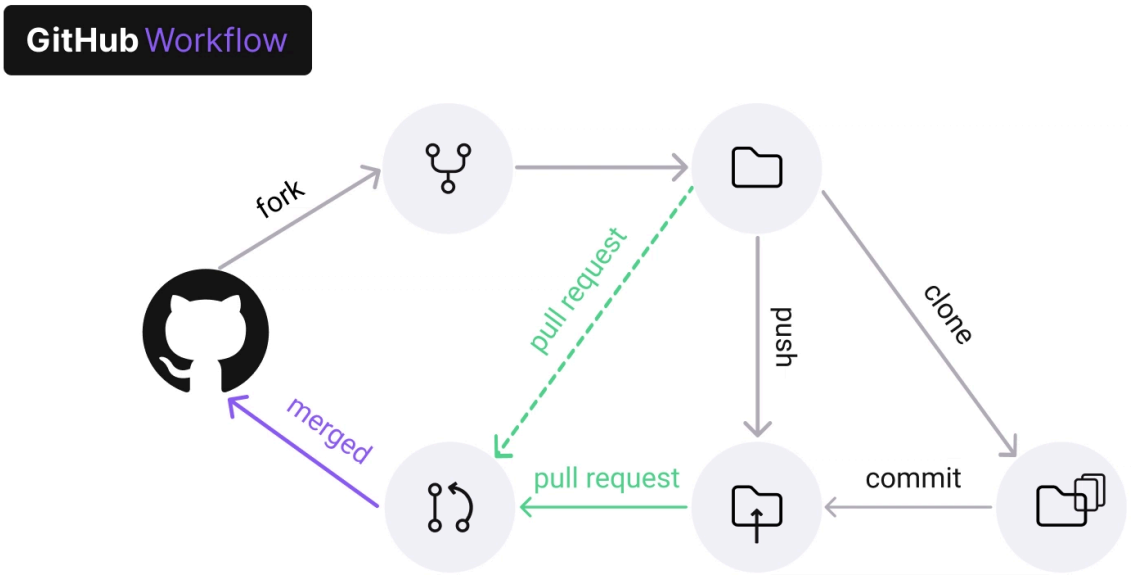
2. GitHub Flow

GitHub Flow es un modelo simple y ligero, diseñado para el desarrollo continuo y la entrega frecuente. A diferencia de Git Flow, se basa en la idea de que la rama main siempre debe ser desplegable.

- Flujo de trabajo:
    1. La rama **main** es el código en producción.
    2. Se crea una nueva rama **feature** descriptiva a partir de main para cada nueva característica o corrección de error.
    3. Se hacen commits y se trabaja en la rama.
    4. Cuando el trabajo está listo, se abre una "pull request" para discutir y revisar los cambios.
    5. Una vez que se aprueba, la rama se fusiona con main y se despliega inmediatamente en producción.
    6. La rama de la característica se elimina.

Ventajas: Es simple, ágil y promueve la entrega continua. No necesita múltiples ramas a largo plazo, lo que reduce la complejidad.

Desventajas: No es adecuado para proyectos con ciclos de lanzamiento largos o cuando se necesita mantener varias versiones simultáneamente.



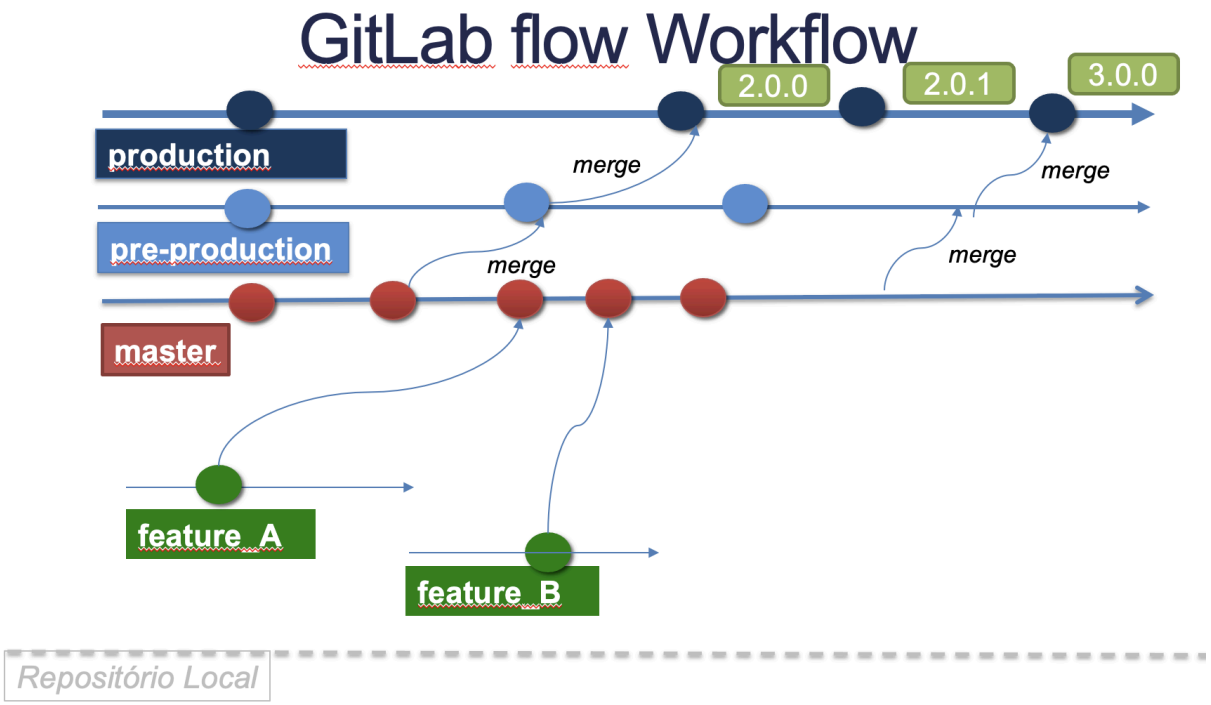
3. GitLab Flow

GitLab Flow es un modelo que combina la simplicidad de GitHub Flow con la estructura de Git Flow. Utiliza ramas de entorno (**production**, **pre-production**, **staging**) para gestionar los despliegues.

- Flujo de trabajo: Similar a GitHub Flow, se crean ramas de características a partir de main. Sin embargo, una vez que la rama se fusiona con main, los cambios no se despliegan directamente a producción. En su lugar, se fusionan en ramas intermedias (como staging) y luego en production para controlar el proceso de despliegue.

Ventajas: Ofrece un buen equilibrio entre simplicidad y control. Proporciona un entorno de prueba antes de la producción.

Desventajas: Puede ser un poco más complejo que GitHub Flow sin la rigidez de Git Flow.



33. ¿Cómo se deshace un cambio en GIT después de hacer un commit pero antes de hacer push?

En Git, si se ha realizado un commit local pero aún no se ha ejecutado el comando push para enviarlo al repositorio remoto, existen distintas formas de deshacer dicho commit, dependiendo del efecto deseado:

- **git reset --soft HEAD~1**
  Este comando revierte el último commit, pero mantiene los cambios en el área de staging. Resulta útil cuando se requiere corregir el mensaje del commit o agregar archivos que fueron omitidos.

- **git reset --mixed HEAD~1** (opción por defecto)
  Deshace el último commit y mantiene los cambios en el directorio de trabajo, retirándolos del área de staging. Es conveniente cuando se desea rehacer el commit desde el inicio.

- **`git reset --hard HEAD~1`**
  Elimina por completo el último commit y también los cambios en el directorio de trabajo. Se debe usar con precaución, ya que los cambios no guardados en otro lugar se perderán de forma irreversible.

Conceptos importantes:

- Directorio de trabajo (working directory): Es la carpeta del proyecto donde se encuentran los archivos que se editan directamente.
- Área de staging (staging area o index): Es un espacio intermedio en el que se seleccionan los cambios que se incluirán en el próximo commit. Se completa mediante el comando git add <archivo>.

34. ¿Qué es un "pull request" y cómo contribuye a la revisión de código en un equipo?

Un Pull Request (PR) es una funcionalidad de plataformas de control de versiones como GitHub, GitLab o Bitbucket, que permite a un desarrollador proponer cambios realizados en una rama de un repositorio para integrarlos en otra rama, generalmente la principal del proyecto.

Su principal función es servir como un mecanismo de colaboración y revisión de código dentro de un equipo de desarrollo. Cuando se abre un Pull Request, se presenta un resumen de los cambios propuestos, incluyendo commits, archivos modificados y diferencias. Otros miembros del equipo pueden revisar, comentar y discutir esos cambios. Además, se pueden ejecutar pruebas automáticas (CI/CD) para verificar que los cambios no afecten negativamente la aplicación. Tras la aprobación, los cambios se fusionan (merge) en la rama de destino.

Los beneficios de un Pull Request incluyen:

- **Revisión por pares (code review):** Permite que más de una persona revise el código antes de integrarlo, asegurando calidad y consistencia.
- **Discusión centralizada:** Facilita que todo el equipo vea, comente y debata los cambios, formalizando el proceso de revisión.
- **Detección de errores y mejoras:** Permite identificar fallos, vulnerabilidades o sugerir mejoras en legibilidad, rendimiento y arquitectura antes de la fusión.
- **Cumplimiento de estándares:** Garantiza que el código cumpla con las normas y buenas prácticas definidas en el proyecto.
- **Mantenimiento de un historial limpio y trazable:** Cada cambio queda documentado con autor, descripción y motivo, evitando que código problemático llegue a la rama principal.
- **Checkpoint de integración:** Actúa como un punto de control antes de fusionar cambios a la rama principal, asegurando que todo esté revisado y aprobado.

Un pull request no es solo una solicitud de fusión, sino una herramienta clave para la colaboración, el control de calidad y la comunicación en un equipo de desarrollo.

35. ¿Cómo puedes clonar un repositorio de GIT y cuál es la diferencia entre git clone y git pull?

Para clonar un repositorio de Git se utiliza el comando:

git clone <URL-del-repositorio>

Este comando crea una copia completa del repositorio remoto en tu máquina local, incluyendo todo el historial de commits, todas las ramas y etiquetas, y la configuración para mantener la conexión con el repositorio remoto (origin). Al ejecutarlo, Git también crea automáticamente una rama local que apunta a la rama por defecto del repositorio remoto.

La diferencia entre *git clone* y *git pull* radica en su propósito y momento de uso:

- **git clone:** Se utiliza una sola vez al inicio, para obtener una copia nueva del repositorio remoto. Descarga todo el historial completo y configura la conexión con el remoto, permitiendo comenzar a trabajar con el proyecto.

- **git pull:** Se utiliza de manera recurrente después de haber clonado el repositorio, para actualizar la copia local con los últimos cambios realizados en el remoto. Equivale a ejecutar git fetch seguido de git merge, sincronizando la rama local con los cambios remotos.

La diferencia principal es que git clone inicializa un proyecto en tu máquina desde cero, mientras que git pull lo mantiene sincronizado con el repositorio remoto.

Preguntas de PHP con Laravel

36. ¿Qué es Laravel?

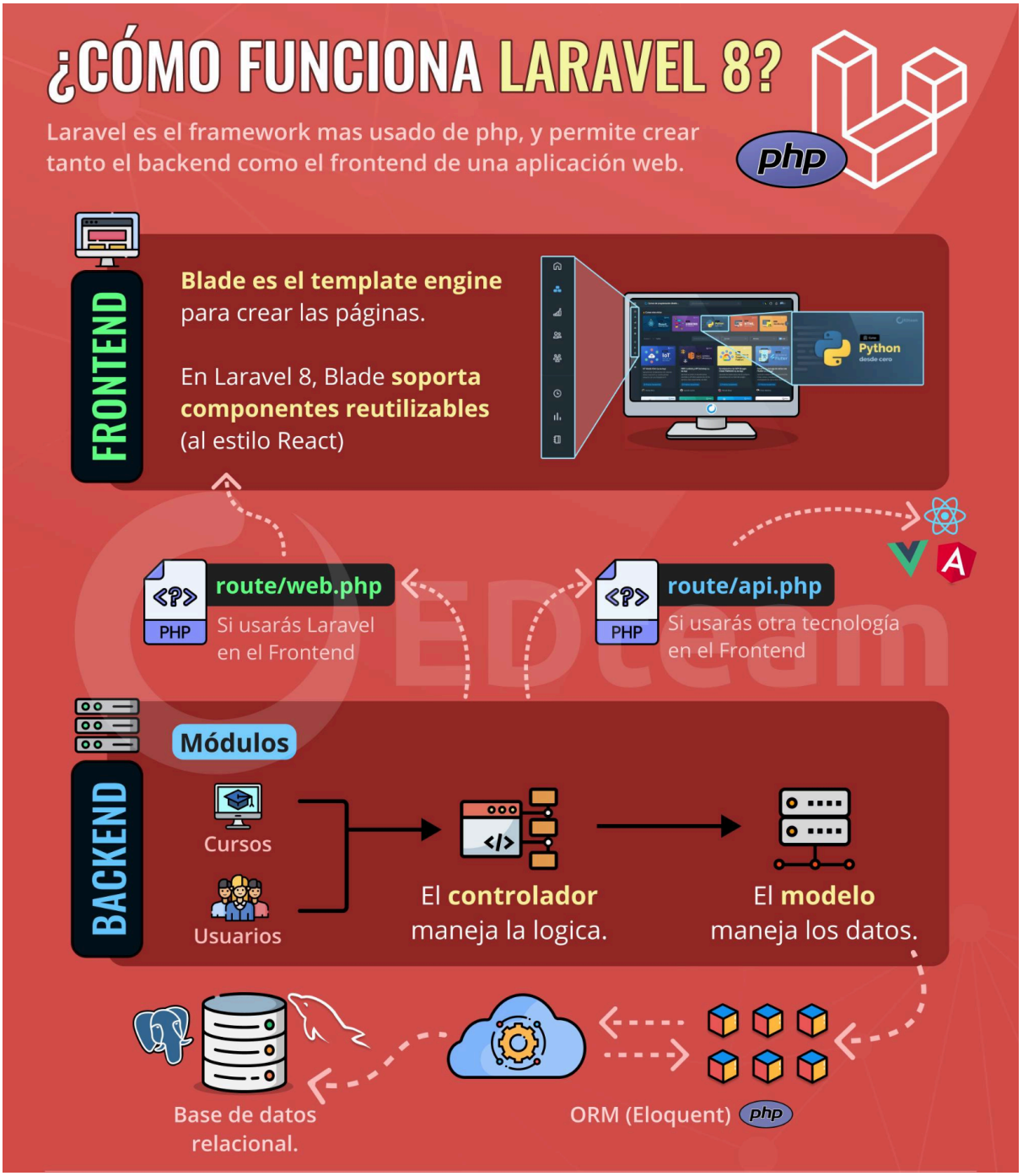
Un framework es un conjunto de herramientas, librerías y convenciones predefinidas que proporciona una base sólida para el desarrollo de software. Su función es ofrecer una estructura y componentes reutilizables que agilizan la programación. En lugar de escribir cada línea de código desde cero, un desarrollador utiliza el framework como un esqueleto para construir su aplicación.

Laravel es un framework de desarrollo web de código abierto, escrito en PHP, basado en el patrón de diseño MVC (Modelo-Vista-Controlador). Su objetivo principal es simplificar y acelerar la creación de aplicaciones web modernas, proporcionando una estructura organizada y herramientas listas para usar. Permite a los desarrolladores centrarse en la lógica de negocio, reduciendo el tiempo dedicado a tareas repetitivas y fomentando buenas prácticas.

Principales características y herramientas de Laravel:

- **Sistema de enrutamiento:** Maneja las peticiones HTTP y las asocia a funciones específicas del código.
- **ORM Eloquent:** Facilita el trabajo con bases de datos mediante objetos en lugar de SQL crudo, haciendo la interacción más intuitiva y orientada a objetos.
- **Motor de plantillas Blade:** Permite generar vistas dinámicas de manera sencilla, con HTML y sintaxis PHP limpia y reutilizable.
- **Migraciones de base de datos:** Versionan los cambios en el esquema de la base de datos para una gestión controlada de actualizaciones.

- **Seguridad integrada:** Protege la aplicación contra ataques comunes como inyección SQL, CSRF y XSS.
- **Herramientas adicionales:** Incluye autenticación y autorización, colas de trabajo, envío de correos y pruebas automatizadas.
- **Artisan:** Interfaz de línea de comandos que simplifica tareas de desarrollo como la gestión de migraciones, creación de controladores y ejecución de pruebas.



37. ¿Cómo maneja Laravel el modelo de ejecución de peticiones HTTP y en qué se diferencia del manejo tradicional en PHP puro?

Laravel utiliza un modelo centralizado de ejecución de peticiones HTTP basado en el patrón Front Controller, mientras que en PHP puro cada petición suele manejarse de manera descentralizada, ejecutando directamente archivos individuales. Esto implica diferencias significativas en organización, escalabilidad y mantenimiento del código.

Laravel abstrae el manejo de peticiones HTTP mediante un flujo centralizado y estructurado, que facilita la gestión de rutas, seguridad, lógica de negocio y generación de respuestas. En contraste, PHP puro ofrece un manejo directo y funcional, pero descentralizado, propenso a duplicación de código y difícil de escalar en proyectos grandes.

En Laravel, las peticiones HTTP se gestionan mediante un Front Controller, centralizando todo el flujo de ejecución:

- Todas las peticiones HTTP entran por public/index.php, que actúa como Front Controller. Este archivo inicializa la aplicación y el kernel HTTP de Laravel.
- El servidor web (Apache, Nginx) utiliza reglas de reescritura (ej. .htaccess) para que cualquier URL apunte a index.php.
- Laravel examina la URL y determina qué controlador o función debe manejar la petición, usando los archivos de rutas (routes/web.php para web, routes/api.php para APIs).
- Antes de llegar al controlador, la petición pasa por una cadena de middlewares, que pueden incluir autenticación, validación, seguridad, logging, entre otros. Esto permite aplicar reglas de manera centralizada y reutilizable.
- El controlador recibe la petición, interactúa con los modelos (base de datos mediante Eloquent) si es necesario y genera la respuesta.
- La respuesta se envía a la vista (Blade) para HTML o como JSON en APIs, pudiendo incluir también redirecciones, archivos o cualquier otro tipo de salida.

Diferencias en el modelo de peticiones HTTP entre Laravel y PHP puro:

En PHP puro, cada petición HTTP se maneja de forma descentralizada. Cada archivo .php actúa como un punto de entrada independiente; por ejemplo, login.php o productos.php se ejecutan directamente cuando se accede a su URL. No existe un flujo centralizado ni un enrutador global, por lo que la lógica de manejo de la petición debe escribirse en cada archivo, incluyendo validaciones, autenticación y preparación de la respuesta. Esto hace que el flujo de ejecución sea más rígido, menos flexible y difícil de mantener a medida que la aplicación crece.

En Laravel, en cambio, todas las peticiones pasan por un único punto de entrada, public/index.php, siguiendo el patrón Front Controller. Desde ahí, el enrutador analiza la URL y determina qué controlador o acción debe manejar la solicitud. Antes de llegar al controlador, la petición atraviesa una serie de middlewares, que aplican reglas de seguridad, autenticación y validación de manera centralizada. Luego, el controlador procesa la lógica de negocio, interactúa con los modelos si es necesario y prepara la respuesta, que puede ser HTML, JSON, archivos o redirecciones. Este modelo centralizado permite un flujo uniforme, fácil de mantener y extensible.

Cuadro comparativo entre Laravel y PHP puro:

Aspecto	PHP Puro	Laravel
Entrada de peticiones	Cada archivo .php puede ser accedido directamente (ej. login.php, productos.php)	Punto de entrada único: public/index.php
Ruteo	Basado en archivos físicos	Basado en rutas definidas que apuntan a controladores o closures
Middleware / seguridad	Implementación manual en cada script	Middleware centralizados, aplicables de manera global o por ruta
Arquitectura	Lógica y presentación mezcladas	MVC: separación clara entre Modelo, Vista y Controlador
Respuestas	Principalmente HTML	HTML, JSON, archivos, redirecciones, etc., de manera uniforme y flexible
Mantenimiento y escalabilidad	Difícil de mantener en proyectos grandes	Código organizado, escalable y reutilizable

38. ¿Qué es el ciclo de vida de una petición en Laravel y cuál es el rol del Kernel (HTTP Kernel) en dicho proceso?

El ciclo de vida de una petición en Laravel describe los pasos que sigue una solicitud HTTP desde que llega al servidor hasta que se envía la respuesta al cliente.

El flujo principal es el siguiente:

1. Entrada de la solicitud: Todas las peticiones entran por public/index.php, el punto de entrada único del framework. Este archivo recibe la solicitud y actúa como el primer punto de control, asegurando que todas las peticiones sigan un flujo centralizado y consistente.

2. Carga del framework: Antes de procesar la petición, Laravel carga el autoloader de Composer, inicializa las configuraciones de la aplicación y arranca los servicios esenciales. Esto incluye registrar proveedores de servicios, configurar conexiones a la base de datos y preparar cualquier componente necesario para el manejo de la solicitud.

3. HTTP Kernel: La solicitud se envía al HTTP Kernel (App\Http\Kernel), que centraliza el manejo de todas las peticiones HTTP. El Kernel se encarga de ejecutar los middlewares definidos, tanto globales como de grupo, que aplican reglas de

seguridad, autenticación, manejo de sesiones, CSRF, CORS, logging y otras validaciones antes de que la petición llegue al controlador.

4. Enrutamiento: Una vez que la petición ha pasado por los middlewares iniciales, el Kernel la entrega al router, que analiza la URL y determina qué controlador y método deben encargarse de procesar la solicitud según las rutas definidas en routes/web.php o routes/api.php.

5. **Controlador:** El controlador procesa la solicitud, interactúa con modelos, servicios o repositorios según sea necesario, y genera la respuesta que se enviará al cliente.

6. Respuesta y salida: La respuesta resultante (HTML, JSON, archivos, redirecciones, etc.) retorna al Kernel, que puede aplicar middlewares posteriores para modificar la salida si es necesario. Finalmente, la respuesta se envía al navegador o al cliente HTTP.

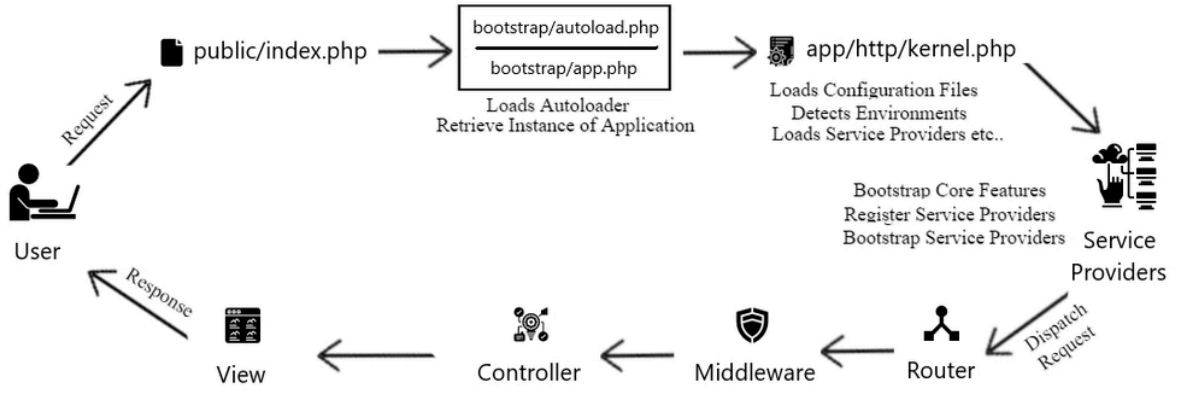
Rol del HTTP Kernel

El Kernel es el orquestador del ciclo de vida de la petición en Laravel. Es el intermediario principal que asegura que cada petición pase por todas las capas de seguridad y lógica antes de ser procesada y respondida, haciendo el ciclo de vida de la petición ordenado y controlado.

Su primer rol es garantizar que la solicitud entre en un flujo controlado, aplicando todas las reglas de seguridad y lógica de la aplicación antes de que llegue al controlador. Gracias al Kernel, los middlewares se ejecutan de manera centralizada, lo que permite manejar autenticación, sesiones, CSRF, CORS y otros aspectos de manera consistente y reutilizable.

Además, el Kernel administra el flujo de la solicitud hacia el router, asegurando que cada petición se asigne al controlador correcto según la ruta definida. Esto separa la lógica de ruteo de la lógica de negocio, haciendo que el código sea más ordenado y mantenible.

Por último, el Kernel también es responsable de procesar la respuesta final. Puede aplicar middlewares de salida que modifiquen o complementen la respuesta (por ejemplo, agregando cabeceras, compresión o logging) antes de enviarla al cliente. De esta manera, asegura que cada petición sea procesada de forma segura, consistente y predecible, cerrando el ciclo de vida de la solicitud de manera controlada.



39. ¿Cuál es la diferencia entre require/include de PHP y el sistema de autoloading de Composer en Laravel?

La diferencia principal entre require/include de PHP y el sistema de autoloading de Composer radica en la automatización de la carga de clases y la escalabilidad.

- En PHP puro, la carga de código se realiza mediante las funciones **require** e **include**, que permiten incluir archivos dentro de otros.

  Por ejemplo, se puede usar *require 'config.php'*; para asegurarse de que el archivo sea cargado, deteniendo la ejecución si no existe o tiene errores, o i*nclude 'funciones.php',* que emite un aviso si el archivo no se encuentra pero continúa la ejecución. Este método funciona correctamente en proyectos pequeños, pero a medida que la aplicación crece puede volverse complicado de mantener. Es necesario recordar manualmente la ruta exacta de cada archivo, y la gestión de clases, librerías y dependencias externas se vuelve engorrosa y propensa a errores.

- Laravel, en cambio, utiliza **Composer** para manejar automáticamente la carga de clases siguiendo el estándar PSR-4. Gracias al sistema de autoloading, no es necesario escribir require o include para cada clase que se use. Composer registra un autoloader que busca y carga la clase automáticamente desde la ruta correspondiente cuando es requerida en el código. Esto aplica tanto a las clases propias de la aplicación como a librerías externas instaladas mediante Composer.

  Por ejemplo, si se utiliza *use App\Models\User*; y luego se llama a *User::find(1)*;, no hace falta incluir manualmente *User.php*; Composer se encarga de cargar la clase bajo demanda.

Mientras que require/include es manual y explícito, obligando al desarrollador a incluir cada archivo necesario, el autoloading es automático. Al definir en composer.json la estructura de namespaces y carpetas, Composer genera un archivo vendor/autoload.php que basta con incluir una sola vez en la aplicación, normalmente en public/index.php en Laravel. A partir de ese momento, cualquier clase utilizada se carga automáticamente según el namespace y la ubicación configurada, sin necesidad de referencias manuales.

Require/include es manual y propenso a errores, mientras que Composer autoloading automatiza la carga de clases y dependencias, haciendo el proyecto más limpio y mantenible.

Funcionamiento del Autoloading de Composer

El sistema de autoloading de Composer se basa en un archivo composer.json y el comando composer dump-autoload. El proceso es el siguiente:

1. En primer lugar, se realiza la **configuración del autoloading** a través del archivo *composer.json*. En este archivo se definen las reglas de autoloading, estableciendo la correspondencia entre los namespaces y las carpetas donde se encuentran las clases. El estándar más utilizado es PSR-4, que permite mapear de manera sistemática cada namespace a una ruta específica dentro del proyecto.
2. A continuación, se procede a la **generación del autoloader**. Al ejecutar comandos como *composer install* o *composer dump-autoload*, Composer analiza las rutas y namespaces definidos, y genera un archivo central de autoloading ubicado en *vendor/autoload.php*. Este archivo contiene toda la información necesaria para localizar y cargar las clases de forma automática cuando se utilicen en la aplicación.
3. Posteriormente, se realiza la **carga inicial del autoloader**. En la aplicación Laravel, generalmente se incluye este archivo una sola vez en el punto de entrada del framework, que es *public/index.php*, mediante la instrucción *require **DIR**.'/../vendor/autoload.php';*. Esto garantiza que el sistema de autoloading esté disponible durante toda la ejecución de la aplicación.
4. Finalmente, se produce la **carga bajo demanda de las clases**. Cada vez que el código hace referencia a una clase, por ejemplo al instanciar *new App\Http\Controllers\HomeController*, el autoloader de Composer busca automáticamente la clase correspondiente en la ubicación definida por el namespace y la carga en memoria sin requerir llamadas manuales adicionales. De esta manera, se asegura una gestión eficiente, ordenada y escalable de todas las dependencias y clases del proyecto.

Cuadro comparativo:

Característica	require/include	Autoloading de Composer
Carga de archivos	Manual, cada archivo debe ser incluido explícitamente	Automática, basada en nombres de clases y namespaces
Mantenimiento	Difícil en proyectos grandes	Escalable y organizado
Estándares	No requiere un estándar específico	Sigue PSR-4, facilita interoperabilidad
Configuración	No requiere configuración	Se configura en composer.json
Librerías externas	Deben ser incluidas manualmente	Composer maneja todas las dependencias automáticamente

40. ¿Qué es Composer y cuál es su función dentro del ecosistema de PHP y Laravel?

Composer es un gestor de dependencias para PHP que simplifica la instalación, actualización y carga de librerías externas necesarias para un proyecto. Su función principal es automatizar la gestión de paquetes, evitando la necesidad de descargar y administrar manualmente cada librería.

Dentro del ecosistema de PHP y, en particular, de Laravel, Composer cumple varias funciones esenciales.

- En primer lugar, se encarga de la **gestión de dependencias**, leyendo el archivo de configuración *composer.json*, donde se especifican las librerías requeridas. Al ejecutar *composer install*, Composer descarga las versiones correctas de esas librerías y las coloca en la carpeta *vendor/*, asegurando compatibilidad entre ellas y con el proyecto.
- Otra función clave es el **autoloading automático**. Composer genera un archivo único, *vendor/autoload.php*, que permite cargar automáticamente todas las clases del proyecto y de las librerías instaladas, evitando el uso manual de *require* o *include*. Esto hace que el código sea más limpio, organizado y fácil de mantener.
- Composer también facilita la **actualización de paquetes** mediante el comando *composer update*, lo que permite mantener las librerías al día sin comprometer la estabilidad del proyecto. Además, la herramienta asegura que las versiones de las **dependencias sean compatibles entre sí**, evitando conflictos y problemas de compatibilidad.
- Un beneficio adicional muy relevante es la **integración con Laravel**. Todos los paquetes del framework, como Sanctum, Horizon o Telescope, se instalan y gestionan a través de Composer, lo que permite que Laravel sea modular y extensible. Esto facilita la incorporación de nuevas funcionalidades sin alterar la estructura central del framework y mantiene la aplicación organizada y escalable.

41. ¿Cómo se inicializa un nuevo proyecto de Laravel usando Composer y cuál es el propósito del archivo composer.json?

Para inicializar un nuevo proyecto de Laravel usando Composer, se utiliza el comando *create-project*, que se encarga de descargar el framework y configurar todo lo necesario de manera automática. El comando específico es:

composer create-project --prefer-dist laravel/laravel nombre-del-proyecto

En este comando, la opción *--prefer-dist* indica que Composer descargará la versión distribuida del proyecto, lo que suele ser más rápido que clonar el repositorio completo, y *nombre-del-proyecto* es la carpeta donde se instalará Laravel. Al ejecutarlo, Composer realiza varias acciones importantes:

- crea un nuevo directorio con el nombre especificado,
- descarga el paquete *laravel/laravel* desde Packagist,
- instala todas las dependencias necesarias del framework en el directorio *vendor/*,
- configura el autoloader de Composer,

- genera la estructura de carpetas estándar de Laravel, incluyendo directorios como app/, routes/, database/ y resources/.
- y genera un archivo .env que contiene las variables de entorno para la configuración de la aplicación.

El comando *composer create-project* se basa en el *composer.json* del paquete que se va a instalar para inicializar un proyecto. Este archivo define todas las dependencias necesarias, sus versiones compatibles, la configuración del autoloading y cualquier script de instalación, de manera que Composer pueda descargar e instalar automáticamente las librerías, generar el autoloader y preparar la estructura del proyecto, dejando la aplicación lista para usar sin intervención manual.

Propósito del Composer.json

El archivo composer.json es la guía de dependencias y configuración del proyecto. Es un archivo en formato JSON que actúa como manifiesto de dependencias y configuración, y sus principales propósitos son:

- **Declarar dependencias y versiones compatibles:** Este archivo indica qué paquetes necesita el proyecto y qué versiones son compatibles entre sí. Gracias a esto, Composer puede instalar y mantener automáticamente todas las librerías necesarias, evitando conflictos de versiones y garantizando que todos los desarrolladores trabajen con el mismo conjunto de dependencias.

- **Configurar el autoloading PSR-4:** El *composer.json* define cómo se cargan automáticamente las clases del proyecto según sus namespaces y rutas. Esto permite que Composer genere un autoloader que carga las clases bajo demanda, eliminando la necesidad de usar *require* o *include* manualmente, y manteniendo el código limpio y organizado.

- **Almacenar información básica del proyecto:** Se puede incluir el nombre del proyecto, su descripción, los autores y otra información relevante. Esto facilita la identificación del proyecto y sirve como referencia para otros desarrolladores o herramientas que interactúan con el proyecto.

- **Definir scripts personalizados:** El archivo permite definir comandos automatizados que se ejecutan con Composer, como migraciones de base de datos, pruebas unitarias o tareas de mantenimiento. Esto ayuda a estandarizar procesos y reducir errores manuales durante el desarrollo o despliegue.

- **Especificar requisitos mínimos de PHP y extensiones:** Composer utiliza esta información para asegurarse de que el proyecto se ejecute en un entorno compatible. Esto evita problemas de compatibilidad y asegura que las dependencias funcionen correctamente con la versión de PHP instalada.

42. ¿Qué son las dependencias en Composer y cómo se instalan? Explica la diferencia entre dependencias normales y dependencias de desarrollo.

En Composer, las dependencias son librerías, paquetes o frameworks externos que un proyecto necesita para funcionar correctamente. Estas pueden incluir desde frameworks y clientes HTTP hasta herramientas de seguridad, librerías de validación y paquetes de pruebas.

La gestión de dependencias con Composer permite instalarlas, actualizarlas y cargarlas automáticamente, evitando la inclusión manual de archivos y asegurando que todos los desarrolladores del proyecto trabajen con versiones consistentes de cada paquete.

- Para instalar **dependencias normales**, necesarias para el funcionamiento de la aplicación en producción, se utiliza el comando:
    - *composer require nombre/paquete*.
    - Esto agrega el paquete al proyecto y lo registra en la sección *require* del archivo *composer.json*.
- Por otro lado, las **dependencias de desarrollo**, que solo se necesitan durante el desarrollo para tareas como pruebas, depuración o análisis de código, se instalan con
    - *composer require nombre/paquete --dev*.
    - Se registran en la sección *require-dev* del *composer.json*. Estas dependencias no son necesarias en producción y se pueden omitir al desplegar el proyecto mediante el comando *composer install --no-dev*.

Cuando se ejecuta *composer install*, Composer lee el *composer.json*, descarga las dependencias necesarias desde Packagist y las coloca en el directorio *vendor/* del proyecto. Además, genera o actualiza el archivo *vendor/autoload.php*, que permite cargar automáticamente todas las clases de las librerías instaladas sin necesidad de usar *require* o *include* manualmente.

La diferencia principal entre los tipos de dependencias en Composer se basa en el entorno en el que son requeridas:

- Las **dependencias normales** (listadas en la sección *require* del *composer.json*) corresponden a los paquetes indispensables para el funcionamiento de la aplicación en un entorno de producción. Esto incluye el framework Laravel, el ORM Eloquent, el motor de plantillas Blade y cualquier otra librería necesaria para que la aplicación opere correctamente en vivo. Estas dependencias se instalan automáticamente al ejecutar *composer install*.
- Por su parte, las **dependencias de desarrollo** (registradas en la sección *require-dev* del *composer.json*) son aquellas necesarias únicamente durante el desarrollo y las pruebas, y no resultan imprescindibles para el entorno de producción.

Esta distinción permite, al momento de desplegar la aplicación en un entorno de producción, instalar únicamente las dependencias normales, manteniendo el paquete de la aplicación más ligero y reduciendo potenciales riesgos de seguridad. Para este fin, se puede utilizar el comando *composer install --no-dev*, que omite la instalación de las dependencias de desarrollo.

Cuadro comparativo entre dependencias nromales y de desarrollo

Característica	Dependencias Normales (*require*)	Dependencias de Desarrollo (*require-dev*)
Propósito	Son esenciales para el funcionamiento de la aplicación en producción.	Solo se utilizan durante el desarrollo y las pruebas; no son necesarias en producción.
Ejemplos	Laravel, Eloquent, Blade, paquetes de funcionamiento en vivo.	PHPUnit, Xdebug, Faker, herramientas de análisis de código.
Ubicación en composer.json	Se listan en la sección *require*.	Se listan en la sección *require-dev*.
Instalación en producción	Siempre se instalan al ejecutar *composer install*.	Se pueden omitir al ejecutar *composer install --no-dev*.
Impacto en el despliegue	Necesarias para que la aplicación funcione correctamente en producción.	No afectan la funcionalidad en producción; ayudan a mantener ligero el paquete y reducir riesgos.

43. ¿Cómo puedes gestionar versiones específicas de paquetes en Composer y cuál es el propósito del archivo composer.lock?

En Composer, las versiones de los paquetes se gestionan mediante restricciones definidas en el archivo *composer.json*. Esto permite especificar exactamente qué versiones de una librería se desean instalar o establecer rangos compatibles para evitar problemas por actualizaciones inesperadas.

Por ejemplo, al ejecutar *composer require guzzlehttp/guzzle:^7.5*, se solicita que Composer instale cualquier versión compatible con la serie 7.x, asegurando compatibilidad y acceso a parches de seguridad.

Ejemplos de restricciones de versión

Sintaxis	Significado
"^1.2"	Compatible con la versión 1.2 y todas las actualizaciones menores (1.2.x, 1.3.x), pero no con 2.0
"~1.2"	Compatible con 1.2 y todas las actualizaciones de patch, hasta la próxima versión mayor (1.2.x, 1.3.x)
"1.2.3"	Una versión exacta
">=1.2 <2.0"	Rango específico de versiones
"*"	Cualquier versión disponible

El archivo *composer.lock* complementa esta gestión de versiones. Se genera automáticamente la primera vez que se ejecuta composer install y registra:

- La **versión exacta** de cada paquete instalado.
- El **hash de integridad** de cada paquete.
- La información sobre dependencias de las dependencias (subdependencias).

Su propósito principal es garantizar consistencia y reproducibilidad: cuando cualquier miembro del equipo ejecuta *composer install*, Composer utiliza las versiones exactas registradas en *composer.lock*, ignorando los rangos definidos en *composer.json*. Esto asegura que todos los entornos —desarrollo, staging o producción— utilicen las mismas versiones, evitando errores inesperados y manteniendo la estabilidad y seguridad del proyecto.

Composer.json es el contrato de tu proyecto, donde defines las versiones que necesitas de forma flexible.

Composer.lock es la ejecución de ese contrato, que congela las versiones exactas para garantizar la coherencia en todo el ciclo de vida del desarrollo.

44. ¿Qué es Eloquent ORM y cómo facilita la interacción con bases de datos en Laravel?

Un ORM, o Mapeo Objeto-Relacional, es una técnica de programación que actúa como un "traductor" entre el código orientado a objetos de una aplicación y las tablas de una base de datos relacional. Su función principal es permitir a los desarrolladores interactuar con la base de datos utilizando objetos del lenguaje de programación en lugar de escribir sentencias SQL directamente

Eloquent ORM es el *Object-Relational Mapper* incluido en Laravel, cuyo propósito es facilitar la interacción con bases de datos mediante un enfoque orientado a objetos. En lugar de escribir sentencias SQL de forma manual, permite a los desarrolladores trabajar con modelos en PHP que representan tablas, y con objetos que representan los registros de

dichas tablas. Esto permite realizar operaciones de creación, lectura, actualización y eliminación (CRUD) de manera más intuitiva y expresiva.

Entre sus principales funcionalidades se destacan:

- **Mapeo de tablas a modelos**: cada tabla se vincula a una clase en PHP (modelo), y cada fila de la tabla se representa como una instancia de dicho modelo.
- **Sintaxis sencilla para operaciones CRUD**: por ejemplo, es posible obtener un registro con *User::find(1)*, modificar sus atributos como propiedades de objeto y persistir los cambios con *save()*.
- **Consultas avanzadas con métodos encadenables**: Eloquent permite construir consultas complejas usando una sintaxis fluida y legible, como *User::where('active', 1)->orderBy('created_at', 'desc')->get()*.
- **Gestión de relaciones entre tablas**: soporta relaciones como *hasOne*, *hasMany*, *belongsTo* y *belongsToMany*, facilitando el acceso a datos relacionados de manera natural, por ejemplo *$post->comments*.
- **Seguridad integrada**: todas las consultas se parametrizan automáticamente, lo que protege contra ataques de inyección SQL.
- **Integración con migraciones y Query Builder**: permite definir y versionar esquemas de base de datos, así como construir consultas más complejas cuando se requiera mayor flexibilidad.

Eloquent ORM actúa como un puente entre el código PHP y la base de datos, simplificando el acceso a la base de datos, organizando las operaciones en torno a modelos bien definidos y protegiendo contra errores comunes.

Ejemplos de sintaxis en Elocuent:

- **CRUD sencillo y expresivo:**

```
$user = User::find(1);                                          // Leer
$user->name = 'Juan';                                           // Actualizar
$user->save();                                                  // Guardar cambios
$user->delete();                                                // Eliminar
User::create(['name' => 'Ana', 'email' => 'ana@mail.com']); // Crear
```

- **Consultas avanzadas con métodos encadenables:**

```
$activeUsers = User::where('active', 1)
    ->orderBy('created_at', 'desc')
    ->get();
```

- **Relaciones entre tablas:**
  - hasOne,
  - hasMany,
  - belongsTo,
  - belongsToMany

45. ¿Cómo se manejan errores en Laravel? Explica el rol del Handler, el uso de excepciones personalizadas y cómo se combinan con middlewares y validaciones.

En Laravel, el manejo de errores es un proceso centralizado y robusto que te permite gestionar, registrar y mostrar los errores de manera consistente. La arquitectura de manejo de errores del framework se basa en el Handler (Manejador de Excepciones), el uso de excepciones personalizadas, y la integración con middlewares y validaciones.

1. El **Handler**, ubicado en app/Exceptions/Handler.php, es el centro del sistema de manejo de errores. Cada vez que ocurre una excepción, Laravel la envía al Handler, que se encarga de dos tareas principales:
    a. **Registrar**: A través del método *report*(), el Handler registra o reporta los errores, ya sea en los logs de Laravel o en servicios externos como Sentry o Bugsnag
    b. **Renderizar**: mediante el método *render*(), transforma esa excepción en una respuesta adecuada para el cliente: puede devolver una vista HTML, un mensaje JSON en el caso de una API, o incluso redirigir a otra ruta.

    Además, el Handler se puede personalizar para manejar distintos tipos de errores de forma diferenciada, como mostrar una página especial para un 404 o devolver un mensaje claro en caso de un error de permisos.

2. Por otro lado, Laravel permite la creación de **excepciones personalizadas**, que ayudan a dar mayor claridad y semántica al código. Estas excepciones se definen como clases que heredan de Exception, y dentro de ellas se puede incluso definir cómo deben ser renderizadas.

    Por ejemplo, una excepción llamada UsuarioNoEncontradoException puede lanzarse cuando un usuario buscado no existe en la base de datos, y dentro de su método render() puede devolverse directamente un JSON con un código de estado 404 y un mensaje claro para el cliente. De esta forma, la aplicación no solo resulta más legible, sino que también se vuelve más sencilla de mantener y depurar.

3. El sistema de excepciones de Laravel también se combina de manera natural con los **middlewares y las validaciones**.
    a. Los **middlewares** actúan como una primera línea de defensa: antes de que una petición llegue al controlador, pueden evaluar condiciones como si el usuario está autenticado, autorizado o si proviene de una IP permitida. Si no se cumplen estas condiciones, el middleware puede lanzar una excepción que será capturada por el Handler.
    b. Por su parte, las **validaciones**, ya sea en formularios o APIs, también generan automáticamente excepciones. Cuando los datos enviados por el cliente no cumplen con las reglas definidas, Laravel lanza una ValidationException, que luego el Handler transforma en una respuesta clara y estructurada que detalla los errores de validación.

El flujo de manejo de errores en Laravel puede verse como una cadena organizada. Primero, los **middlewares** filtran las peticiones inválidas. Luego, dentro del controlador o servicios, se ejecuta la lógica principal, y si ocurre un problema, se pueden lanzar **excepciones personalizadas**. Finalmente, el **Handler** centraliza todo este proceso, registrando los errores y devolviendo una respuesta coherente según el contexto (ya sea una vista en una aplicación web o un JSON en una API).

Este enfoque tiene varias ventajas:

- La centralización asegura que los errores se manejen de manera uniforme.
- La consistencia evita que las respuestas varíen según el tipo de cliente.
- La seguridad impide exponer información sensible en los mensajes de error.
- La flexibilidad permite personalizar el tratamiento de cada excepción según las necesidades de la aplicación.

Práctica

Para realizar los ejercicios prácticos deberás contar en tu ambiente de trabajo con las siguientes herramientas:

- Postman
- GIT
- Node.js instalado

La entrega deberá realizarse en un repositorio de código público que se deberá compartir al equipo de reclutamiento. El repositorio deberá contener tanto la parte teórica como la práctica

Actividad práctica número 1
Pasos:

1) Realizar una petición GET a la siguiente URL a través de Postman:
https://reclutamiento-dev-procontacto-default-rtdb.firebaseio.com/reclutier.json

Ejemplo con CURL:

curl --location --request GET
'https://reclutamiento-dev-procontacto-default-rtdb.firebaseio.com/reclutier.json' \
--header 'Content-Type: application/json'

2) Realizar una petición POST a la siguiente URL a través de Postman:
https://reclutamiento-dev-procontacto-default-rtdb.firebaseio.com/reclutier.json
y con el siguiente body:

```
{
   "name":"TuNombre",
   "suraname":"TuApellido",
   "birthday":"1995/11/16",
   "age":29,
   "documentType":"CUIT",
   "documentNumber":20123456781
}
```

Reemplazar los campos por los valores personales tuyos.

Resuelto en el siguiente repositorio:
https://github.com/LuisDiNicco/TP-PHP-con-Laravel---Di-Nicco-Luis-Demetrio/tree/main/Resolucion/Actividad%20Practica%201

Actividad práctica número 2

Construir una mini-API en Laravel que:

A. Consuma por **GET** el recurso

`https://reclutamiento-dev-procontacto-default-rtdb.firebaseio.com/reclutier.json`
y lo presente de forma legible para humanos.

B. Exponga un **POST /recluta** en PHP con Laravel:

```
{
  "name":"TuNombre",
  "suraname":"TuApellido",
  "birthday":"1995/11/16",
  "documentType":"CUIT",
  "documentNumber":20123456781
}
```

y lo **mapee** a:

```
{
  "name":"TuNombre",
  "suraname":"TuApellido",
  "birthday":"1995/11/16/",
  "age":29,
  "documentType":"CUIT",
  "documentNumber":20123456781
}
```

para luego **POST**earlo a

`https://reclutamiento-dev-procontacto-default-rtdb.firebaseio.com/reclutier.json`

Reglas de negocio y validaciones

- `name` y `suraname`: convertir a “Title Case” (primera letra de cada palabra en mayúscula). Si llegan en otro formato, **normalizar**.
- `birthday`: formato **YYYY/MM/DD**. No puede ser posterior a hoy ni anterior a **1900/01/01**. Si no cumple, **rechazar** (400 con detalle).
- `documentType`: sólo **“CUIT”** o **“DNI”**; otros valores → **rechazar**.

- `age`: calcular a partir de `birthday` (años cumplidos al día de la solicitud).

- La salida enviada a Firebase deberá incluir `birthday` **con una barra final** (`.../`) para coincidir con el ejemplo provisto.

Resuelto en el siguiente repositorio:
https://github.com/LuisDiNicco/TP-PHP-con-Laravel---Di-Nicco-Luis-Demetrio/tree/main/Resolucion/Actividad%20Practica%202