DevOps:

Cultura de valor

Definición

En inglés **Minimun Viable Product (MVP),** es una entrega del producto que nos permite validar u obtener información con el usuario real sobre nuestras hipótesis. Nos invita al pensamiento: “si no nos avergonzamos de la primera versión de nuestro producto es que hemos salido tarde”

PROPÓSITOS DE UN PMV

- Reunir las características mínimas para **validar las necesidades de los clientes finales**, siendo la base para futuras versiones que incluyan más funcionalidades.
- Un PMV tiene también como objetivo **reducir al mínimo el tiempo de lanzamiento al mercado** (en inglés **Time To Market).**
- Comprobar si el **producto** tiene **garantías de futuro en el mercado**, teniendo éxito entre los usuarios finales.
- A **nivel técnico,** tiene los siguientes objetivos:
  - Comprobar la **validez del equipo técnico.**
  - **Reducir la sobreingeniería** y horas de desarrollo.
  - **Identificar los evolutivos** y nuevas versiones de los artefactos software.
  - **No perder el foco** y objetivos marcados como alcance.
  - **Ser capaces de identificar** desviaciones en la planificación y poder reaccionar a tiempo.

ENFOQUES DE UN PMV

El PMV puede tomar dos vertientes:

- **Prueba de concepto**, es un esfuerzo del equipo en probar unas hipótesis o adquirir unos conocimientos necesarios antes de avanzar en el producto (rápido y “barato”). Puede ser algo tan sencillo como crear una página web «tonta» que presenta información del producto para ver métricas de leads que tengan potencialmente interés. No necesariamente tiene que haber funcionalidad construida
- **Producto mínimo viable**, que es cuando se libera un mínimo de funcionalidad construida (una unidad “comercializable” sea de pago o no). Puede ser una versión muy reducida que busca explorar sobre qué características tenemos que trabajar porque son las que más aceptación tienen entre nuestros clientes.

Más información en el siguiente enlace.

Definición

En inglés **User Story (US),** su función es **definir una necesidad** del usuario mediante el uso de un **lenguaje comprensible** para cualquier persona que la lea, disponga o no del contexto de la misma. Es la **unidad mínima** de funcionalidad que **aporta por sí misma valor** al usuario.

ESTRUCTURA

Una historia de usuario debe cumplir con el patrón de las 3 Ces, es decir debe estar compuesta por:

- **Card** - Título claro y suficientemente descriptivo (p.e. **COMO usuario QUIERO poder registrarme PARA poder hacer login en el sistema)**
- **Conversation** - Es mucho más importante que la propia Card y recoge los detalles.
- **Confirmation** - Recoge de forma transparente los criterios para considerar esa funcionalidad finalizada.

LA CONFIRMACIÓN

La confirmación debe contar con unos **criterios claros** de aceptación y es recomendable que contemple al menos un caso de **éxito**, un caso de **fallo** y un caso de **error**.

Podemos ayudarnos del patrón **Given-When-Then**:

DADO QUE **[ESCENARIO]** CUANDO **[ACCIÓN]** ENTONCES **[RESULTADO]**.

¿CÓMO RECONOCER UNA BUENA HISTORIA?

SMART:

- **Specific** - Ser comprensible, no abstracta y reflejar claramente su propósito.
- **Measurable** - Poder medirse para ver si cumple su objetivo.
- **Achievable** - Ser realista, alcanzable y/o realizable.
- **Relevant** - Contribuir de forma relevante al producto o servicio.
- **Time-boxed** - Ser realizada en un plazo máximo de tiempo.

INVEST:

- **Independent** - No debe depender de otras historias.
- **Negotiable** - No es un contrato, y pueden cambiar a lo largo del tiempo.
- **Valuable** - Aportar valor por sí misma al usuario.
- **Estimable** - Permitirnos estimar el esfuerzo de realizarla.
- **Small** - Su tamaño debe ser lo más pequeño posible, de esta manera podremos ir cerrando los avances dentro de las iteraciones.
- **Testeable** - Debe contar con unos criterios de prueba que permitan comprobar que se comporta de la forma esperada.

¿Qué es?

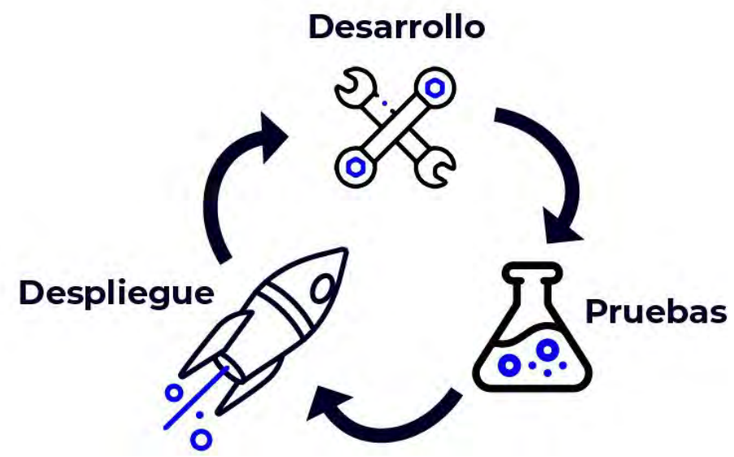
En inglés **Continuous Integration (CI)**, es la práctica que tiene como objetivo integrar los cambios en el repositorio central de forma periódica a través de varios de procesos automatizados donde cada versión generada se comprueba mediante pruebas y tests para detectar posibles errores de forma temprana.

¿EN QUÉ CONSISTE?

Durante el proceso de integración continua se pasan ciertas fases o tareas a ejecutar como por ejemplo instalar las dependencias necesarias, lanzar los tests, entre otras. En caso de que alguien del equipo decida hacer un Pull Request/Merge Request, comprobamos que dicho pipeline ha pasado correctamente y procedemos a integrar dichos cambios ya sea en la rama principal o en nuestra propia rama local en caso de necesitarlos.

¿Qué beneficios nos aporta?

- **Detección rápida de fallos** de forma continua.
- **Aumento de la productividad del equipo**.
- **Automatización** y ejecución inmediata de procesos.
- **Monitorización** continua de las métricas de calidad del proyecto.



Debemos tener en cuenta:

- **No dejar pasar más de dos horas sin integrar los cambios** que hemos programado.
- La programación en equipo es un problema de **“divide, vencerás e integrarás”.**
- **La integración es un paso no predecible** que puede costar más que el propio desarrollo.
- **Integración síncrona:** cada pareja después de un par de horas sube sus cambios y espera a que se complete el build y se hayan pasado todas las pruebas sin ningún problema de regresión.
- **Integración asíncrona**: cada noche se hace un build diario en el que se construye la nueva versión del sistema. Si se producen errores se notifica con alertas de emails.
- **El sistema resultante debe ser un sistema listo para lanzarse**.

¿Qué es?

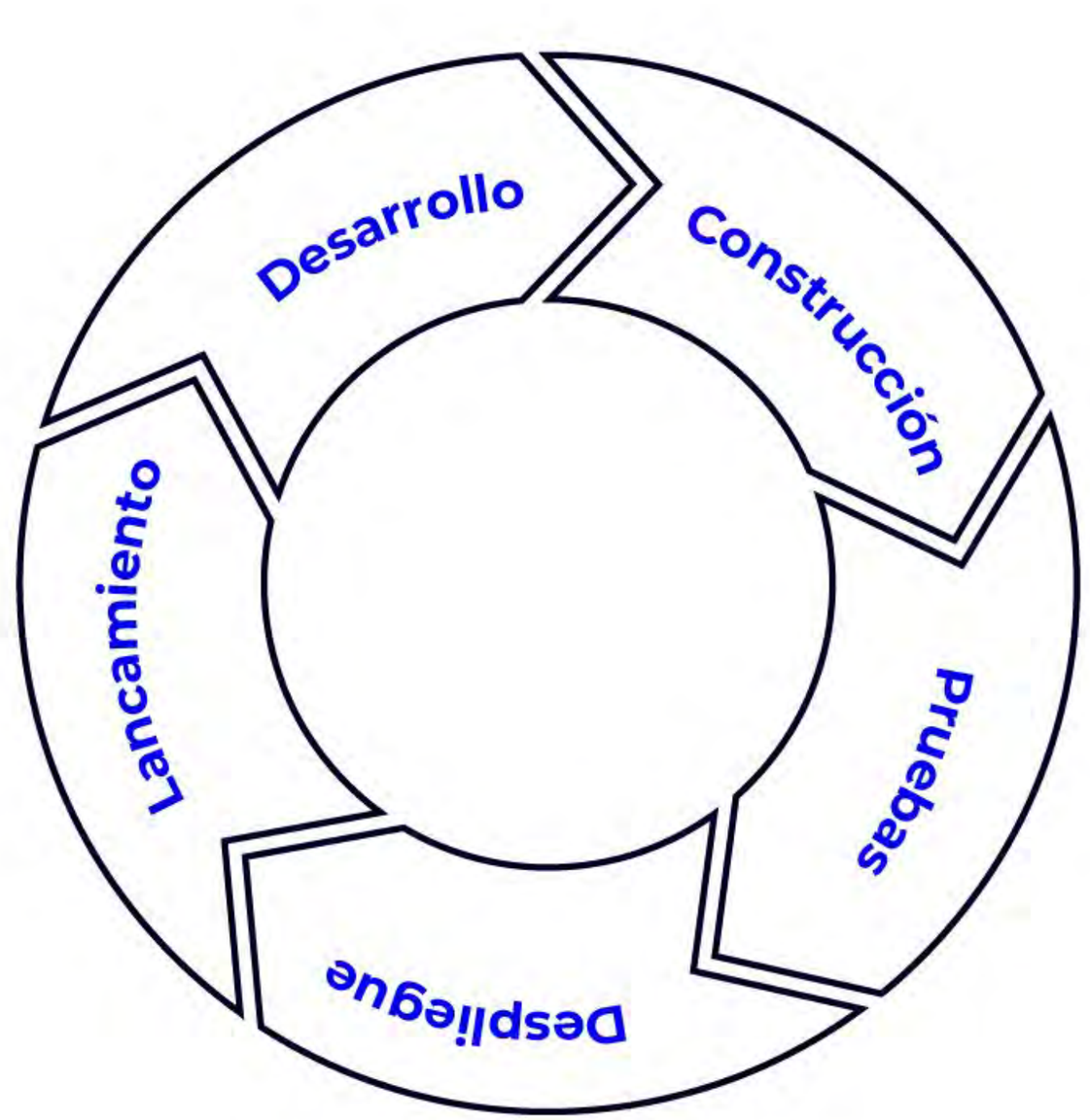
En inglés **Continuous Deployment (CD),** es una práctica que tiene como objetivo proporcionar una manera ágil, fiable y **automática** de poder entregar o desplegar los nuevos cambios en el entorno específico, normalmente producción. Suele utilizarse junto con la integración contínua.

¿EN QUÉ CONSISTE?

El despliegue continuo se basa en automatizar todo el proceso de despliegue de la aplicación en cualquiera de los entornos disponible, ya sea sólo en algunos de ellos o en todos, todo ello sin que haya **ninguna intervención humana** en el procedimiento.

El objetivo es hacer **despliegues predecibles, automáticos y rutinarios** en cualquier momento, ya sea de un sistema distribuido a gran escala, un entorno de producción complejo, un sistema integrado o una aplicación.

Dependiendo de cada proyecto o propósito de negocio de la empresa, el despliegue estará configurado para hacerse de forma semanal, quincenal o cada 2 o 3 días, aunque el objetivo es hacerlo de manera constante y en periodos cortos para obtener feedback rápido del cliente.



Sistema de control de versiones centralizado

autentia

¿Qué es?

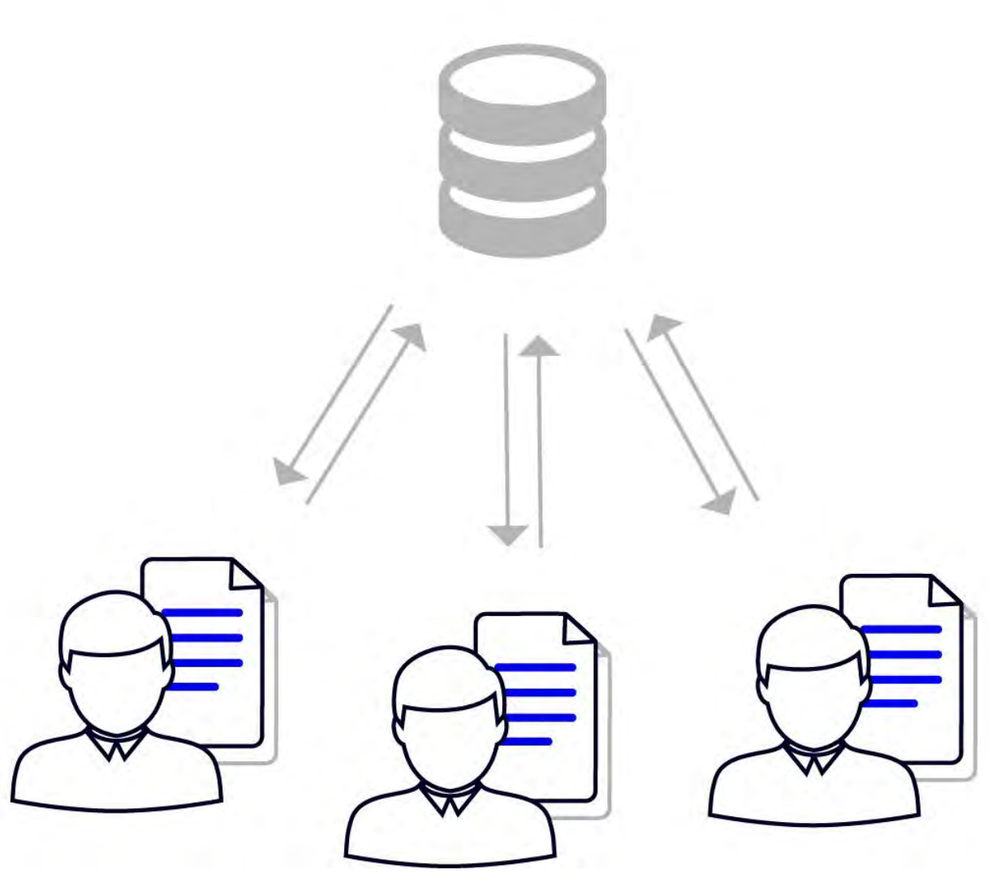
En inglés **Centralized Version Control System (CVCS)**, es un sistema que permite a los usuarios trabajar en un proyecto común compartido a través de un **único servidor central** que funciona como un punto de sincronización común.

¿EN QUÉ CONSISTE?

Los sistemas de control de versiones centralizados nacieron para reemplazar a los sistemas de control de versiones locales. Estos almacenaban los cambios y versiones en el disco duro de los desarrolladores.

Se quería solucionar el problema que se encuentran las personas que necesitan colaborar con desarrolladores en otros sistemas. **CVCS se basa en tener un único servicio o repositorio central** que está situado en una máquina concreta y contiene todo el histórico, etiquetas, ramas del proyecto etc. Sin embargo, esta configuración tiene muchas desventajas ya que se depende exclusivamente del servidor central. En caso de caída de dicho servidor, nadie podría trabajar y si era un proyecto muy grande con muchos contribuyentes, el número de conflictos diarios podría ser muy elevado.

Los CVCS más populares son Subversion o CVS, aunque **hoy en día muchos han migrado a los sistemas de control de versiones distribuidos** como Git o Mercurial**.**



¿Qué es?

Un proyecto de código abierto **(Open Source)** se publica bajo una licencia que define los términos para que otros usuarios puedan modificar, descargar o incluso redistribuir su propia versión. Es una práctica muy utilizada en la industria del software para que la comunidad aporte valor y contribuya a la mejora del código.

VENTAJAS

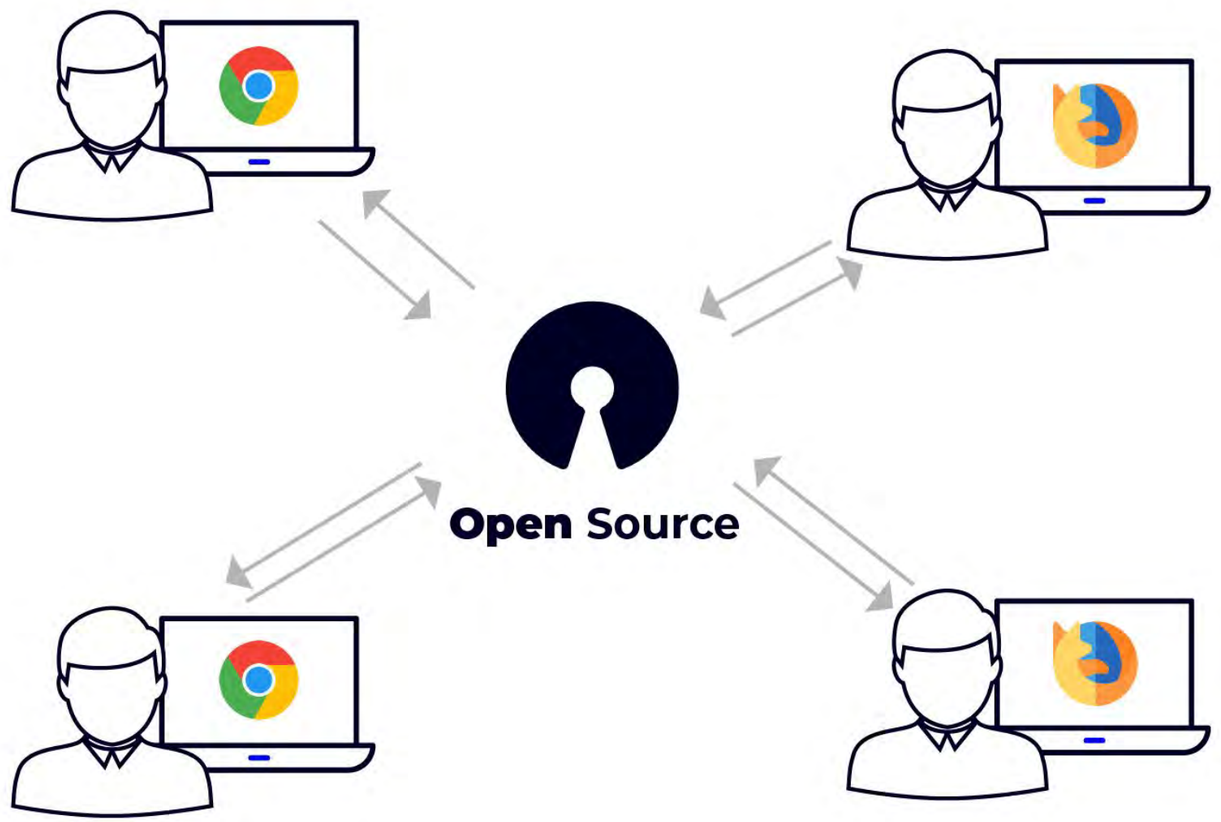
Feedback: soporte continuo para apoyar y mejorar una solución que beneficia tanto a la empresa como a la comunidad aumentando la fiabilidad en cada fase del desarrollo gracias al respaldo de todos los colaboradores.

Transparencia: brinda una mayor confianza a los usuarios finales saber qué se está desarrollando y cómo avanza el proyecto.

Seguridad: debido a su amplio uso y feedback continuo, se detectan errores con mayor rapidez por lo que en general es menos propenso a bugs u otro tipo de fallos.

Independencia: no nos acoplamos a un proveedor en particular ni a sus sistemas.

Muchos proyectos Open Source se encuentran en Github donde los usuarios tienen acceso directo al repositorio. Algunos muy conocidos son Linux®, Ansible, o Kubernetes.



¿Qué es y para qué sirve?

En inglés **Versioning**. Cada release generada de un artefacto software debe estar etiquetada a través de un número de versión. Dicho número de versión debe seguir un formato específico para así cada release ser identificada de manera única y aportar información de su naturaleza y objetivo (nuevas features, corrección de bugs, evolutivos...). El esquema comúnmente adoptado es el indicado en la especificación *Semantic Version (SemVer).*

RELEASE VERSION X.Y.Z

- **X** representa el número de versión **MAJOR.**
- **Y** representa el número de versión **MINOR.**
- **Z** representa el número de versión **PATCH.**

Incremento del número de versión **MAJOR, MINOR o PATCH** por cada nueva release del artefacto software en función de la naturaleza del cambio. Dada una release con número de versión X.Y.Z y una nueva versión a partir de ella:

- **Incrementar X (MAJOR version)** si la nueva versión incluye cambios de API incompatibles.
- **Incrementar Y (MINOR version)** para evolutivos o cambios retrocompatibles con la release X.Y.Z.
- **Incrementar Z (PATCH version)** para correcciones de bugs de la release X.Y.Z.

A TENER EN CUENTA

1. Si se **incrementa la versión MAJOR se resetean los valores de MINOR y PATCH** (p.e. La nueva major de la release 3.2.1 sería la 4.0.0).
2. Si se **incrementa la versión MINOR se resetea el valor de PATCH** (p.e. La nueva minor de la release 3.2.1 sería la 3.3.0).
3. Si se **incrementa la versión PATCH los valores de MAJOR y MINOR no se modifican** (p.e. Un hotfix sobre la release 3.2.1 genera una nueva release con versión 3.2.2).
4. El **incremento de MAJOR, MINOR y PATCH es secuencial**.
5. Los **cambios sobre una release generada** deben hacerse sobre **una nueva versión.**
6. Cada **release debe ser identificada de manera única.**
7. Pueden incluirse nuevos tags en las versiones de releases actualmente en desarrollo (3.0.0-alpha, 3.0.0-SNAPSHOT, 1.0.0-0.3.7).
8. La precedencia de las releases viene determinado por el valor de MAJOR, MINOR y PATCH y pre-release en este orden (0.2.0 < 0.2.1 < 0.3.1 < 1.0.0-SNAPSHOT < 1.0.0).

¿Qué es?

Una base de datos es una especie de “almacén” que nos permite guardar grandes cantidades de información para su posterior **recuperación, análisis y/o transmisión**. Podemos encontrar diversos tipos de bases de datos y se clasifican de acuerdo a las necesidades que busquen solucionar en dos grandes bloques.

TIPOS

Concepto	Definición	Ejemplos
Relacionales	Los datos se relacionan entre ellos a través de identificadores en tablas. El lenguaje predominante es SQL (Structured Query Language). Aportan a los sistemas mayor robustez y ser menos vulnerables ante fallos gracias a su **atomicidad, consistencia, aislamiento y durabilidad (ACID)**.	MySQL, PostgreSQL, Oracle, SQLite.
No relacionales	No siguen un patrón fijo como estructura de almacenamiento por lo que **su uso es muy común cuando no se tiene un esquema exacto de lo que se va a almacenar.** Se pueden encontrar bases de datos de gráficos, orientada a documentos, de columnas (Wide column store) o de claves-valor. Algunas desventajas es que no todas contemplan la atomicidad ni la integridad de los datos.	MongoDB, Redis, Elasticsearch, Cassandra, DynamoDB.

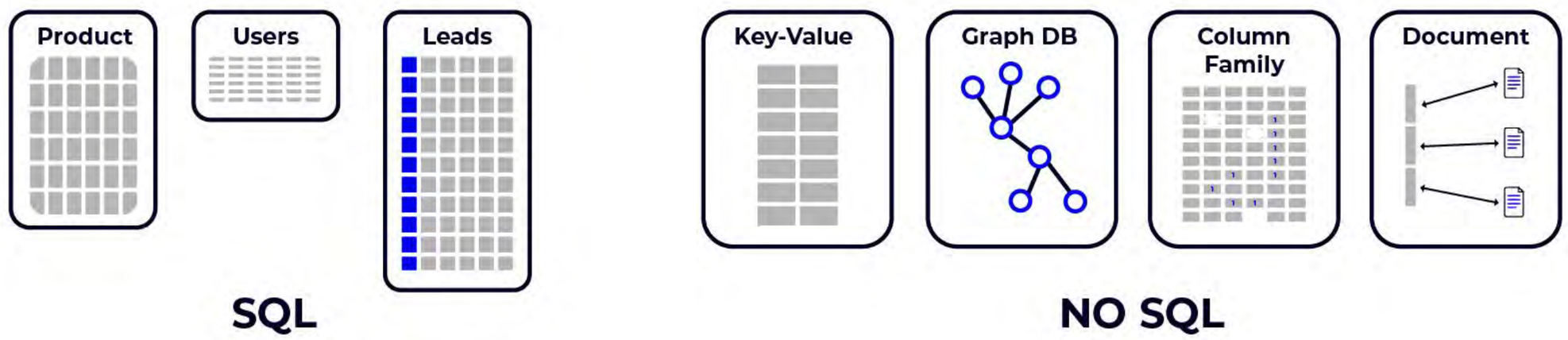
No Relacionales	Definición	Ejemplos
Clave-Valor	Cada elemento en la base de datos se almacena como un par (clave-valor) donde la clave sirve como un identificador único. Tanto la clave como valor pueden ser cualquier tipo de primitivo, como objetos compuestos.	DynamoDB, Redis, Riak, Voldemort
Columnas (Wide column stores)	También conocidas como familia de columnas o base de datos columnar. **Permiten manejar un gran volumen de datos** y mezclan conceptos de las bases de datos relaciones con una base de datos clave-valor. **Almacenan tablas de datos como secciones de columnas** en lugar de filas de datos y en cada sección se puede encontrar elementos con clave-valor	Cassandra ,HBase, Microsoft Azure Cosmos

¿Qué es?

Una base de datos es una especie de “almacén” que nos permite guardar grandes cantidades de información para su posterior **recuperación, análisis y/o transmisión**. Podemos encontrar diversos tipos de bases de datos y se clasifican de acuerdo a las necesidades que busquen solucionar en dos grandes bloques.

TIPOS

No Relacionales	Definición	Ejemplo
Documentales	**Los datos se almacenan y se consultan como documentos tipo JSON**. Los documentos pueden contener muchos pares diferentes clave-valor, o incluso documentos anidados. Son más flexibles al cambio ya que si el modelo de datos necesita cambiar, solo se deben actualizar los documentos afectados y no todo el esquema.	MongoDB
De Gráfos	**Usan nodos para almacenar entidades de datos y aristas para almacenar las relaciones entre nodos**. El valor de estas bases de datos se obtiene de las relaciones entre nodos y no hay un límite para el número de relaciones que un nodo pueda tener. Un borde siempre tiene un nodo de inicio, un nodo final, un tipo y una dirección.	Neo4J, HyperGraphDB



¿Qué es?

Un entorno en la **nube o cloud** trabaja con servidores remotos alojados en internet. Son todas aquellas instalaciones donde nuestras aplicaciones se ejecutan en ordenadores que no son de la propiedad de nuestra compañía, sino de un proveedor externo.

VENTAJAS Y DESVENTAJAS

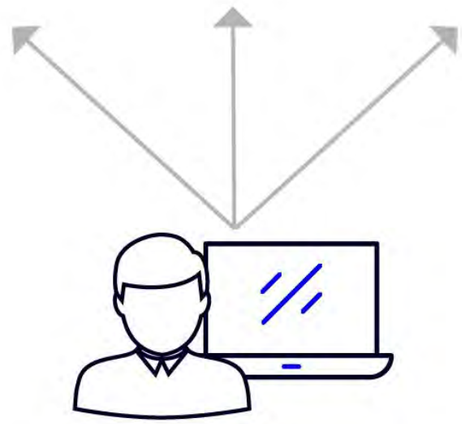
Los entornos en la nube son manejados por terceros , por lo que nosotros no tenemos acceso a las máquinas físicas. Los servicios más comunes son a la hora de montar entornos son PaaS o IaaS. Estos servicios suelen implementar sistemas basados en pagar a medida que se necesiten más recursos (servidores, memoria, almacenamiento...). Los más conocidos son Amazon Web Services, Azure y Google Cloud.

Ventajas de los entornos en la nube:

- **Se paga según las necesidades** del momento, no tenemos que ser previsores y comprar de más.
- **Facilidad para escalar**, mejorando el **time to market** y mejor respuesta ante subidas de tráfico, campañas publicitarias etc.
- **No necesitamos comprar el equipo.** El mantenimiento de equipo a la hora de instalar aplicaciones o parches corre por cuenta del proveedor que nos ofrece el servicio.

Desventajas:

- Si no se configuran bien y se ponen límites **se puede disparar el gasto de forma descontrolada**.
- **Menor control** sobre los servidores.
- **Desconocimiento** sobre cómo se tratan nuestros datos.



¿Qué es?

Se dice de aquellas instalaciones tradicionales, donde se tienen una gran cantidad de servidores en los conocidos como “data centers”. Esta instalación se lleva a cabo dentro de la infraestructura de la empresa que se encarga de comprarlos, instalarlos, y mantenerlos.

VENTAJAS Y DESVENTAJAS

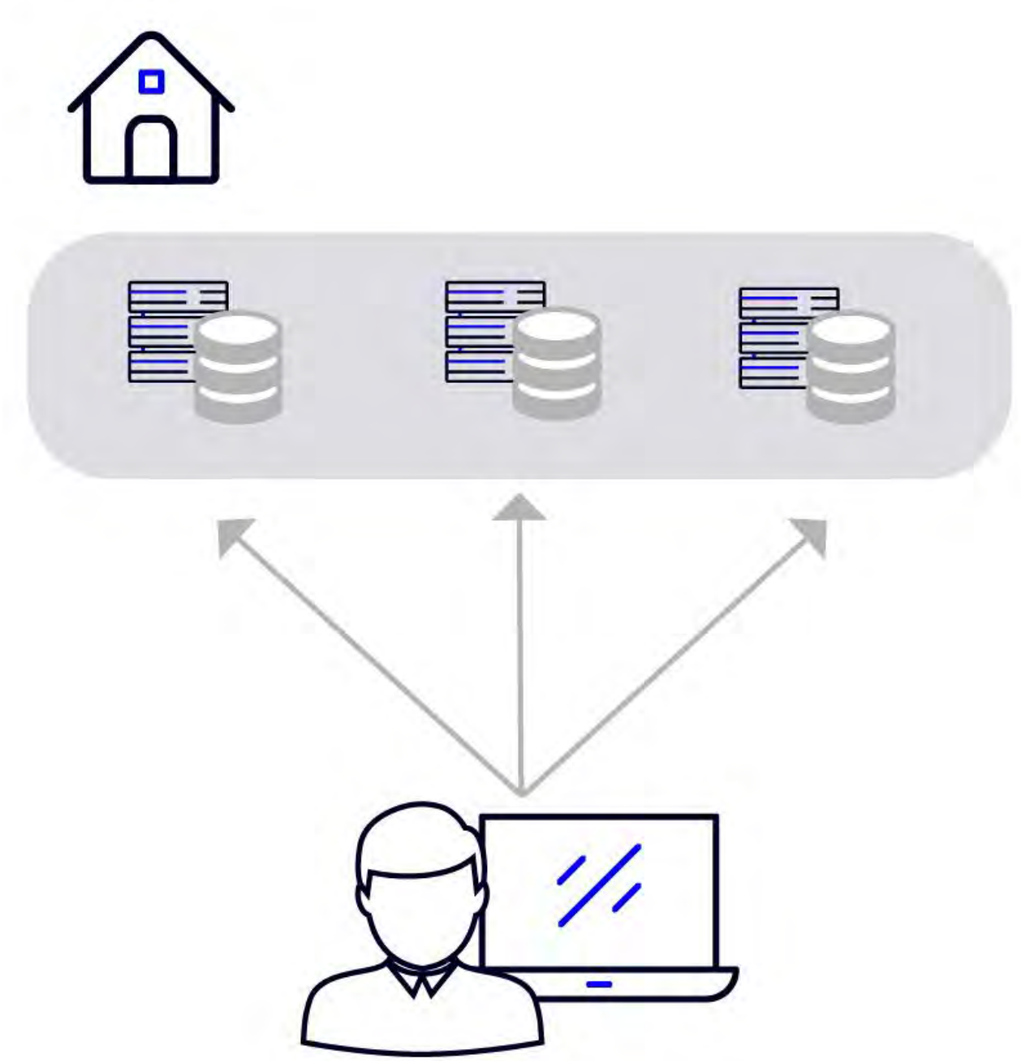
Existen distintos tipos de entornos de producción que según las necesidades de negocio, se podría optar por una opción u otra. Estos entornos pueden ser tanto **on-premise**, como en la **nube**, como **híbridos.**

Ventajas de los entornos on-premise:

- **Mayor control** sobre los servidores ya que somos nosotros los responsables de que todo funcione correctamente. Aunque esto también podría ser una desventaja, depende de cómo lo miremos.
- Implementación **personalizada.**
- **Protección de datos**, ya que la información sensible puede permanecer en nuestros equipos y no en terceros.

Desventajas:

- **Alto coste** de mantenimiento tanto hardware como software ya que se necesitará un equipo específico para estas tareas.
- Mayores adversidades para escalar tanto en tiempo como en dinero debido a que hay comprar las máquinas por adelantado y configurarlas (**peor time to market**).



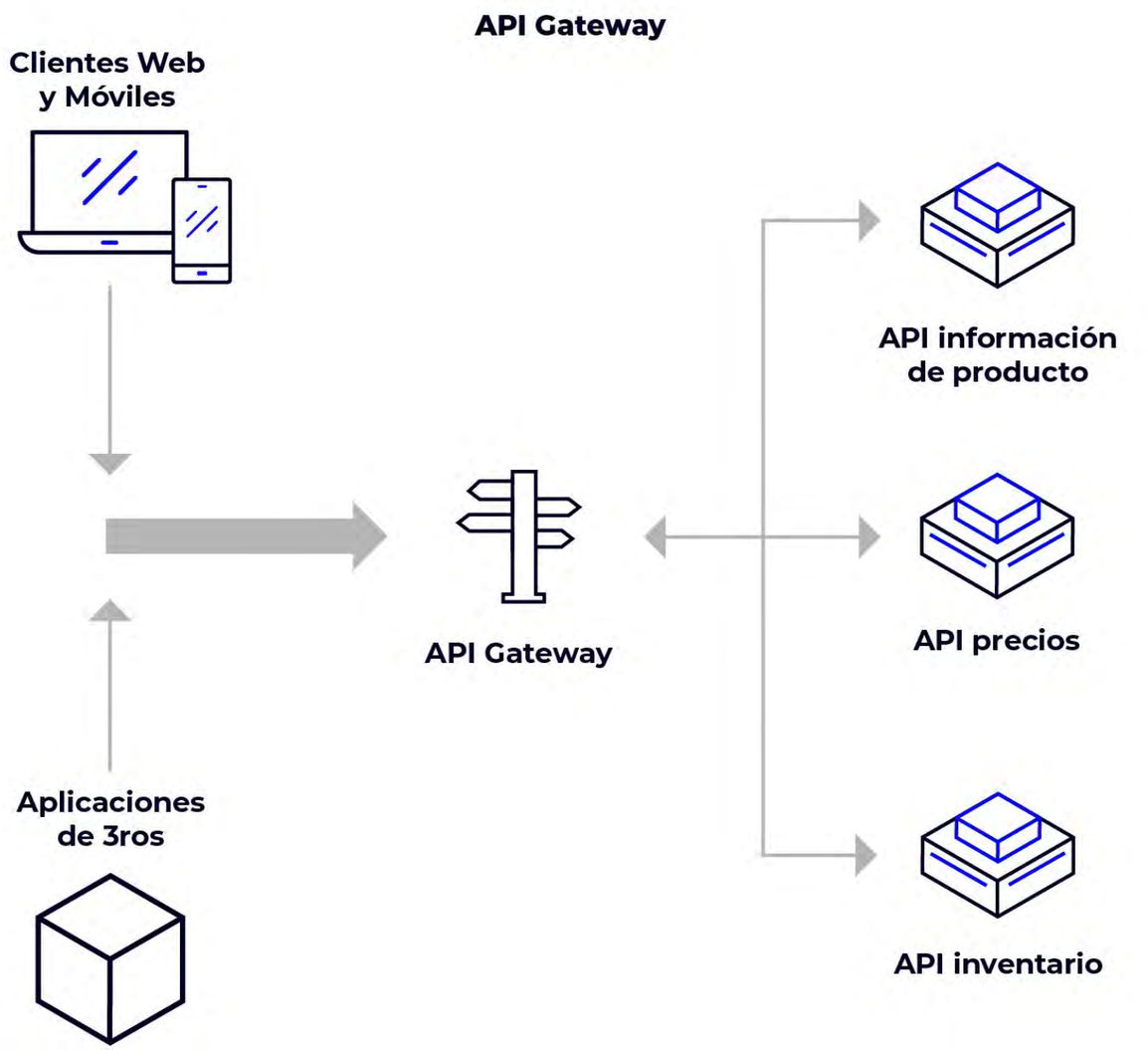
¿Qué es?

Sistema intermediario que proporciona una interfaz para hacer de enrutador entre los servicios y los consumidores desde un **único punto de entrada**. Es similar al patrón estructural Facade.

CONCEPTO

Si pensamos en una arquitectura de servicios distribuidos, habrá numerosos clientes que necesitarán intercomunicarse para completar las operaciones que se les soliciten. A medida que el número de servicios crece, es importante un intermediario que simplifique la comunicación entre los distintos clientes y servicios del sistema, en lugar de hacerlo de forma directa. Sin ningún elemento de intermediación, cada elemento debe resolver de manera individual todas las operativas relacionadas a la comunicación. **He aquí donde entra en juego el API Gateway, delegando en éste dicha complejidad, proporcionando entre otras:**.

- **Políticas de seguridad** (autenticación, autorización), protección contra amenazas (inyección de código, ataques de denegación de servicio).
- **Enrutamiento.**
- **Monitorización** del tráfico de entrada y salida.
- **Escalabilidad.**



¿Qué es?

En inglés **High Availability (HA)**, se aplica cuando queremos tener un plan de contingencia sobre cualquier componente en caso de que se presente alguna situación irregular que impida la continuidad de los servicios.

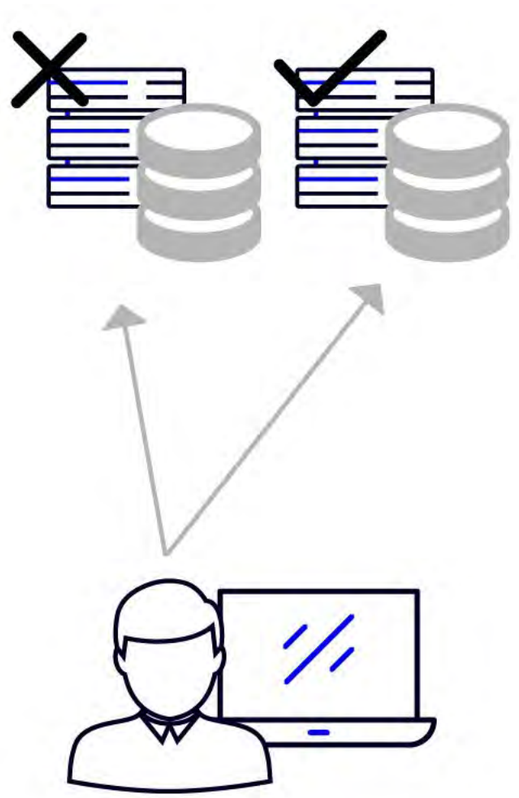
¿EN QUÉ CONSISTE?

Disponibilidad se refiere a la habilidad de los usuarios para acceder y usar el sistema. Se debe tener en cuenta que el tiempo de funcionamiento y disponibilidad no son sinónimos. Un sistema puede estar en funcionamiento y no disponible como en el caso de un fallo de red. **La criticidad del servicio es muy importante**, no podemos comparar la disponibilidad que debe tener un cajero automático (posiblemente 365 días 24x7) que otro servicio que solo esté disponible de Lunes a Viernes. Algunas **causas en la que podemos dejar de prestar servicio** ajenas a nuestra voluntad pueden ser:

- **Desastres naturales** (terremotos, inundaciones, incendios).
- **Cortes de suministro eléctrico.**
- **Errores operativos.**

Tener un sistema altamente disponible puede lograrse utilizando servicios cloud, los más conocidos pueden ser Amazon Web Services, Google Cloud o Azure. Necesitamos tener la capacidad de detectar un fallo en el servicio principal de una forma rápida y que a la vez, sea capaz de recuperarse del problema. La **monitorización** y **auditoría** de los componentes es fundamental para saber en tiempo real que está ocurriendo o como se comporta el sistema.

Si hablamos de un e-commerce se podría ofrecer el uso de **balanceadores de carga**, donde en vez de tener un único servidor, balanceamos la carga en varios servidores según la demanda que tengamos, de este modo, reduciremos las posibilidades de que el servidor principal colapse. Otras soluciones a tener en cuenta podría ser **dividir geográficamente la infraestructura en varias localizaciones físicas**, separadas entre ellas varios kilómetros, que ante cualquier desastre natural, pueda seguir prestando servicio.



DevOps:
Piezas básicas de la infraestructura

SaaS vs PaaS vs IaaS

autentia

Definición

SaaS, PaaS e IaaS son tres tipos de servicio comúnmente asociados a cloud o la nube que significan Software como Servicio, Plataforma como Servicio e Infraestructura como Servicio, respectivamente.

¿EN QUÉ CONSISTEN?

SaaS

Software como servicio, también conocido como servicios de aplicaciones en la nube, representa la opción más utilizada por las empresas en el mercado de la nube. Consiste en proveer a los usuarios aplicaciones a través de internet, administradas por un proveedor externo. La mayoría de estas aplicaciones **se ejecutan directamente a través de su navegador web**, lo que significa que no requieren descargas ni instalaciones en el lado del cliente.

Ejemplos:

- Google Apps.
- Dropbox.
- Salesforce.

PaaS

También conocido como servicios de plataforma en la nube, proporcionan componentes en la nube a cierto software mientras se usan principalmente para aplicaciones. PaaS **ofrece un marco a los desarrolladores sobre el que pueden crear aplicaciones personalizadas.** Todos los servidores, el almacenamiento y las redes pueden ser administrados por la empresa o por un proveedor externo, mientras que los desarrolladores pueden mantener la administración de las aplicaciones.

Ejemplos:

- Windows Azure.
- AWS Elastic Beanstalk.
- Google App Engine.

IaaS

Los servicios de infraestructura en la nube, están hechos de recursos informáticos altamente escalables y automatizados.

Permite a las empresas comprar recursos como almacenamiento, redes o virtualización, a pedido y según sea necesario en lugar de tener que comprar hardware directamente.

IaaS ofrece a los usuarios alternativas basadas en la nube a la infraestructura local, para que las empresas puedan evitar invertir en costosos recursos on premise.

Ejemplos:

- Amazon Web Services (AWS).
- Microsoft Azure.
- Google Compute Engine (GCE).

¿Qué es?

Un entorno híbrido es aquel que combina los entornos en la **nube (cloud)** con los entornos **on-premise**. Se basa en tener parte de los servidores en la propia infraestructura de la empresa y otra parte en servidores remotos gestionados por un proveedor externo.

VENTAJAS Y DESVENTAJAS

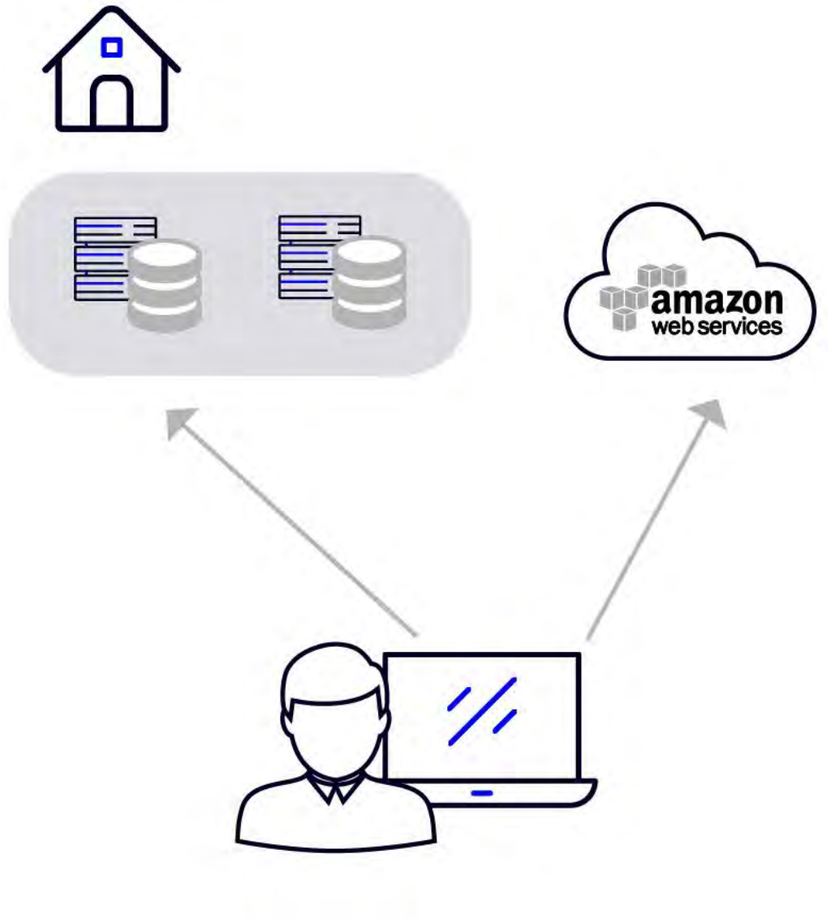
Los sistemas híbridos pueden tener todas las ventajas e inconvenientes de los otros dos entornos. Para gestionar esas desventajas, tendremos que **optimizar los servicios** que irán en la nube para maximizar las ventajas, y minimizar los costes. Lo mismo debemos hacer para las instalaciones on premise, tendremos que optimizar su uso para no incurrir en altos costes, o no poder escalar de forma rápida.

Ventajas de los entornos en la nube:

- **Se paga según las necesidades** del momento..
- **Facilidad para escalar**, mejorando el **time to market** y mejor respuesta ante subidas de tráfico, campañas publicitarias, etc.
- **No necesitamos tener a un equipo para su mantenimiento** ya que el proveedor nos ofrece ese servicio, aplicando parches de seguridad y otras actualizaciones.

Ventajas de los entornos on-premise:

- **Mayor control** sobre los servidores ya que somos nosotros los responsables de que todo funcione correctamente.
- Implementación **personalizada**.
- **Protección de datos** ya que la información sensible puede permanecer en el sistema y no en terceros.



¿Qué es?

HTTPS (Hypertext Transfer Protocol **Secure**) es una extensión de HTTP utilizada para una comunicación segura, identificando a los interlocutores (servidor y opcionalmente el cliente) y estableciendo una comunicación privada a través de internet.

¿EN QUÉ CONSISTE?

Para establecer una comunicación privada, **el protocolo de comunicación es encriptado utilizando TLS** (Transport Layer Security). Gracias a esta encriptación bidireccional, se puede determinar con seguridad que nos estamos comunicando con la página web que deseamos sin la interferencia de posibles atacantes. Esta encriptación consiste en una **encriptación simétrica**, con unas claves que son generadas de forma única para cada nueva conexión, y están basadas en un secreto compartido que se negocia con el servidor al principio de la sesión, en lo que se denomina **TLS handshake**.

El hecho de que sea una encriptación simétrica significa que **ambas claves** (la que utiliza el servidor y la que utiliza el cliente) **son privadas e idénticas**.

A TENER EN CUENTA

Aparte de la encriptación, para confirmar que el servidor es de confianza, el protocolo HTTPS requiere de una autenticación por parte de un tercero de confianza (una **autoridad certificadora**) que firme **certificados digitales.**.

Los servidores web típicamente abren 2 puertos. El **puerto 80** de un servidor se utiliza para las comunicaciones HTTP, y el **puerto 443** para las comunicaciones HTTPS. Estos puertos no se ponen de forma aleatoria, sino que **están estandarizados** para esta función.

En muchas ocasiones y por seguridad, las comunicaciones HTTP están deshabilitadas, y lo que se hace es que se redirige el puerto 80 al 443, para permitir únicamente establecer comunicación HTTPS.

¿En qué consiste?

Los verbos HTTP nos permiten realizar distintas operaciones (p. ej. alta, baja, lectura, modificación...) sobre los recursos de nuestras APIs REST. Cada uno de ellos se utiliza para una operación y finalidad concreta.

TIPOS DE VERBOS HTTP

- **GET**: **recuperar** la información de un único recurso o un listado (p.e. Un listado de cursos, un curso a partir de su id...).
- **POST: dar de alta un recurso**. En el cuerpo de la petición se le proporciona la información a dar de alta.
- **PUT: modificar un recurso existente.** En el cuerpo de la petición se le proporciona la información del recurso actualizada.
- **PATCH:** una **modificación parcial de un recurso.** En el cuerpo de la petición se le proporciona la información a actualizar.
- **DELETE:** dar de **baja un recurso**. En la URL de la petición se le especifica el identificador del recurso a dar de baja.
- **OPTIONS:** se utiliza para describir las opciones de comunicación con el recurso.
- Otros: **HEAD, CONNECT y TRACE.**

CARACTERÍSTICAS

Un verbo HTTP puede ser:

- **Idempotente:** el resultado de la operación deja al servidor en el mismo estado tantas veces se ejecute. GET, PUT, DELETE y HEAD son idempotentes. POST *no* es idempotente.
- **Seguro:** un verbo HTTP es seguro cuando no altera el estado del servidor. GET y OPTIONS son seguros. POST, PUT y DELETE no son seguros. *Todo verbo seguro es idempotente.*
- **Cacheable:** la respuesta a la petición HTTP se guarda en caché de modo que pueda utilizarse en próximas peticiones. GET y HEAD son cacheables mientras que PUT y DELETE *no*, llegando a invalidar resultados cacheados de GET o HEAD.

HTTP 1.0

autentia

¿Qué es?

HTTP es el **protocolo que utilizan los navegadores entre el cliente y el servidor.** Específicamente HTTP 1.0 lanza nuevas funcionalidades en 1996 después de algunas carencias en la versión anterior.

HISTORIA

HTTP 0.9 no usaba cabeceras, transmitía solo ficheros HTML planos, únicamente soportaba el método GET y la conexión se cerraba inmediatamente después de obtener la respuesta. Por estas limitaciones se introdujeron nuevas funcionalidades en la versión 1.0:

- **Cabeceras** como *Content-type* que permite transmitir otro tipo de ficheros como scripts, imágenes etc.
- **Códigos de estado** que indican si la petición se resolvió con éxito o no.
- Soporte para **POST, HEAD y GET.**

A pesar de estas mejoras, la primera versión estándar no llegó hasta el año 1997 cuando se lanza HTTP 1.1.

HTTP 1.1

HTTP 1.1 presentó mejoras de rendimiento sobre las dos versiones anteriores:

- Soporte para **GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS.**
- **Conexiones persistentes con la cabecera *Keep-alive***, permitiendo múltiples peticiones/respuestas por conexión.
- **Cabecera *Upgrade*** que permite cambiar el protocolo de conexión.
- **Cabecera *Cache-control*** que permitía especificar políticas de cacheo en las peticiones.
- Soporte para **transmisión por partes** (chunk transfers) que permitía el envío de contenido de forma dinámica.

¿Qué es?

HTTP 2.0 es un protocolo binario que **busca aumentar la velocidad y reducir la complejidad de las aplicaciones**. Esto lo consigue cambiando el modo en el que se formatean los datos y se transportan.

CARACTERÍSTICAS

HTTP 2.0 conserva la semántica de HTTP 1.0, se mantienen sus conceptos básicos (verbos, códigos de estado, campos de cabecera, etc.) y se introducen mejoras con el objetivo de mejorar el rendimiento y optimizar nuestras aplicaciones.

De este modo **no es necesario cambiar cómo las aplicaciones existentes funcionan**, pero los nuevos desarrollos pueden aprovechar las funcionalidades que este protocolo ofrece.

HISTORIA

Se considera una evolución del **protocolo SPDY**, desarrollado por Google en 2009 y que también buscaba reducir los tiempos de carga de la web. Muchas de las mejoras que introducía SPDY fueron llevadas a HTTP 2.0.

En 2012 aparecen los primeros borradores de HTTP 2.0 y en los años siguientes ambos protocolos continuaron evolucionando en paralelo, hasta que en 2015 SPDY fué deprecado en favor de HTTP 2.0.

MEJORAS

- **Multiplexación**: permite enviar y recibir varias peticiones al mismo tiempo usando una misma conexión. Así se consigue mejorar la velocidad de carga de la página y se disminuye la carga en los servidores web.
- **Protocolo binario en lugar de texto**: ofrece mayor eficiencia en el transporte del mensaje y en su interpretación. Además son menos propensos a errores ya que con el texto hay problemas de espacios en blanco, capitalización, finales de línea, etc.
- **Servicio 'server push'**: el servidor envía información al cliente antes de que el cliente la pida. Así cuando el cliente necesite esos recursos ya los tiene, ahorrando tiempo.
- **Compresión de cabeceras**: en general las cabeceras cambian poco entre peticiones y además con el uso de cookies su tamaño puede incrementar mucho. Por eso con HTTP 2.0 se van a enviar comprimidas.
- **Priorización de transmisiones**: dentro de una misma conexión se da más peso a aquellas transmisiones que tengan más importancia para gestionar mejor los recursos.

Http/2

Definición

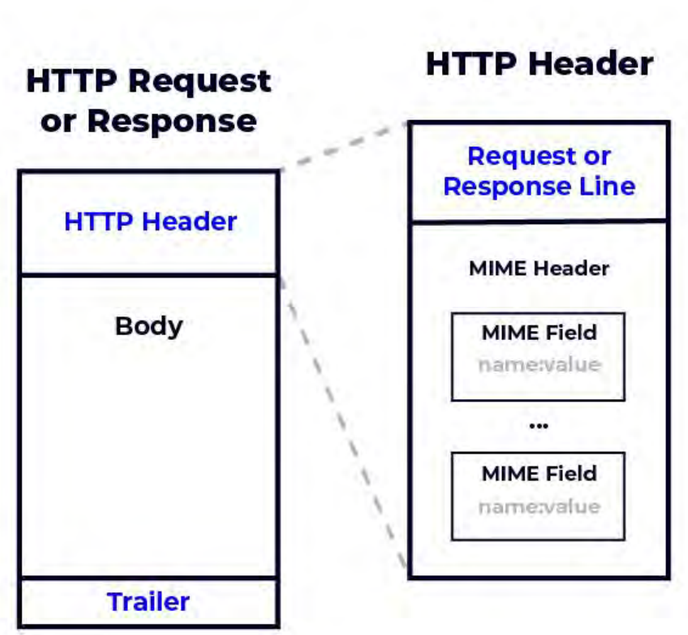
Los cabeceras HTTP **son parámetros opcionales**. Permiten tanto al cliente como al servidor **enviar información adicional**. Una cabecera consta de un nombre (sensible a mayúsculas y minúsculas), separado por ":", seguidos del valor a asignar. Por ejemplo "Host: www.example.org". Las cabeceras **varían dependiendo de si se trata de una request o una response**.

CABECERAS PRINCIPALES EN LA REQUEST

- **Cookie**: la cookie HTTP es un dato que permite al servidor **identificar las peticiones que viene de un mismo cliente**. HTTP es un protocolo sin estado. El servidor asigna una cookie a cada cliente y este las referencia usando la cabecera Cookie.
- **User-Agent**: identifica el tipo de aplicación, el sistema operativo, **la versión del navegador desde donde se hace la petición**, permitiendo al servidor bloquearla si la desconoce.
- **Host**: especifica el dominio del servidor, la versión HTTP de la solicitud y el número de puerto TCP en el que escucha (opcional).
- **X-Requested-With**: identifica solicitudes AJAX hechas desde JavaScript con el valor del campo XMLHttpRequest.
- **Accept-Language**: anuncia al servidor **qué idiomas soporta el cliente**.

CABECERAS PRINCIPALES EN LA RESPONSE

- **Content-Type**: determina el tipo de cuerpo (tipo MIME) y la codificación de la respuesta.
- **Content-Length**: indica el tamaño del cuerpo de la respuesta en bytes.
- **Set-Cookie**: es la cabecera que utiliza el servidor para enviar la cookie asignada al cliente. Con esta cookie, el servidor puede identificar y restringir el acceso a ciertas rutas al cliente. También se puede indicar la duración o fecha de vencimiento de la cookie.



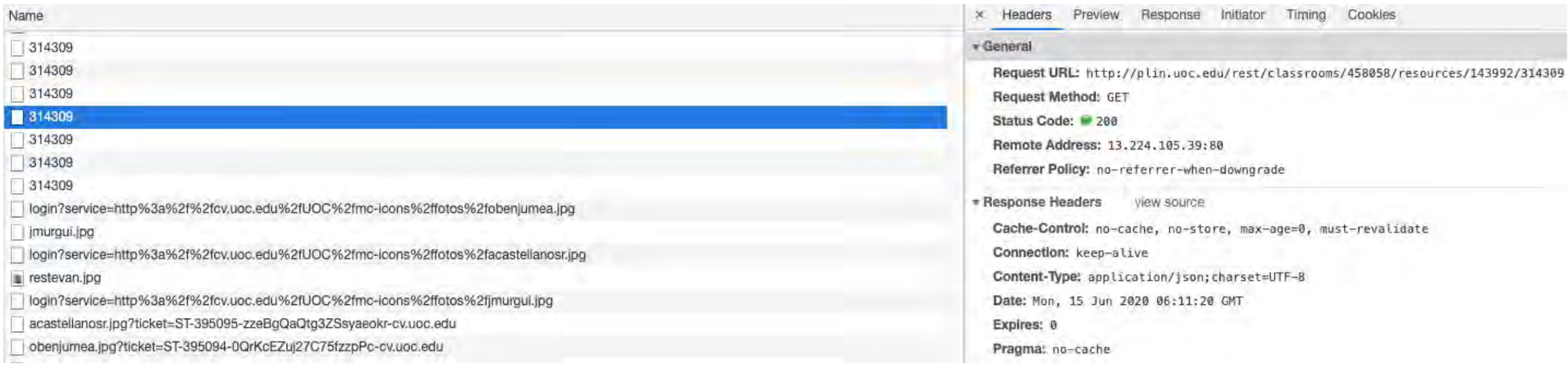
Http/2

¿Para qué se utilizan?

Las cabeceras de caché se utilizan para especificar *las políticas de almacenamiento de caché* que debe **usar el navegador tanto para las request como las response**. Con estas políticas podemos especificar tanto el cómo almacenar un recurso así como donde se almacena y su vigencia.

CABECERAS

- **Cache-Control**: esta cabecera se utiliza para especificar parámetros de cacheado en el navegador.
  - ***no-cache***: obliga a validar un recurso caducado contra el servidor.
  - ***no-store***: no permite cachear la respuesta. Muy usada en páginas con datos confidenciales.
  - ***private*** vs. ***public***: si solo se permite la caché del navegador o también a terceros (CDN, Proxies, etc).
  - ***max-age***: vida útil del recurso cacheado (en segundos).
- **Validators:** utiliza la cabecera ETag como hash del recurso que tiene el servidor y verificar si el recurso cacheado caducado ha cambiado. Si no ha cambiado, no lo vuelve a solicitar, actualizando el recurso.
- **Extension Cache-Control:** Estas son algunas cabeceras que extienden la directiva Cache-Control.
  - ***immutable***: no valida con el origen, solo se refresca si caduca.
  - ***stale-while-revalidate***: especifica el tiempo que se mostrará el archivo obsoleto mientra se valida en origen.
  - ***stale-if-error***: especifica tiempo extra de un recurso caducado sin validar.



Definición

Una base de datos distribuida no está limitada a un sistema, **se extiende por diferentes sistemas en distintas ubicaciones**, permitiendo un **escalado horizontal** pero dando la sensación a los usuarios finales de trabajar con un único sistema.

CONCEPTO

Las bases de datos distribuidas surgen con el objetivo de almacenar la mayor cantidad de datos sin sacrificar el rendimiento del sistema. Las distintas máquinas del sistema se llaman nodos y se comunican entre ellos formando una red.

Hay distintas formas de distribuir los datos:

- **Mediante partición**: consiste en almacenar un dato una vez, pero la información se reparte entre los nodos.
- **Mediante réplica**: consiste en que cada nodo tiene su propia copia completa de la base de datos.

En general se toma un enfoque **híbrido** en el que la información se almacena de forma particionada entre los nodos y además se cuenta con réplicas de los datos. Esto permite que **el sistema sea tolerante a fallos**, al tener replicada la información entre los nodos.

VENTAJAS

- **Mayor disponibilidad y fiabilidad** gracias a la distribución de los datos y redundancias en el sistema.
- **Mejor rendimiento.** Las transacciones que afectan a varios sitios se realizan en paralelo y las bases de datos son más pequeñas.
- **Proporciona autonomía local**. Cada parte de una organización puede controlar los datos que le pertenecen.

DESVENTAJAS

- **Aumento de la complejidad del diseño**. Son necesarias una serie de tareas para mantener la transparencia y coordinar los distintos nodos.
- Esta complejidad puede afectar a los usuarios si no se toman las medidas oportunas. **Si los datos se distribuyen de manera incorrecta puede afectar al rendimiento** del sistema, generando sobrecargas o cuellos de botella.

¿Qué es?

La **caché** es una **capa de almacenamiento de datos de alta velocidad** que almacena datos normalmente **temporales** o de naturaleza transitoria, de modo que las solicitudes de acceso a esos datos **se atiendan más rápido que si se recogieran de la ubicación principal** del almacenamiento de esos datos.

¿EN QUÉ CONSISTE?

Cuando tenemos un servidor expuesto a un gran número de peticiones, una técnica muy común y eficaz es utilizar un **sistema de caché**. De esta forma, si hay ciertas peticiones que se repiten mucho y la **respuesta es siempre la misma**, esa respuesta se puede cachear. Las cachés se pueden dividir en dos tipos: distribuida y no distribuida.

La primera se suele utilizar para guardar resultados de operaciones que son **poco pesados**. La segunda, en vez de utilizar la memoria de la misma máquina, se dedica un **conjunto** de máquinas explícitamente para este fin. Una de las **desventajas** de una caché distribuida es que su acceso es **más lento** que una en memoria, ya que normalmente tendremos que acceder a una **máquina externa**. Por otro lado, este tipo de caché es capaz de guardar **más información**, y por tanto es más útil para almacenar **resultados de operaciones más pesadas o costosas**.

VENTAJAS

- La **respuesta a una petición será más rápida** que si se tuviera que acceder a base de datos.
- Se **libera de carga al servidor**, pudiendo dedicarse a otras peticiones, y por tanto soportando una mayor carga de peticiones en general.

¿Qué es?

Transport Layer Security (TLS) es un protocolo criptográfico que proporciona autenticación y cifrado de la información intercambiada entre las distintas partes que operan sobre una red.

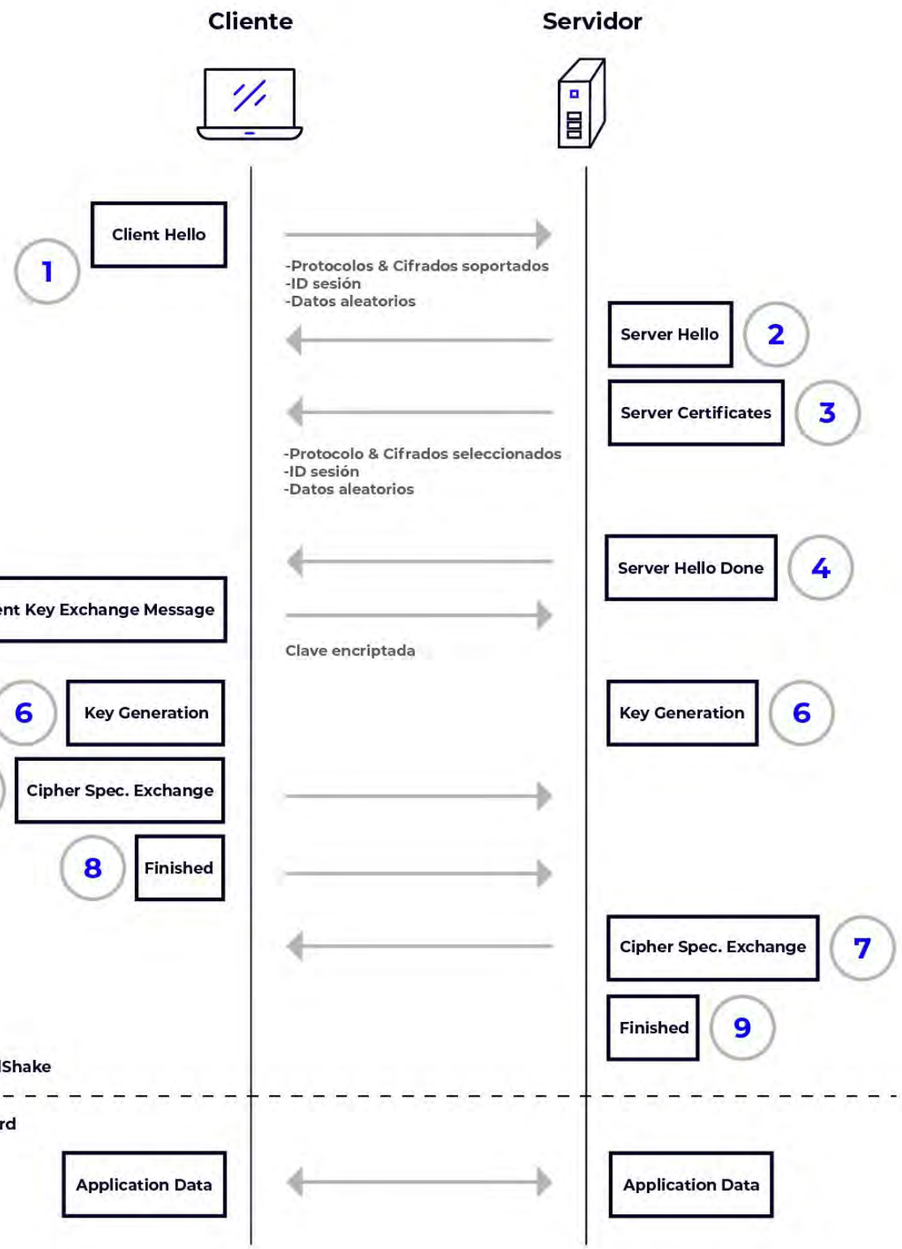
¿EN QUÉ CONSISTE?

TLS es el **sucesor de SSL (Secure Socket Layer)**. Se caracteriza por:

- **Comunicación privada y encriptada** tanto en el lado cliente como en el lado servidor a través de **un cifrado simétrico.**
- **Intercambio de claves** públicas y autenticación a través de **certificados digitales.**
- **Negociación del algoritmo** de intercambio de mensajes entre las partes.

TLS **intercambia registros** entre las partes en un **formato específico**. Cada registro es comprimido, cifrado y empaquetado con un código de MAC. Hay dos protocolos para ello:

- **TLS Handshake:** es un protocolo que sirve para que dos partes se verifiquen entre sí y puedan establecer un tráfico cifrado e intercambiar claves. Las claves generadas para dicho cifrado se consiguen a través de una negociación entre las partes.
- **TLS Record:** se lleva a cabo la autenticación de los datos para que su transmisión sea mediante una conexión fiable y segura (p.e. TCP). Los mensajes que se intercambien estarán cifrados simétricamente mediante las claves negociadas en el Handshake.



¿Qué es?

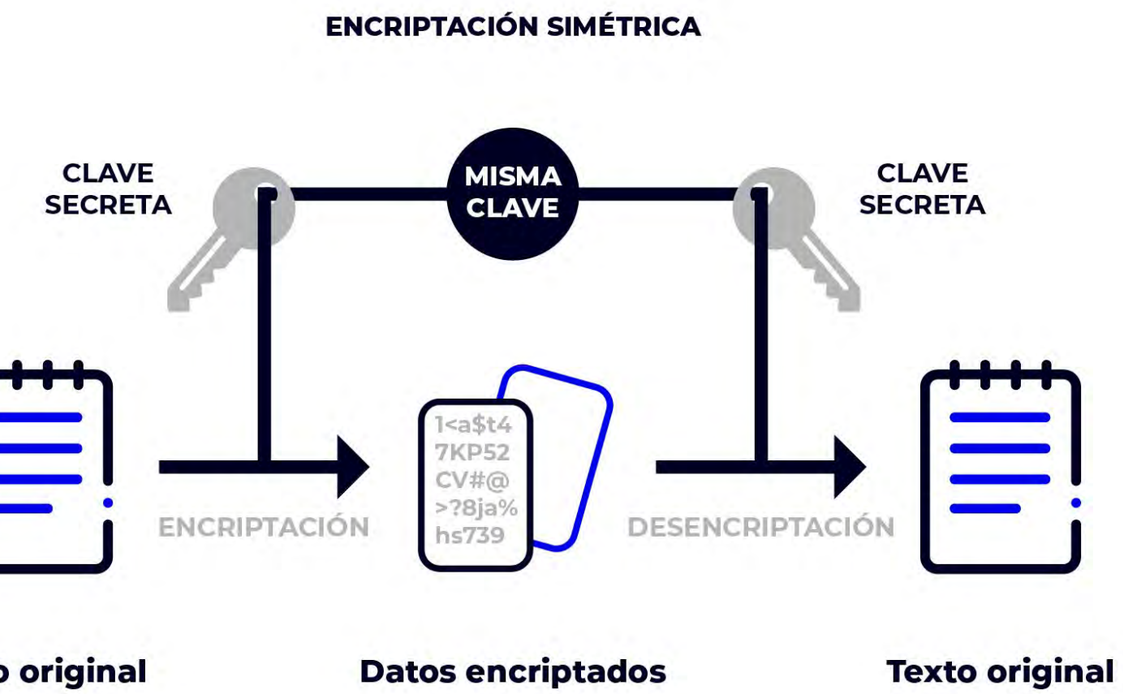
El **cifrado simétrico** es una forma de encriptación en la que sólo se utiliza una clave, tanto para encriptar como para desencriptar.

¿EN QUÉ CONSISTE?

Al haber sólo una clave, esta es **privada**, y las distintas entidades comunicándose **deben compartirla entre ellas** para poder utilizarla en el proceso de desencriptación.

El hecho de que todas las partes tengan acceso a la clave privada es la **principal desventaja** de este tipo de encriptación, ya que hay más probabilidad de que esa clave pueda ser vulnerada. Además, si un atacante consigue esta clave podría desencriptar y leer todos los mensajes y datos de esa conversación. La **ventaja** que tiene sobre la alternativa de usar un cifrado asimétrico es que tiene mejor rendimiento, porque los algoritmos usados son más sencillos.

La clave puede ser una contraseña/código o puede ser una cadena de texto o números aleatoria generada por un software especializado en generar este tipo de claves.



Certificado

autentia

¿Qué es?

Documento virtual que contiene los datos identificativos de una persona física o de un sitio web y que está autenticado por un organismo oficial encargado de emitir y validar dicha información.

TIPOS

Cuando queremos identificar a una persona física, una entidad o un dominio de manera digital, hablamos de **certificados digitales**. Una **Autoridad Certificadora (AC)** será la encargada de firmar y emitir los certificados, verificando que quien lo solicite es quien dice ser.

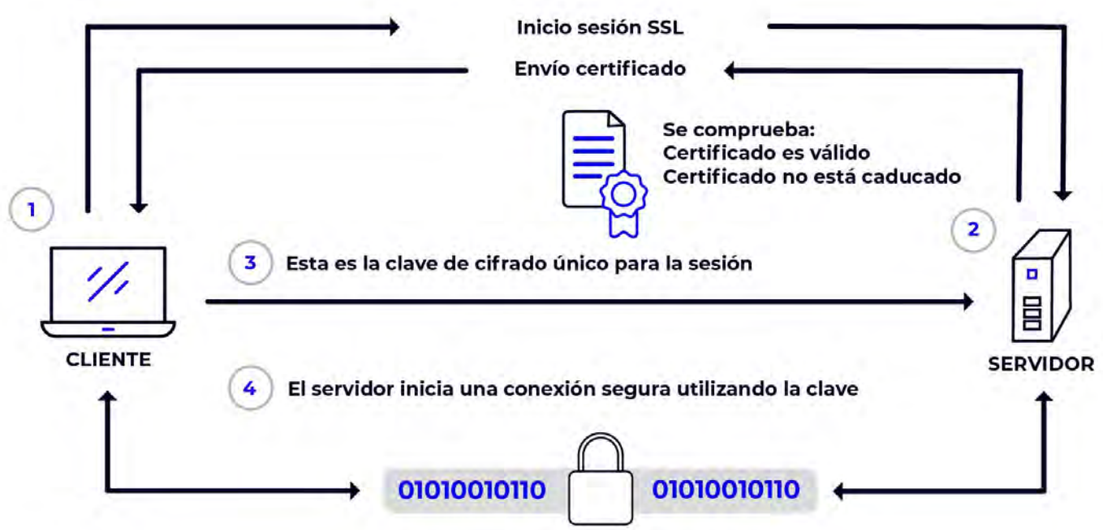
También podemos validar un sitio web a través de certificados. Se configura el servidor para que use dicho certificado y usando el protocolo SSL se cifra la información enviada al servidor. Esto es lo que comúnmente se conoce como **certificado SSL** debido a que este protocolo es el más extendido hoy en día. Los navegadores suelen incluir por defecto un repositorio de AC de confianza, pero si hay alguna AC en la que en principio no se confía (porque lleva poco tiempo en funcionamiento, por ejemplo), habrá que instalar manualmente el certificado en el repositorio del navegador. **Los certificados caducan** y esto se hace para asegurar que toda la información es precisa y demuestra una validez como propietario de confianza del dominio

¿Qué ventajas nos ofrecen los certificados?

- Ahorro de espacio, tiempo y dinero.
- Confidencialidad y seguridad en las comunicaciones.

¿CÓMO FUNCIONA?

Se hace uso de claves públicas y claves privadas. En el caso de un sitio web, la clave privada permanece en el servidor donde se ha configurado el certificado. Cuando un navegador web desea establecer una nueva conexión, el servidor comparte la clave pública con el cliente para establecer un método de cifrado y el cliente confirma que reconoce y confía en la entidad emisora del certificado. Este proceso inicia una sesión segura que protege la privacidad y la integridad del mensaje.



Cifrado asimétrico

autentia

¿Qué es?

El **cifrado asimétrico** es una forma de encriptación en la que las claves vienen en parejas. Lo que una de las llaves encripta, sólo puede ser desencriptado por la otra.

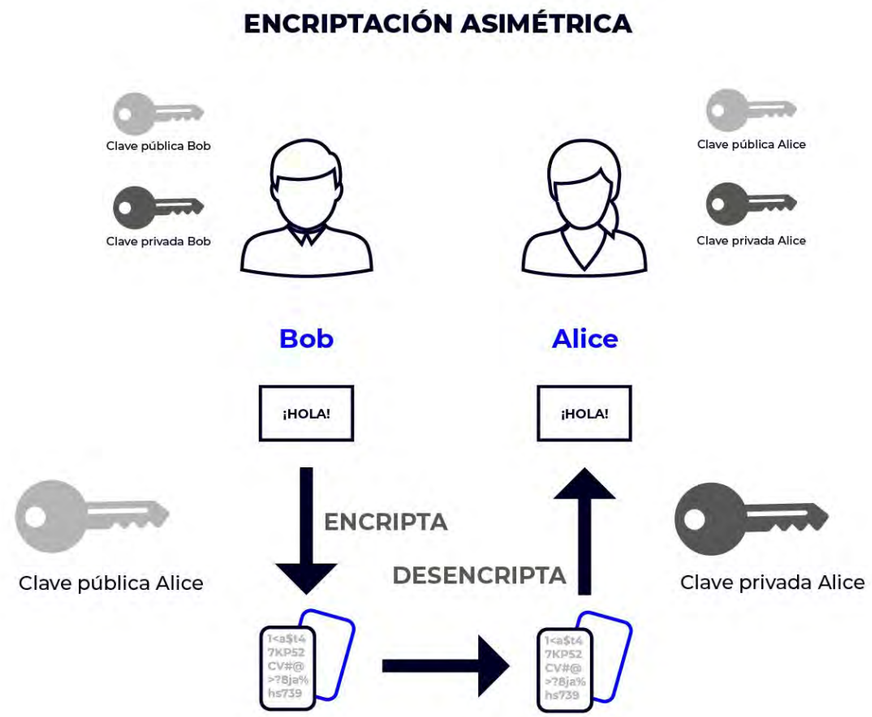
¿EN QUÉ CONSISTE?

El cifrado asimétrico tiene **dos claves** por persona, a diferencia del cifrado simétrico que sólo tiene una. Pongamos que tenemos dos personas: Bob y Alice. Los pasos para encriptar y desencriptar un mensaje que Bob le envía a Alice serían los siguientes:

- Bob dispone de la clave pública de Alice, que sirve para encriptar (las claves públicas las puede ver cualquiera). **Encripta un mensaje con esa clave pública**.
- **Manda el mensaje encriptado a Alice**.
- Los mensajes que se hayan encriptado usando la clave pública de Alice sólo se pueden desencriptar usando la clave privada de Alice, clave que solo ella conoce. **Alice usa su clave privada para desencriptar el mensaje**.

A TENER EN CUENTA

Si un atacante consiguiera la clave privada de Bob podría leer los mensajes que le llegan a Bob, ya que han sido encriptados usando su clave pública, **pero no podrían descifrar mensajes que Bob mande a otros**, ya que estos mensajes se encriptan usando claves públicas de otros.



Definición

El DNS (Domain Name System) es un sistema de nombramiento jerárquico y descentralizado para ordenadores, servicios, etc. conectados a internet o a una red privada. Permite **traducir un nombre fácilmente memorizable para una persona a una IP**.

CONCEPTO

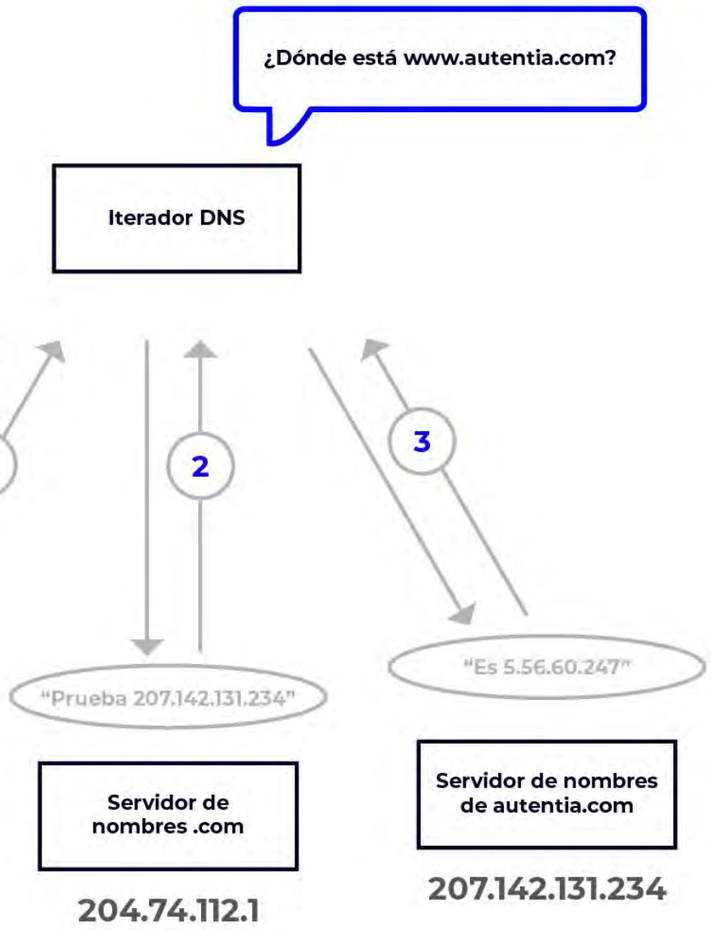
El DNS nació de la necesidad de recordar fácilmente los nombres de los servidores de Internet. **Cada dominio tiene al menos un servidor de nombres** que publica información sobre ese mismo dominio y sobre los servidores de nombres de cualquier dominio subordinado.

Cuando intentas acceder a www.autentia.com comienza un proceso de consultas iterativo para resolver la IP del sitio.

1. Primero se pregunta al servidor de nombres raíz, que responde con la IP de un servidor de nombres subordinado.
2. El servidor de nombres subordinado responde con la dirección del servidor de nombres de autentia.com.
3. Finalmente este servidor conoce la IP del dominio y lo devuelve.

En general, la resolución de los nombres **se hace de forma transparente al usuario a través del cliente** (navegador, aplicación de correo, etc.).

Además, para reducir el número de peticiones, ya que las direcciones se cachean a distintos niveles. Por ejemplo los navegadores cachean los resultados de estas consultas y también los servidores de nombres disponen de cachés para evitar hacer tantas peticiones.



Balanceadores de carga

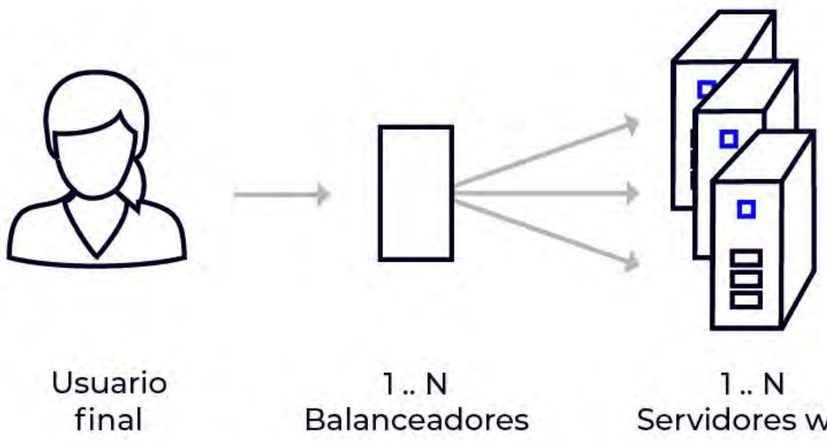
autentia

¿Qué son?

Esta parte de la infraestructura **optimiza** el **rendimiento** y la **disponibilidad** de un servicio al distribuir las solicitudes a lo largo de múltiples recursos en una red.

¿EN DÓNDE ENCAJAN?

Los balanceadores se integran **entre el usuario y el servicio solicitado**. Un **algoritmo de balanceo** determinará el servidor que va a procesar la solicitud del usuario.

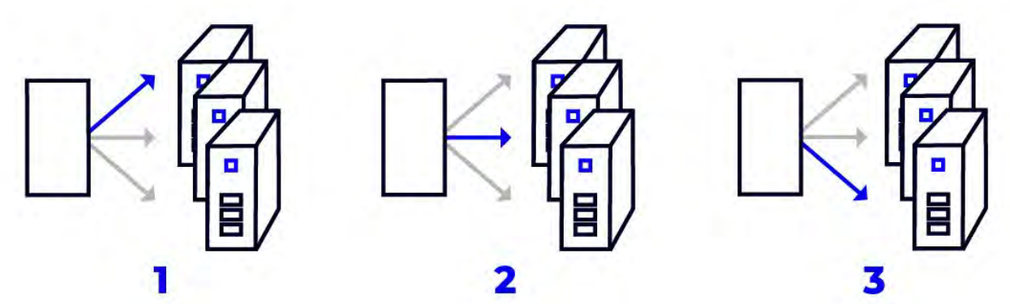


OTROS USOS PRÁCTICOS

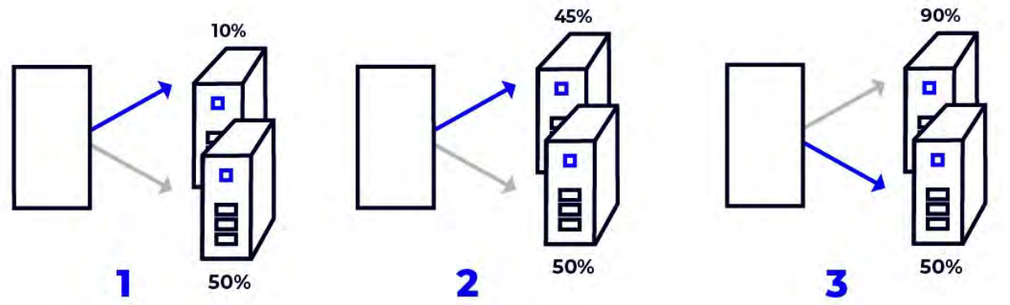
- Redireccionar el tráfico de una aplicación antigua a una con una versión más actualizada, sin afectar la disponibilidad del servicio.
- Enviar la solicitud a un microservicio u otro a partir de la URL.

ALGORITMOS DE BALANCEO

- **Estáticos**: determinan el servidor a partir de criterios fijos. Por ejemplo, el algoritmo **round-robin** reparte equitativamente las solicitudes.



- **Dinámicos**: utilizan información expuesta por los servidores para determinar cual la recibirá. Esta información se refiere generalmente a la **carga actual del servidor**.



API

autentia

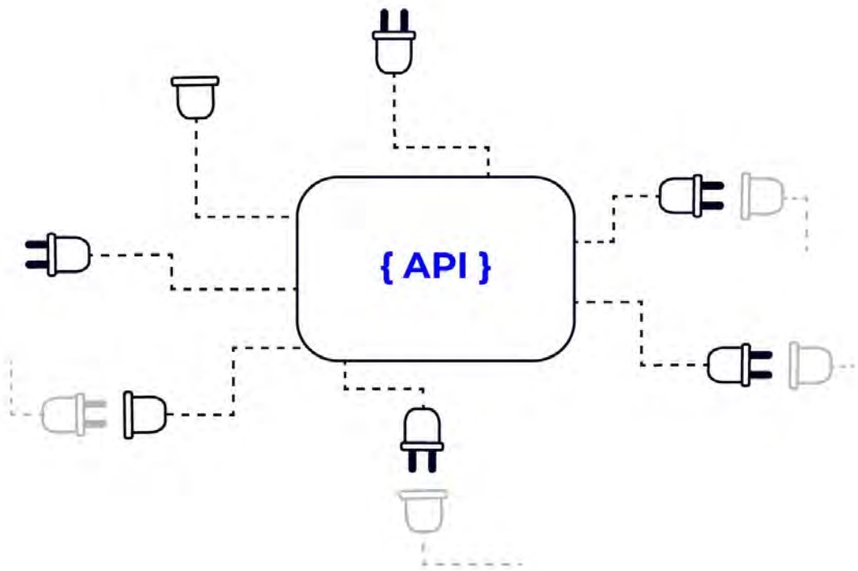
¿Qué es?

API es el acrónimo de Application Programming Interface, que **es un intermediario de software que permite que dos aplicaciones se comuniquen entre sí**. Es un conjunto de definiciones y protocolos para construir e integrar software. Cada vez que utilizas una aplicación como Facebook, envías un mensaje instantáneo o compruebas el clima en su teléfono, está utilizando una API.

CONCEPTO

Una API simplifica la programación al abstraer la implementación subyacente y sólo exponer los objetos o acciones que el desarrollador necesita.

Una API web consta de uno o más endpoints expuestos públicamente de un sistema definido de mensajes de petición-respuesta, generalmente en formato JSON o XML que se expone a través de la web, más comúnmente por medio de un servidor HTTP.



TIPOS DE API

Entre los tipos de APIs de servicios web más conocidos encontramos:

- **SOAP** (Simple Object Access Protocol): este es un protocolo que utiliza XML, definiendo la estructura de los mensajes y los métodos de comunicación, y WSDL, o lenguaje de definición de servicios web, en un documento legible por máquina para publicar una definición de su interfaz.
- **XML-RPC**: este es un protocolo que utiliza un formato XML específico para transferir datos. También es más antiguo que SOAP. XML-RPC utiliza un ancho de banda mínimo y es mucho más simple que SOAP.
- **JSON-RPC**: este protocolo es similar al XML-RPC, pero en lugar de utilizar el formato XML para transferir datos, utiliza JSON.
- **REST** (Representational State Transfer): REST no es un protocolo como los otros servicios web, sino que es un conjunto de principios arquitectónicos. Un servicio REST debe tener ciertas características, incluidas las interfaces simples, que son recursos que se identifican fácilmente dentro de la solicitud y la manipulación de los recursos que utiliza la interfaz.

¿Qué es?

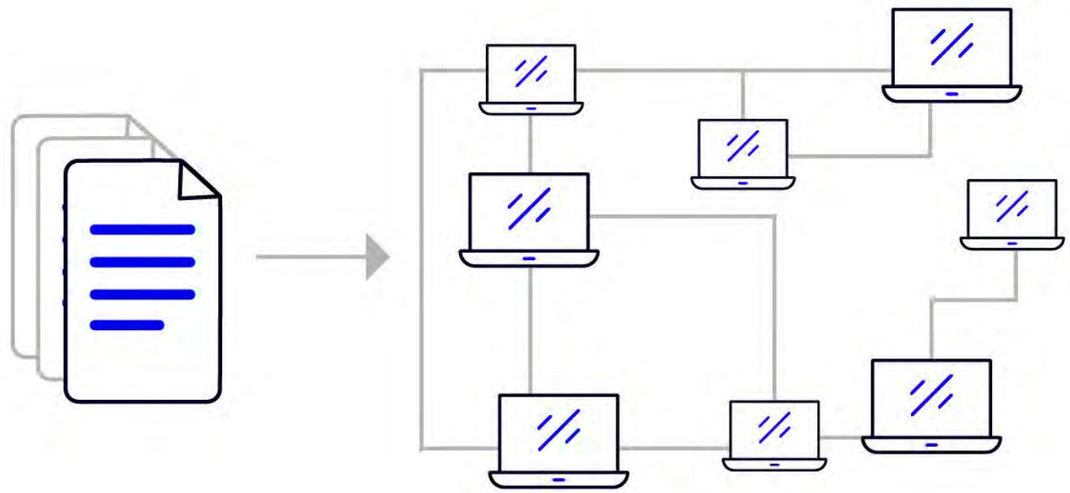
La infraestructura como código (IaC) es la gestión de la infraestructura (redes, máquinas virtuales, balanceadores, etc.) mediante un **modelo descriptivo y usando herramientas de control de versiones**.

CONCEPTO

De igual modo que un mismo código fuente genera siempre el mismo binario, **un modelo de IaC genera el mismo entorno cada vez que se aplica**. Es clave en la práctica DevOps y se usa en conjunto con despliegue continuo.

Sin IaC los equipos deben mantener la configuración de cada entorno de despliegue por separado. Con el paso del tiempo cada entorno evoluciona y se va volviendo más difícil de mantener. Estas inconsistencias entre los entornos dan lugar a errores en los despliegues.

Con IaC los equipos hacen cambios en los entornos en un archivo de configuración, que normalmente tiene formato JSON, YAML o similar. Cuando se registra este cambio en el control de versiones hay un sistema de integración que genera los entornos tal y como se describen en el archivo del modelo. En definitiva, los cambios se hacen en el modelo, nunca directamente en los entornos.



BENEFICIOS

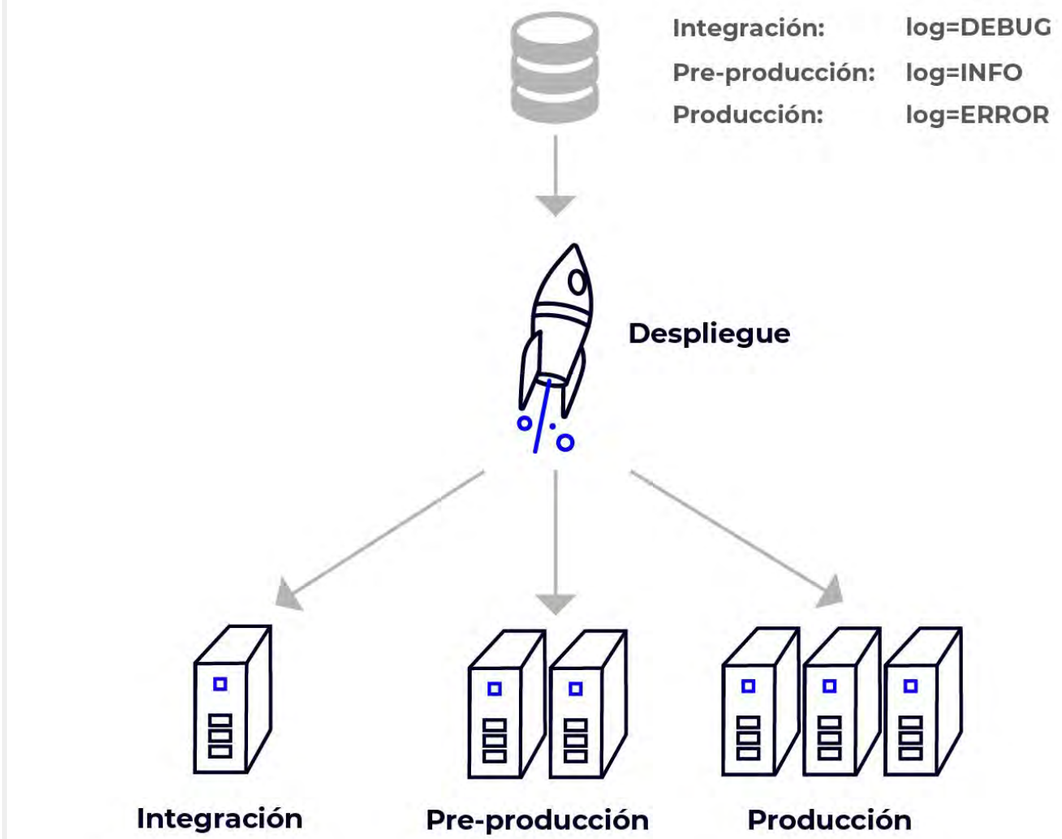
- Facilidad para probar las aplicaciones en **entornos parecidos al de producción** pronto en el desarrollo.
- La representación como código **permite que la configuración se valide y pruebe** y así evitar problemas comunes en el despliegue.
- **Evita la configuración manual** de los entornos, que es propensa a errores.
- Es más fácil que **diferentes equipos pueden trabajar juntos** estableciendo una serie de prácticas y herramientas comunes.

¿De qué se trata?

La configuración como código (o CaC, Configuration as Code, en sus siglas en inglés) es una práctica que involucra el versionado de los parámetros que configuran nuestras aplicaciones en distintos entornos.

BENEFICIOS

- Se especifican claramente las propiedades necesarias para el funcionamiento de la aplicación en distintos entornos.
- Se mantiene un **histórico** de los cambios realizados. Si se requiere desplegar una versión más antigua del software, se puede volver a una versión anterior en el histórico.
- Sirve de plantilla para **replicar una configuración** a través de distintas máquinas en un mismo entorno, facilitando el despliegue de servicios distribuidos.
- Proporciona una **fuente única de información**, sirviendo como documentación verídica del estado de configuración en todos los entornos.
- Reduce los tiempos de despliegue de la aplicación, ya que la configuración se puede hacer de manera automática.
- Si hay que cambiar una propiedad en un entorno, no hay que entrar a cada servidor del entorno a realizar el cambio. La modificación se hace en un solo lugar, **reduciendo el margen de error humano**.



Definición

Las bases de datos noSQL manejan grandes cantidades de información no estructurada, almacenando los datos de diferentes formas (documentos, grafos o colecciones de clave-valor, etc.)

VENTAJAS

- Asegura que **la base de datos no se convierta en un cuello de botella** si la cantidad de datos aumenta.
- **Almacena grandes cantidades de datos desestructurados**, pudiendo almacenar cualquier dato sin establecer restricciones por su tipo o estructura.
- Las bases de datos NoSQL **permiten un escalado horizontal en múltiples nodos o servidores** de forma inmediata y sin problemas.
- Debido a su naturaleza no relacional, **no necesita un modelo de datos muy detallado, lo que ahorra tiempo de desarrollo**.

DESVENTAJAS

- La comunidad **NoSQL carece de madurez**, ya que es relativamente nueva frente a los RDBMS de SQL.
- La falta o **escasez de herramientas para generar informes** sobre el análisis y pruebas de rendimiento en noSQL frente a la amplia gama que encontramos en SQL.
- En el lenguaje de consulta en las bases de datos noSQL usa sus propias características, por lo que **no es 100% compatible con el lenguaje SQL** utilizado en las bases de datos relacionales.
- Hay muchas bases de datos NoSQL y **una falta de estandarización** en ellas, pudiendo causar un problemas en las migraciones.

Algunas de las bases de datos noSQL más utilizadas son:

- Redis
- Neo4j
- CouchDB
- Postgres
- Hbase
- MongoDB
- BigTable
- Cassandra

Sistemas de versionado de BBDD

autentia

¿Qué son?

Herramientas de migración que permiten tener un control de versión de nuestra base de datos. Se pueden definir versiones nuevas por cada cambio para que nuestros compañeros de trabajo puedan integrar fácilmente estas actualizaciones.

¿EN QUÉ CONSISTE?

Cuando trabajamos con múltiples desarrolladores, puede ser difícil manejar nuestro esquema de base de datos cuando la aplicación crece. Hacer un seguimiento de todos estos cambios y fusionar esas nuevas versiones podría convertirse en una tarea incómoda.

Herramientas como **Flyway** o **Liquibase** permiten gestionar dicho seguimiento y ofrecen las siguientes ventajas:

- **Baselining**: si el esquema de base de datos ya existe (porque se ha integrado en un proyecto que ya está en desarrollo), permite crear las siguientes versiones a partir del esquema existente.
- **Tablas de control de versiones.**
- **Implementación de scripts incrementales** (se ejecutan solo en caso de que no se hayan aplicado previamente.)
- **Versionado por entornos.**
- **Sincronización** con otros desarrolladores.

VERSIONADO CON FLYWAY

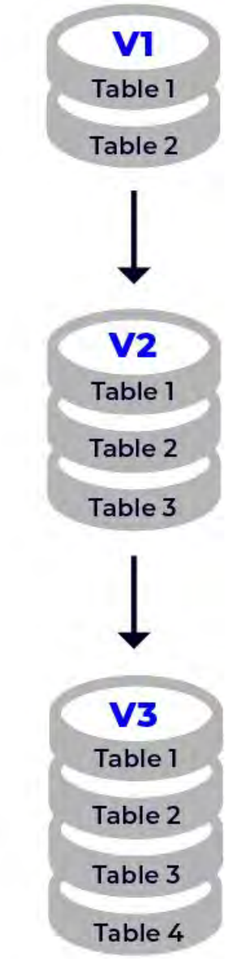
Flyway usa prefijos para conocer el orden en el que se ejecutarán los scripts, seguido de dos guiones bajos y una descripción de lo que se está haciendo.

Por ejemplo:

V1__init _tables.sql

V2__add_age_column_to_user.sql

Una vez definidos las versiones o scripts a ejecutar, arrancamos la aplicación y Flyway crea **flyway_schema_history**. Esta es una tabla que guarda los nuevos registros para los scripts sql que se han ejecutado. Cada vez que ejecutemos un nuevo script, esta tabla se actualizará.



Caché Distribuida

autentia

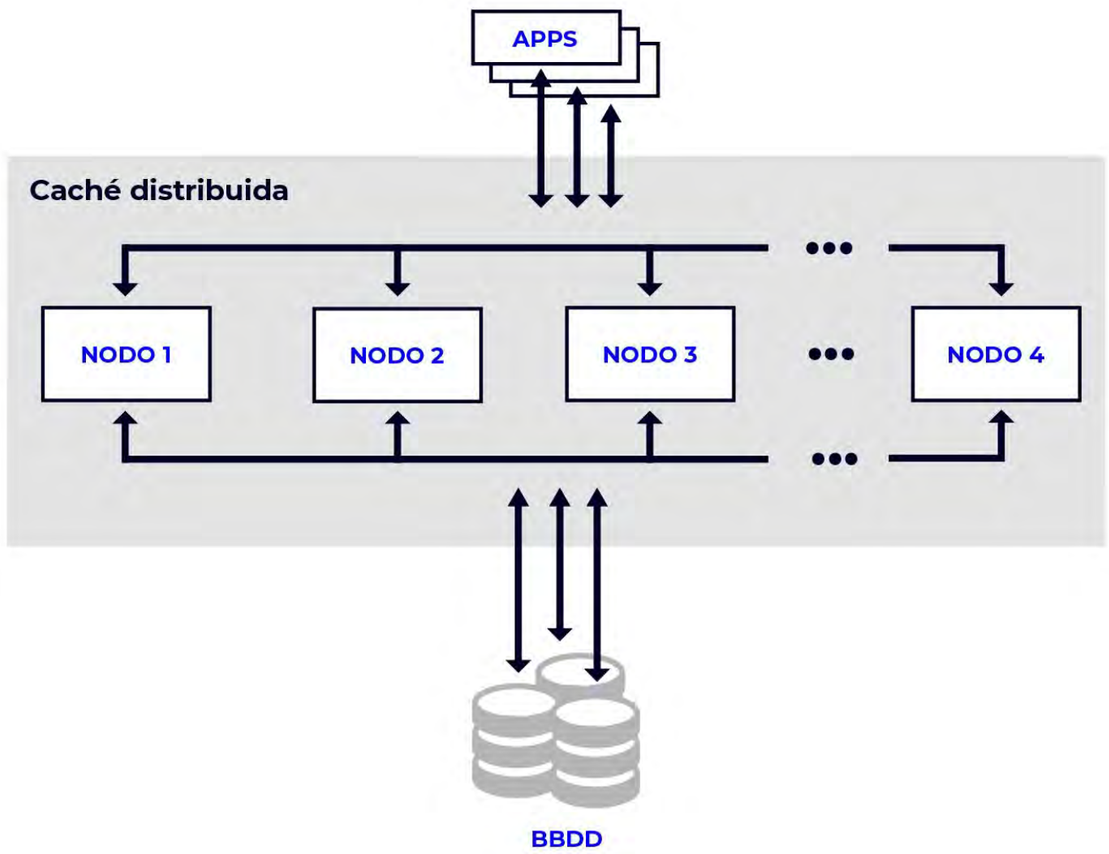
¿Qué es?

Memoria compartida por varios servidores o nodos dentro de la misma red y que almacena información relacionada. Permite que los recursos estén disponibles de manera más rápida, lo que mejora la escalabilidad y el rendimiento.

¿EN QUÉ CONSISTE?

Normalmente una caché tradicional se encuentra en un único servidor físico o componente hardware y esto limita su uso en un sistema distribuido.

Las cachés distribuidas son realmente útiles en entornos con un alto volumen y carga de datos e intentan solventar los cuellos de botella que producen los sistemas tradicionales. **Tienen la ventaja de escalar incrementalmente** al agregar más ordenadores al clúster, lo que permite que la caché crezca al ritmo del crecimiento de los datos.



VENTAJAS

- **Distribuida**: si un nodo falla no se va a dejar de prestar servicio ya que otro nodo puede atender dicha petición.
- Almacenamiento de **datos complejos.**
- **Escalabilidad y mejora del rendimiento** cuando aumenta la carga de transacciones.
- **Disponibilidad de los datos** durante un tiempo de inactividad no planificado.

CDN

autentia

¿Qué es?

CDN (Content Delivery Network) es una red de distribución de contenidos. Cuando tenemos un servicio web o una página web que es accedida desde distintas partes del mundo, con una CDN podemos conseguir que los usuarios de un continente no tengan que ir a servidores de otro continente a por nuestra página web.

¿EN QUÉ CONSISTE?

Consiste en una **red** en la que los contenidos se van a **cachear** y descargar desde servidores propiedad de la CDN. Si por ejemplo un usuario trata de acceder desde Brasil a nuestra web en España y tenemos una CDN contratada, los contenidos serán proporcionados por aquellos servidores que puedan **distribuirlo de manera más eficiente en función de la localización geográfica** de los mismos (probablemente por servidores situados en sudamérica).

VENTAJAS

- **Se reduce en gran medida la latencia** que supone por ejemplo, la distribución de contenidos entre continentes.
- **Aumenta el rendimiento de nuestra infraestructura**. Muchas de las peticiones serán servidas por la CDN sin siquiera llegar a nuestra infraestructura.
- Es una **defensa contra ataques DDoS** (Denial-of-service attack).

A TENER EN CUENTA

- Suele ser un **servicio que nos ofrece un tercero**, y en base al número de peticiones que nos ahorre, o al tamaño de lo que está cacheando nos va a cobrar una tarifa.
- Es un producto que típicamente **no requiere de mucha configuración** por nuestra parte.

Pruebas de rendimiento

autentia

¿Qué son?

Son una práctica que buscan **determinar la respuesta y estabilidad de un sistema** bajo una determinada carga de trabajo. Nos permiten ver qué partes del sistema tienen un peor rendimiento, encontrando cuellos de botella en nuestras aplicaciones.

¿EN QUÉ CONSISTEN?

Las pruebas de rendimiento se realizan utilizando una mezcla de software especializado (como Apache JMeter) y lenguajes de scripting para simular el uso real del sistema y monitorizar el uso de recursos.

Un parámetro importante es el de **throughtput** (producción) que es el número de transacciones que esperamos que nuestro sistema haga en un determinado tiempo. Otro parámetro a tener en cuenta es la **latencia**, que es el tiempo que tarda una petición de un nodo del sistema en ser respondida por otro. Un ejemplo sería el tiempo que tarda en recibir una respuesta un navegador al enviar una petición a un servidor web.

A partir de estos parámetros podemos definir qué tipo de pruebas queremos hacer y qué valores tomamos como objetivo para nuestro sistema. Por ejemplo, qué valores de latencia son aceptables o cuántos usuarios concurrentes esperamos que tenga nuestro sistema.

TIPOS

- Pruebas de **carga**: se somete al sistema a una cantidad fija de peticiones, que puede ser el número esperado de usuarios concurrentes y que realizan un número específico de transacciones.
- Pruebas de **estrés**: el número de usuarios se va aumentando hasta que se alcanza el límite del sistema y deja de funcionar.
- Pruebas de **estabilidad**: se aplica una carga que se mantiene en el tiempo para observar el comportamiento del sistema. Sirve para determinar si hay degradación en el rendimiento del sistema bajo cargas continuas (fugas de memoria, aumento de la latencia, etc.).
- Pruebas de **picos**: se trata de observar el comportamiento del sistema cuando hay cambios bruscos en la carga a la que está sometido.

¿En qué consiste?

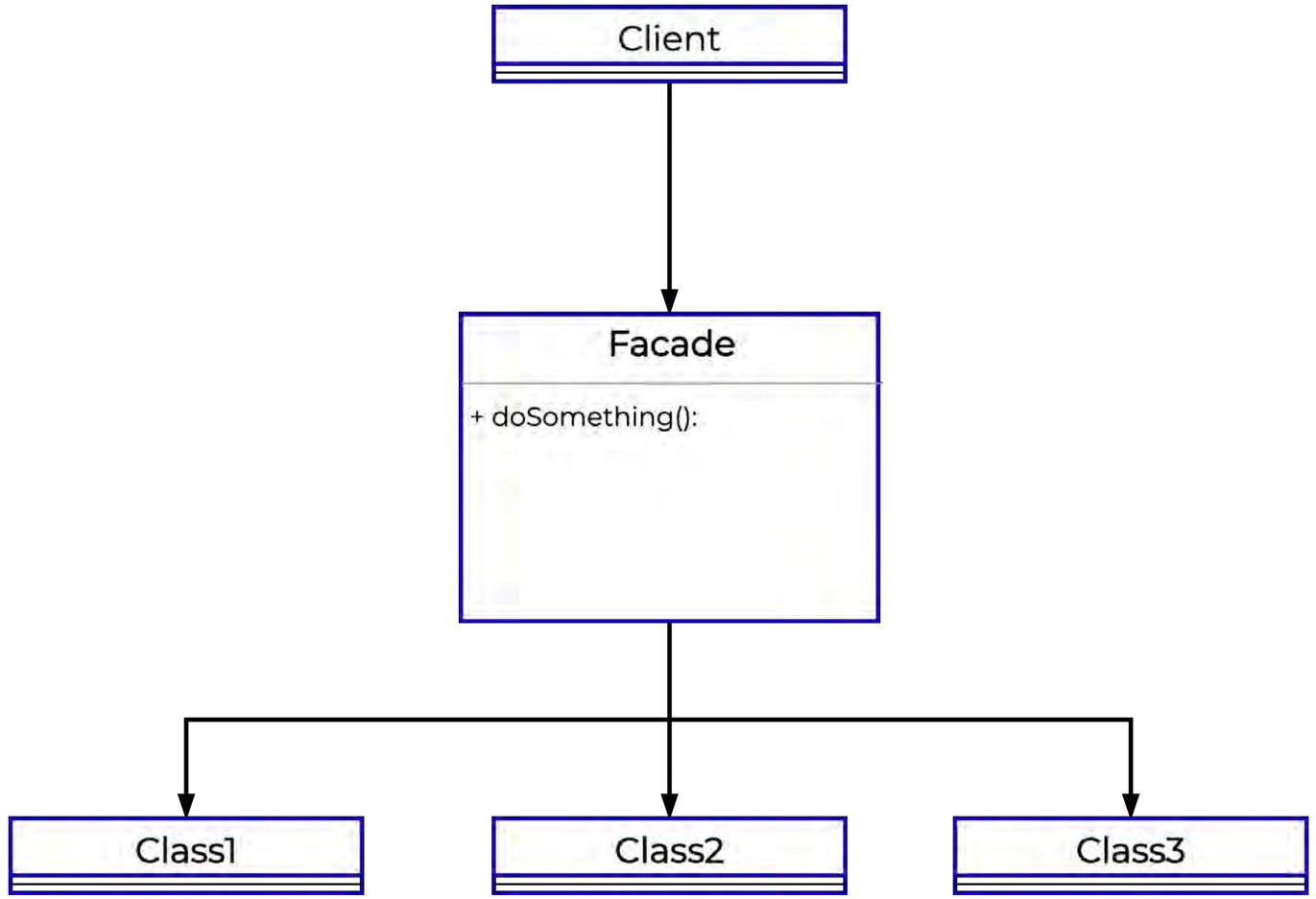
En inglés Facade, es un patrón estructural que provee una **interfaz simplificada (fachada)** a una librería, un framework o un conjunto de clases ofreciendo las funcionalidades que demanda el cliente.

RAZONAMIENTO

Una fachada es útil cuando estamos usando una librería compleja con cientos de funcionalidades pero realmente solo necesitamos ofrecer unas pocas o cuando tenemos varias clases y a partir de estas queremos obtener un único resultado.

La fachada es un intermediario que permite simplificar esta funcionalidad al cliente y aislarlo de una complejidad mayor.

- **Client**: representa al sistema o servicio que quiere hacer uso de la clase compleja o el conjunto de subsistemas.
- **Facade**: ofrece la funcionalidad que demanda el cliente mediante una interfaz sencilla.
- **Class[X]:** conjunto de clases de las que depende la fachada y a las que se pretende dar un punto de acceso sencillo.



Definición

Site Reliability Engineering (SRE) es una disciplina que **incorpora aspectos de la ingeniería de software y los aplica a problemas de infraestructura y operaciones**. Los objetivos principales son crear sistemas de software escalables y altamente confiables.

¿EN QUÉ CONSISTE?

SRE trata los problemas de operaciones como si se tratase de un problema de software. **Su misión es proteger la estabilidad del sistema sin dejar de agregar y mejorar el software** detrás de todos los servicios, **vigilando constantemente la disponibilidad, latencia, rendimiento y consumo de recursos**.

Un ingeniero SRE tiene una estrecha relación con el departamento de operaciones. Velando porque los procesos de supervisión del sistema estén lo más automatizados posible, facilitando la incorporación de nuevas características, el escalado del sistema, así como la detección de problemas y su tolerancia a fallos.

¿CÓMO FUNCIONA?

- Para cada servicio, el equipo de SRE establece un Acuerdo de Nivel de Servicio (SLA por sus siglas en inglés) garantizando una tasa de disponibilidad del servicio a los usuario finales.
- Este SLA indica el tiempo que el sistema puede no estar disponible, y el equipo puede aprovecharlo de la manera que considere mejor (realizando más subidas, automatizar tareas, etc). Por el contrario, si han cumplido o excedido el SLA, se congelan los despliegues de nuevas versiones hasta que se reduzca la cantidad de errores a un nivel que permita reanudarlas.

Front:

HTML y CSS

¿Qué es?

CSS (Cascade Style Sheets) **es un lenguaje de diseño gráfico que nos permite definir la apariencia visual de los elementos HTML** que componen las interfaces web.

¿EN QUÉ CONSISTE?

Junto con HTML y JavaScript, CSS permite crear interfaces web y GUIs de dispositivos móviles visualmente atractivas.

CSS **está pensado para definir el estilo** de una página web, incluyendo **diseño, layout, fuentes y variaciones en la interfaz,** separándolo del contenido de ésta. Gracias al uso de estilos CSS, podremos dar distintas apariencias a una misma página HTML.

Los estilos CSS se definen dentro de las hojas de estilo (.css), aunque pueden incluirse directamente dentro del HTML. Normalmente las hojas de estilos son ficheros independientes de modo que puedan reutilizarse en distintas interfaces para dar una apariencia común.

Como cualquier lenguaje, CSS ha ido evolucionando con el paso del tiempo. La versión más reciente es la conocida como **CSS3. Su especificación** es mantenida por el *World Wide Web Consortium (***W3C***)*.

SINTAXIS

El código CSS se compone de reglas. **Una regla es el conjunto de propiedades** que se van a aplicar a un elemento determinado.

En una regla **distinguimos el selector y la declaración**.

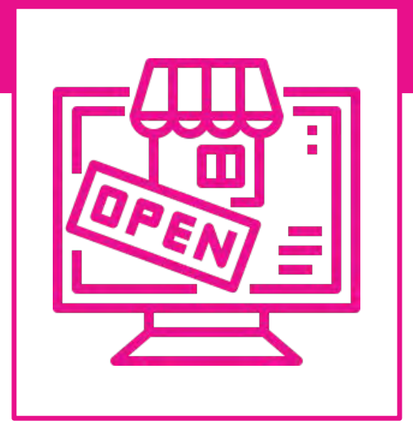
El **selector** nos permite referenciar los elementos que se quieren modificar. Se clasifican en:

- Selector de **etiqueta** <section> y el selector section {..}.
- Selector de **clase** * * y el selector *.relevant {...}*.
- Selectores de **id** * * y selector *#cabecera {...}*
- Selector **descendente** como el del esquema, aplicando estilo a todos los *h2* incluidos en elemento *section*.

La **declaración engloba un listado de pares propiedad-valor** encerrado entre llaves {} y separados por punto y coma (;). Cada propiedad se separa de su valor por dos puntos (:).

Web Abierta. Open Web Platform

autentia



¿Qué es?

Open Web Platform es una colección de tecnologías abiertas y estándares desarrollados por organismos como W3C, Unicode Consortium, Internet Engineering Task Force, y Ecma International. Algunas de las tecnologías que cubre son HTML5, CSS, SVG, MathML, WAI-ARIA, ECMAScript, WebGL, etc.

PRINCIPIOS BÁSICOS

Open Web tiene los siguientes principios :

- **Evita ser controlado** por ninguna empresa u organización, ya que está descentralizado. Pertenece a cualquiera que quiera usarlo.
- **Aporta transparencia**, haciendo visibles todos los niveles desde el origen de documento hasta las URLs y las capas HTTP.
- **Capacidad de integración**. Debería ser posible integrar con facilidad y de manera segura una fuente externa de otro sitio.
- **Documentación y especificaciones abiertas.** Sin derechos de autor o patentes sobre las especificaciones.
- **Libertad de Uso**. Emplea las tecnologías abiertas tanto en los proyectos libres como privados.
- **Discurso abierto.** Fomentar el diálogo y la participación de millones de personas usando la web como hilo conductor.
- **Cadena de favores.** Ser integrante de la Open Web significa compartir lo aprendido con blogs, conferencias o el uso de tecnologías abiertas.

EL FUTURO Y SUS FUNDAMENTOS

Open Web Platform propone **una taxonomía con ocho fundamentos** en los que se centrará **la próxima generación de aplicaciones**. Cada fundamento representa un conjunto de servicios y **capacidades disponibles para todas las aplicaciones**. Los fundamentos a cubrir son:

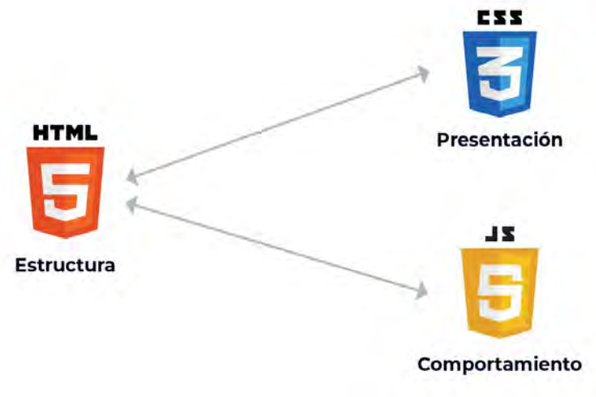
- Seguridad y privacidad.
- Diseño y desarrollo web central.
- Interacción del dispositivo.
- Ciclo de vida de la aplicación.
- Medios y comunicaciones en tiempo real.
- Rendimiento y afinación.
- Usabilidad y accesibilidad.
- Servicios.

¿Qué es?

HTML (Hyper Text Markup Language) es un lenguaje markup que sirve para describir la estructura de una página web, no el diseño, colores, etc., sino sólo la estructura haciendo uso de etiquetas para organizar el contenido.

HISTORIA Y EVOLUCIÓN

El HTML fue inventado por **Tim Berners-Lee** (físico del CERN) en 1989. Se le ocurrió la idea de un sistema de hipertexto (enlaces a contenido) en Internet. Necesitaba **crear documentos con referencias a otros para facilitar el acceso y la colaboración** de otros equipos. Desde esos días hasta hoy, lo que conocemos como HTML (HTML4 publicada W3C en 1999) ha tenido una evolución increíble en su última versión HTML5 (publicada en 2014), introduciendo nuevas características como etiquetas semánticas, incrustar audio y vídeo o soportar SVG y MathML para fórmulas matemáticas.



¿CÓMO FUNCIONA?

El contenido se guarda en archivos con extensión .html o .htm y se ve a través de cualquier navegador.

Cada página HTML consta de un **conjunto de etiquetas**, que representan los componentes básicos de la página web. Con ellos creamos una jerarquía que **estructura el contenido** en secciones, párrafos, encabezados y otros bloques de contenido. La mayoría de los elementos HTML tienen **una apertura y un cierre** que utilizan la sintaxis <tag> </tag>. Un ejemplo de estructura HTML podría ser:

```
<div>
   <p> Este es el primer párrafo </p>
   <p> Este es el segundo párrafo </p>
</div>
```

VENTAJAS E INCONVENIENTES

Ventajas:

- Lenguaje ampliamente utilizado por la comunidad.
- Se ejecuta de forma nativa en todos los navegadores.
- Lenguaje limpio, consistente y sencillo.
- La especificación sigue un estándar mantenido por la W3C.
- Tiene una fácil integración con lenguajes Backend.

Inconvenientes:

- Páginas estáticas. Necesitando de JavaScript para hacerlas dinámicas..
- No permite implementar lógica o comportamiento.

HTML necesita de CSS y JavaScript para implementar algunas funciones. Se puede pensar en HTML como el esqueleto de un persona, CSS como la piel, pelo, etc. y JavaScript los músculos.

Árbol DOM

autentia

¿Qué es?

DOM (Document Object Model) es una interfaz para documentos HTML y XML que se representa como un árbol de elementos. Permite leer y manipular el contenido, la estructura y los estilos de la página con un lenguaje de scripting como JavaScript.

RENDER TREE

La forma en que un navegador pasa de un documento HTML, a mostrar una página con estilo e interactiva, se denomina ***Critical Rendering Path***. Primero se establece que se va a renderizar y se denomina **render tree (DOM + CSSOM).** Luego, el navegador realiza el renderizado.

- CSSOM: representa los estilos asociados a los elementos.
- DOM: representa los elementos.

El *render tree* excluye los elementos que no están visibles como por ejemplo, los que tienen el estilo *display: none*. **El DOM si lo incluiría en su árbol de nodos**.

¿CÓMO MANIPULAR EL DOM?

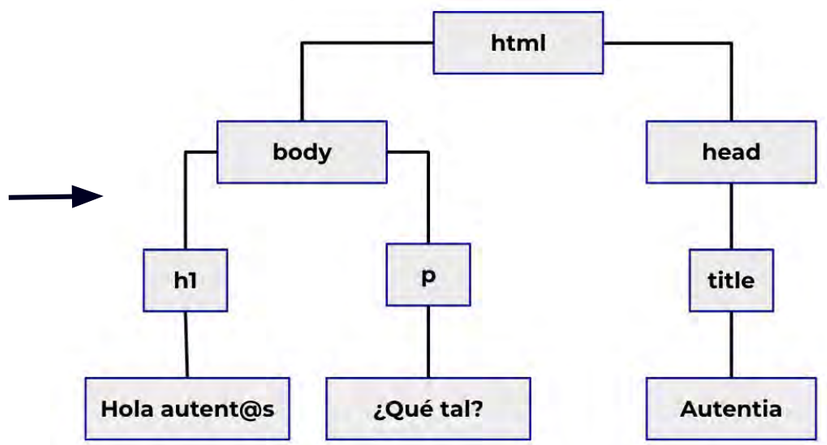
El DOM fue diseñado para ser independiente de cualquier lenguaje de programación, pero JavaScript es uno de los más populares para esta tarea. A través de la etiqueta <script>, se puede comenzar a manipular el documento o los elementos de la ventana.

Tenemos funciones como document.createElement, getElementbyid, window.alert, entre otras muchas.

ÁRBOL DE NODOS

El objetivo del DOM es convertir la estructura y el contenido del documento HTML en un modelo de objeto que puede ser utilizado por varios programas. La estructura del documento es conocida como un **árbol de nodos (node tree).** El elemento raíz es la etiqueta *html*, las ramas son los elementos anidados y las hojas serían el contenido de esos elementos.

```html
<!doctype html>
<html lang="es">
 <head>
   <title>Autentia</title>
  </head>
 <body>
    <h1>Hola autenti@s</h1>
    <p>¿Qué tal?</p>
  </body>
</html>
```



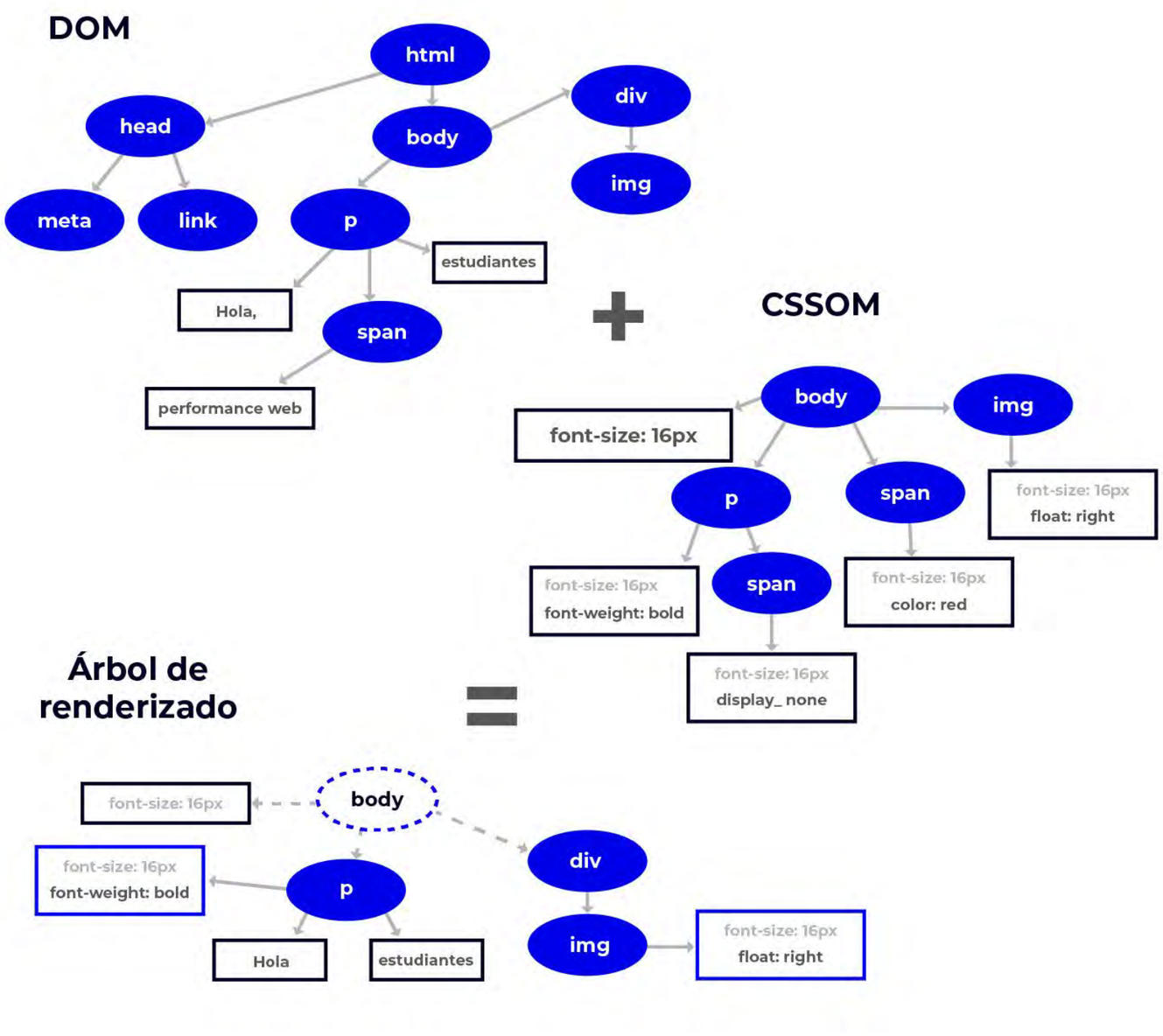
¿Qué es?

También referido como **Critical Rendering Path** (ruta de renderizado crítica) es la **secuencia de pasos** que sigue el navegador **para convertir HTML, CSS y JavaScript en píxeles en la pantalla**. Esta secuencia de pasos es realizada por el motor de renderizado del navegador.

¿EN QUÉ CONSISTE?

Una solicitud de una página web o aplicación comienza con una petición HTML. Al realizar una solicitud, el servidor devuelve los encabezados y datos de respuesta.

1. Se crea el **Document Object Model** (DOM) a partir de la respuesta HTML. También inicia solicitudes cada vez que encuentra enlaces a recursos externos, ya sean hojas de estilo, scripts o referencias de imágenes incrustadas.
    a. Algunas solicitudes son bloqueantes, lo que significa que el parseo del resto del HTML se detiene hasta que se cargue el recurso.
2. Se crea el **CSS Object Model** (CSSOM).
3. Cuando tiene el DOM y el CSSOM listos, se combinan en el **árbol de renderizado** (Render Tree), obteniendo los estilos para todo el contenido visible.
4. Una vez que se completa el árbol de procesamiento, el diseño calcula la posición y el tamaño exactos de cada objeto (**layout**) del árbol de procesamiento.
5. Una vez completado, se procede a la **representación** o “pintado”, que consiste en tomar el árbol de renderizado final y renderizar los píxeles en la pantalla.



¿Qué es?

Al igual que HTTP, WebSocket es un **protocolo de comunicación que permite realizar una conexión bidireccional entre dos dispositivos**, un cliente y un servidor.

¿CÓMO FUNCIONA?

Websocket es un **protocolo con estado (stateful)** que mantiene la conexión abierta hasta que el cliente o servidor decida cerrarla. El cliente comienza una comunicación a través de un proceso llamado **handshake,** esto es basicamente una petición http al servidor. Con la cabecera **Upgrade** informamos al servidor que deseamos establecer una conexión websocket. Además, usamos **ws** o **wss** en vez de http o https.

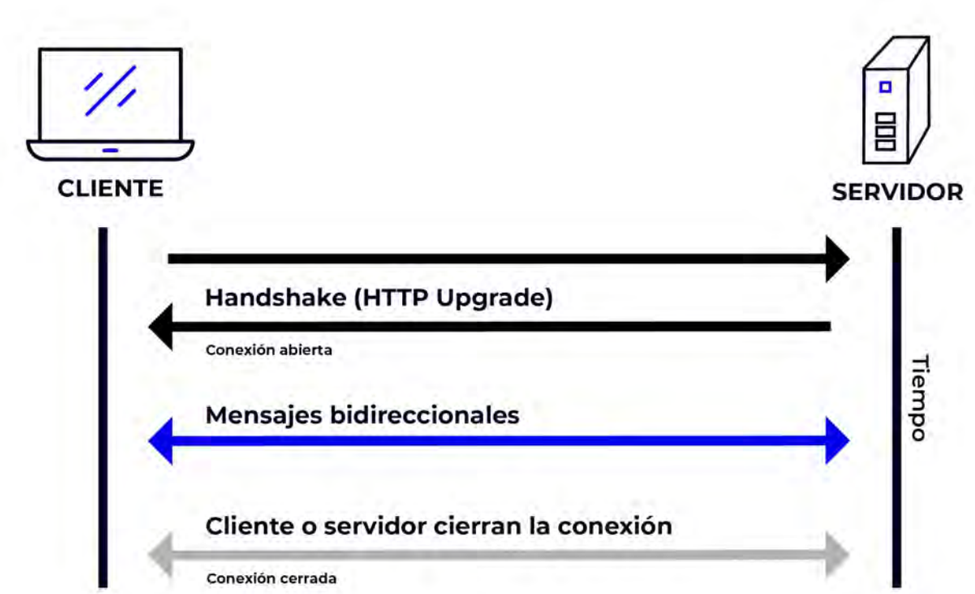
```
GET ws://autentia.com/ HTTP/1.1
Origin: http://autentia.com
Connection: Upgrade
Host:server.autentia.com
Upgrade: websocket
```

Si el servidor soporta el protocolo websocket, responderá con la cabecera **Upgrade** y de inmediato se podrá enviar o recibir datos de forma bidireccional.

```
HTTP/1.1 101 WebSocket Protocol Handshake
Date: Mon, 15 Jun 2020 10:07:34 GMT
Connection: Upgrade
Upgrade: WebSocket
```

¿DÓNDE SE USA?

- **Aplicaciones en tiempo real:** un buen ejemplo podrían ser las aplicaciones de trading, donde se necesita el precio del bitcoin u otro activo de forma casi exacta.
- **Aplicaciones sobre juegos (gaming):** el servidor está constantemente enviando datos sin tener que refrescar la vista (UI).
- **Chats:** al abrir la conexión una sola vez, es perfecto para enviar y recibir mensajes en una conversación.



JavaScript asíncrono

Los *web workers* son scripts escritos en JavaScript que se ejecutan de forma paralela y en segundo plano al procesamiento de la interfaz gráfica. Podemos calcular o solicitar información sin que el usuario perciba una interrupción visual o funcional.

CREACIÓN

Para crear un *web worker* se utiliza el constructor, asígnele el objeto creado a una variable. Se puede hacer dentro de cualquier script de JavaScript.

```
let myWorker = new Worker('worker.js');
```

El *Worker* utilizará el script 'worker.js' para su funcionamiento.

COMUNICACIÓN

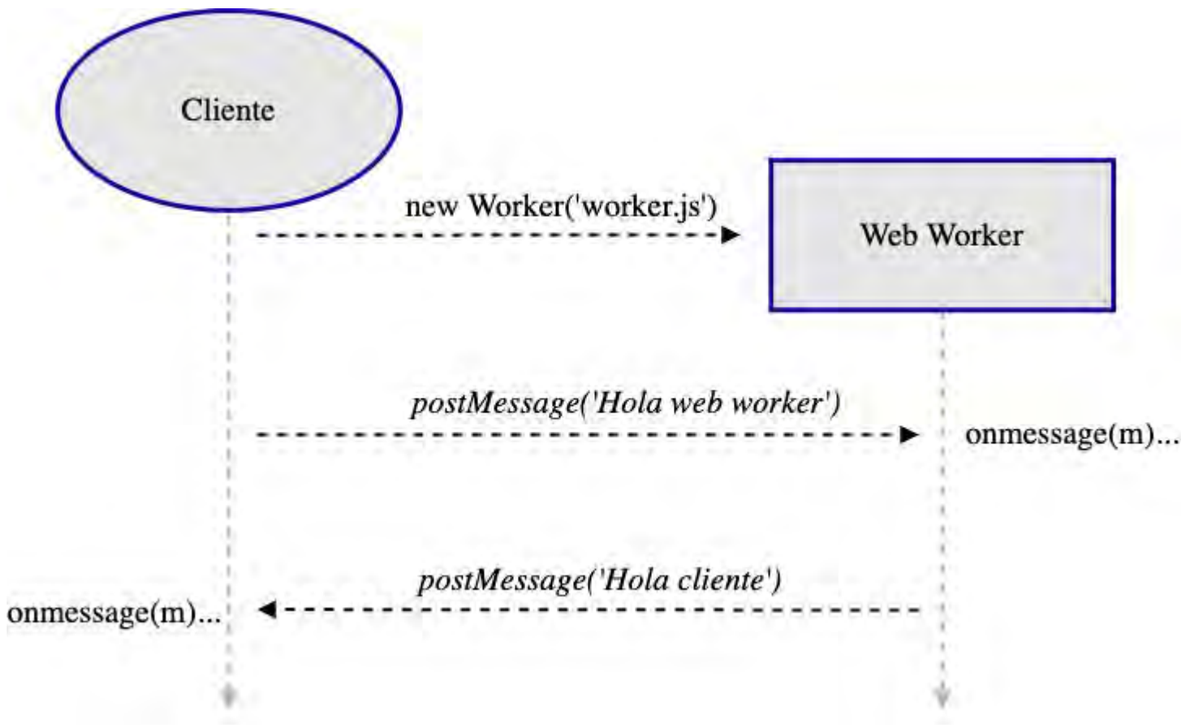
Una vez inicializado, tanto el *worker* como su cliente se comunicarán a través del método *postMessage* y el manejador de eventos *onmessage*.

client.js

```
myWorker.postMessage('Hola web worker');
myWorker.onmessage = function(message) {
    console.log('Worker dice: ' + message);
}
```

worker.js

```
onmessage = function(message) {
    console.log('Cliente dice: ' + message);
    postMessage('Hola cliente');
}
```



LIMITACIONES

1. Los elementos del DOM no son manipulables dentro del contexto de un *web worker*.
2. El estándar que define la API para *web workers* considera su inicialización como "relativamente costosa", con lo cual se debe cuidar de no crear muchos.

¿Qué es?

Search Engine Optimization (SEO) o en español, optimización en motores de búsqueda, es un proceso que pretende mejorar el posicionamiento de una web en los resultados de los motores de búsqueda, como Google o Bing.

CONCEPTO

Search Engine Optimization (SEO) es un conjunto de técnicas para la optimización del posicionamiento en buscadores. Mediante el SEO, un sitio web aparece en más resultados naturales y se aumenta la calidad y cantidad del tráfico. Puede optimizarse el resultado en búsquedas de imágenes, vídeos, artículos académicos, compras, etc.

Hay que diferenciar los **resultados "orgánicos" o "naturales",** que se consiguen porque el motor de búsqueda considera que son relevantes a la búsqueda del usuario, de los resultados pagados que son campañas de marketing dirigidas a un público.

La optimización tiene dos partes:

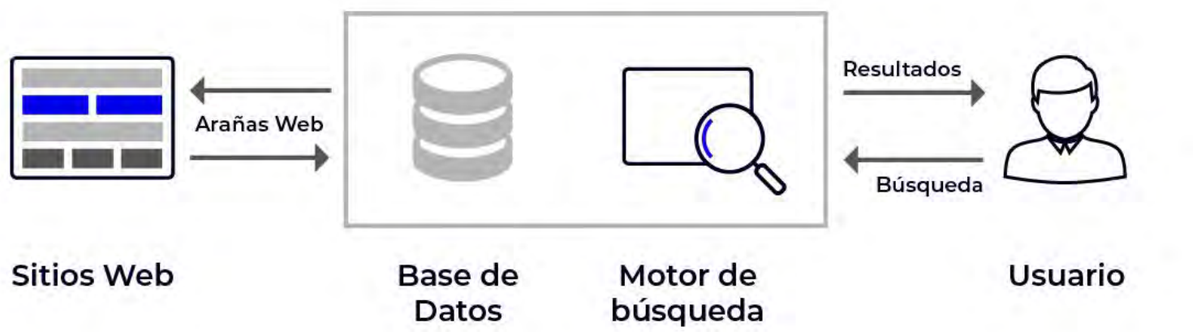
- Optimización interna: se trabaja tanto con **elementos técnicos de la web** (estructura HTML y metadatos), **como con el contenido interno** para hacerlo más relevante al usuario.
- Optimización externa: se mejora la **notoriedad de la página** web al aparecer referencias a ella en otros sitios (enlaces naturales y redes sociales).

¿CÓMO FUNCIONAN LOS BUSCADORES?

Los motores de búsqueda recorren los sitios mediante **arañas web**, navegando a través de las páginas y analizando su estructura y contenido. Las arañas solo analizan un número determinado de páginas (o se detienen un tiempo máximo) dentro del sitio, antes de pasar al siguiente. Los motores de búsqueda recorren cada sitio de forma periódica para mantenerse actualizados.

Una vez las arañas han analizado un sitio, lo indexan, clasificándolo según su contenido y relevancia. A partir de este índice, el motor de búsqueda va a poder mostrar la página en los resultados.

Además de este análisis, los buscadores **priorizan resultados que a otros usuarios con un perfil similar les han resultado útiles.**



¿Qué son?

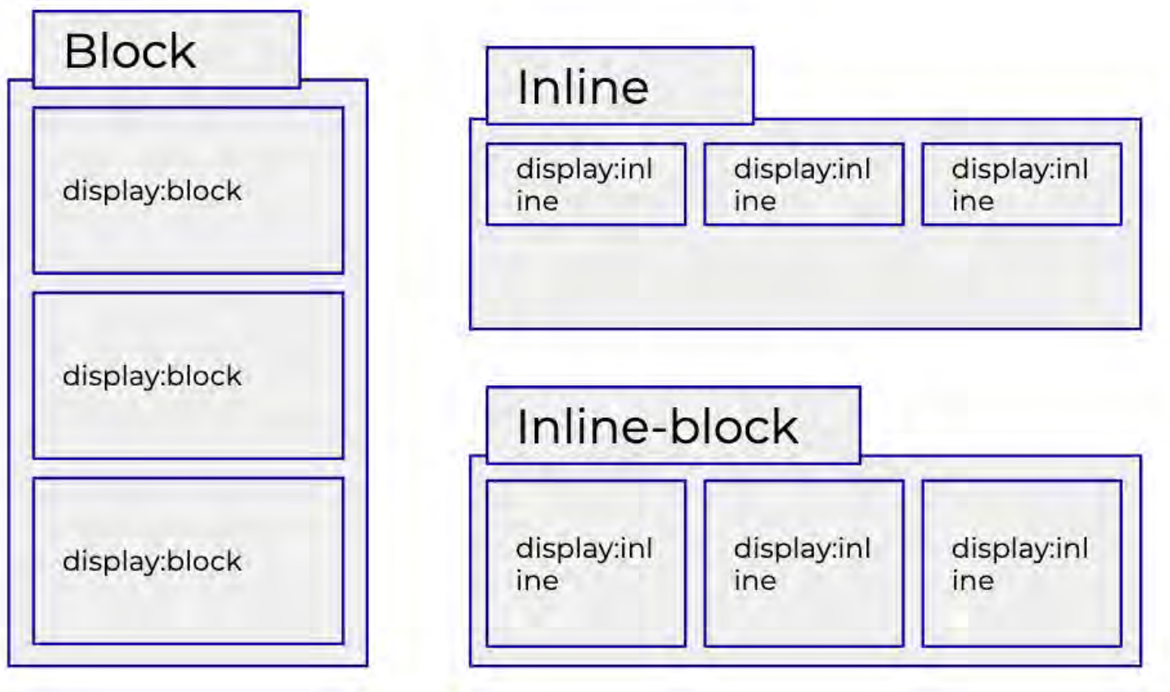
Las propiedades block, inline e inline-block **modifican cómo el cuadro de un elemento HTML se muestra** en la página. Cada elemento HTML tiene un valor de display por defecto, aunque se puede sobreescribir.

DIFERENCIAS

La propiedad CSS **display** tiene dos funciones: cómo el cuadro del propio elemento se muestra y cómo se muestran los hijos del elemento. Para la primera de ellas, tenemos los siguientes valores:

- **Block:** un elemento block siempre empieza en una nueva línea y toma todo el ancho disponible. Un ejemplo de una etiqueta block es div.
- **Inline:** un elemento inline no empieza en una nueva línea y solo trata de ocupar el ancho necesario. Un ejemplo de una etiqueta inline es span.
- **Inline-block:** la diferencia con inline es que inline-block permite establecer un ancho y altura en el elemento. Además se respetan los valores de margin/padding del elemento. La diferencia con block es que no se añade un salto de línea después del elemento, de forma que puede tener elementos a continuación.

Otro valor común es **none**, que hace que el elemento no se muestre y no ocupe espacio en la página.



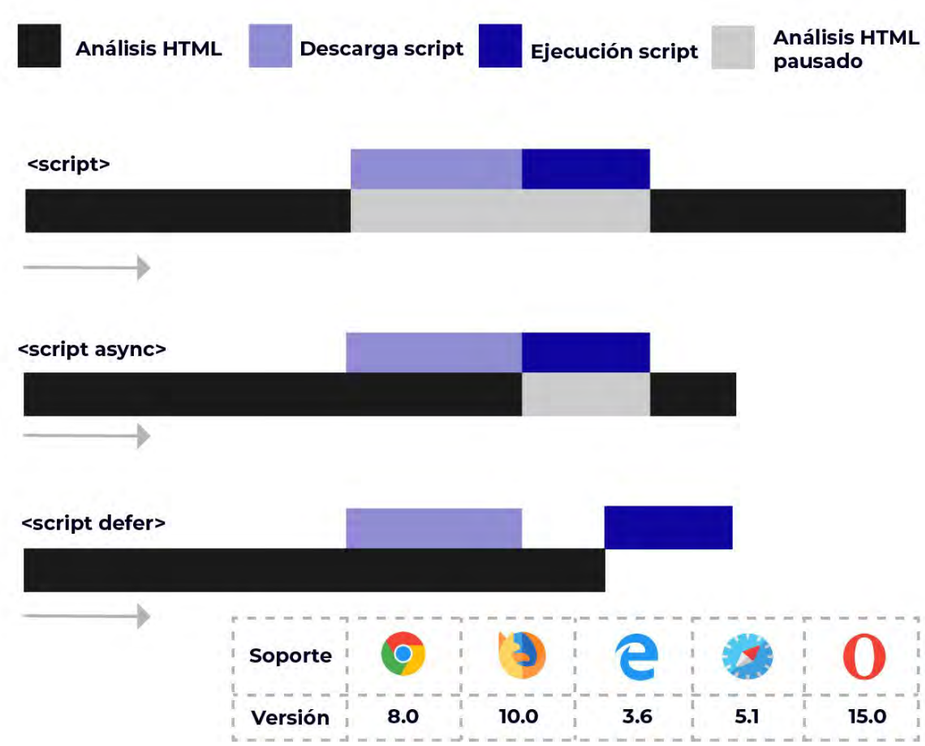
¿Qué problema soluciona?

Los elementos <script> bloquean el análisis y renderizado del HTML de la página. Cuando el navegador encuentra un recurso de este tipo, detiene el análisis de HTML, descarga el recurso, lo ejecuta y prosigue donde lo dejó. HTML5 nos ofrece **async y defer para evitar bloqueos antes de renderizar la página**.

RENDERIZADO Y CARGA DE SCRIPT

Antes de la aparición de estos atributos, se recomendaba colocar los elementos <script> al final del HTML, para que cuando el analizador se los encontrase, ya hubiese analizado y renderizado todo el documento. Los atributos async y defer nos ofrecen **una solución a este problema, sin forzarnos a recolocar los scripts**, con algunas diferencias entre ellos, que son:

- **<script>** (normal): bloquea el análisis y lo reanuda una vez ejecutado.
- **<script async>**: el script se descarga de forma asíncrona pero se sigue bloqueando al ejecutarse. No garantiza la ejecución de los scripts asíncronos en el mismo orden en el que aparecen en el documento.
- **<script defer>**: el script se descarga de forma asíncrona, en paralelo con el análisis HTML, esperando a que termine el análisis HTML para realizar la ejecución. No hay bloqueo en el renderizado HTML. La ejecución de todos los scripts se realiza en orden de aparición.



¿CUÁL USO Y CUÁNDO?

- **defer** parece la mejor opción de forma general. Siempre que **el script a ejecutar no manipule o interaccione con el DOM antes de que se renderice**. También es la mejor opción si el script **tiene dependencias con otros scripts e importa el orden de ejecución**.
- **async** se recomienda para **scripts que manipulan o interaccionan con el DOM antes de su carga y/o no tienen dependencias** con otros scripts.
- **El uso normal, sólo si el script es pequeño**, ya que la parada del análisis HTML será insignificante en comparación a la descarga del script.

¿En qué consiste?

Cada navegador implementa los estándares de manera distinta. Esto puede dar lugar a que los usuarios tengan una experiencia diferente en función del navegador que están usando.

CONCEPTO

La compatibilidad entre navegadores es un concepto a tener en cuenta a la hora de desarrollar aplicaciones web, debido a que para una misma web, **usar distintos navegadores puede resultar en una experiencia distinta**. Puede darse el caso extremo en el que exista incompatibilidad con un navegador, quedando limitada la audiencia de esa web.

Cuando el diseño se desajusta entre navegadores, puede ocurrir que el texto no quepa en la pantalla, que no sea visible la barra de scroll o que cierto código en JavaScript no se ejecute, etc. Como es inviable comprobar la compatibilidad entre todos los navegadores del mercado, merece la pena asegurarlo entre los que tienen más cuota de mercado, como Chrome, Firefox y Safari.

Hay distintas acciones que nos ayudan a asegurar esta compatibilidad, entre las que se encuentran:

- **Validar tanto el HTML como el CSS** de la web para que cumplan el estándar.
- **Resetear los estilos CSS**. Cada navegador tiene unos valores por defecto para ciertas propiedades, haciendo que algunos elementos se vean distintos.
- Usar **técnicas soportadas**. La web de Can I Use muestra la compatibilidad de funciones de la API de JavaScript para distintos navegadores.

DIFERENCIAS ENTRE NAVEGADORES

Hay dos piezas fundamentales en un navegador: por un lado el motor de renderizado (que analiza el código HTML y CSS) y por otro el motor de JavaScript.

Cada **navegador utiliza un motor distinto** que implementa los estándares con pequeñas diferencias. Además, esta implementación puede cambiar con las versiones y con el sistema operativo.

Es por esto que no todos los navegadores interpretan el HTML, CSS y JavaScript igual. Aunque a día de hoy las diferencias sean pequeñas, pueden hacer que un usuario no pueda ver correctamente la página.

Unidades de medida

autentia

¿Qué son?

Propiedades en CSS que se emplean para **establecer la disposición de los elementos en el documento**. Definen la altura, anchura y margen de los elementos a través de un valor numérico (entero o decimal), seguido de una unidad de medida.

UNIDADES ABSOLUTAS

Su valor real es directamente el valor indicado, por lo que no dependen de otros componentes para situar los elementos.

- **px** (píxeles).
- **cm** (centímetros).
- **mm** (milímetros).
- **in** (pulgadas): equivalente a 25,4 milímetros.
- **pt** (puntos): equivalente a 0,35 milímetros.
- **pc** (picas): equivalente a 4,23 milímetros.

No se recomienda utilizar unidades absolutas si queremos un diseño ***responsive,*** ya que al ser unidades fijas, no se adaptan de igual manera a todas las pantallas.

UNIDADES RELATIVAS

Su valor real está relacionado con otro elemento. Son las más utilizadas en el diseño web por la flexibilidad que ofrecen ya que se adaptan a diferentes pantallas si se usan correctamente.

- **em**: unidad relativa a la propiedad *font-size* del elemento padre.
  ```
  html { font-size: 13px }
  h1 { font-size: 2em } // 13px * 2 = 26px
  ```
  Si tuviésemos el elemento hijo 'span' dentro de h1, al ser h1 el padre, 1em = 26px
  ```
  span { font-size: 1.5em } // 26px * 1,5 = 39px
  ```
- **rem**: igual que la anterior, pero esta es relativa con respecto al *font-size* del elemento *root* (*root* es la etiqueta html). **En caso de no definirlo, toma el valor por defecto que son 16px**. En el ejemplo anterior, 1rem = 13px. Esto es muy útil porque si algún usuario decide cambiar el tamaño de letra por defecto del navegador, todos los elementos de nuestro árbol serán flexibles al cambio.
- **ch**: relativa al ancho del cero (0).
- **vw**: relativa al 1% del ancho del *viewport*. El *viewport* es el tamaño de la ventana del navegador.
- **vh**: igual que la anterior, pero relativa al 1% del alto del viewport.
- **%:** relativa al elemento padre.

Flexbox

autentia

¿Qué es?

Módulo de CSS (layout mode) unidimensional utilizado para **distribuir el espacio de los elementos en filas o columnas de una forma dinámica y sencilla** y que permite desarrollos *responsive* gracias a elementos flexibles que se adaptan automáticamente al contenedor.

PREVIO A FLEX

Antes de Flex, se usaban distintos modos para la disposición de los elementos (ojo, todavía se usan):

- **En línea** (display:inline).
- **En bloque** (display:block).
- **En tabla** (display:table).
- **Position** (static, relative, absolute, etc.).
- **Float** (right, left, inherit, etc.).

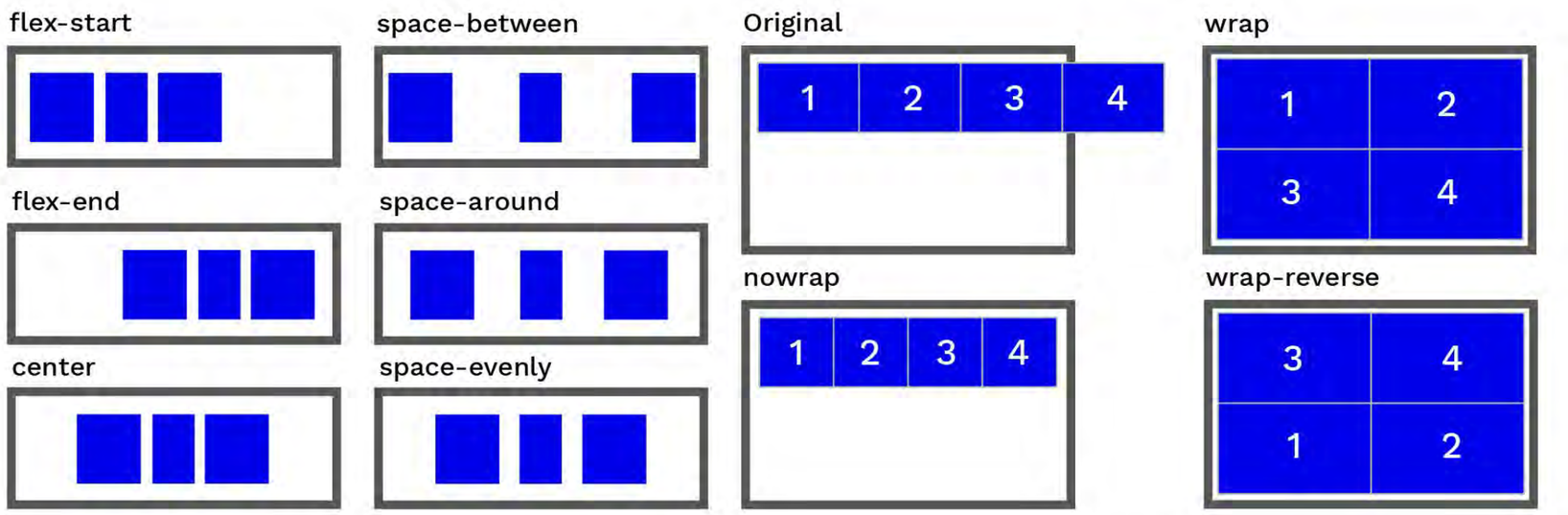
Flex es una mezcla de estas propiedades en cuanto a cómo afecta a la disposición de los elementos contenidos en un contenedor. Un diseño Flexbox consiste en un ***flex container*** que contiene elementos flexibles (**flex items)**.

PROPIEDADES MÁS USADAS

Para que un contenedor sea flexible, se usa la propiedad **display: flex.**

Al usar flexbox, el eje principal es el horizontal en caso de que *flex-direction* sea una fila, y vertical en caso de que sea una columna. El eje secundario será el perpendicular al principal.

- Alineamiento a lo largo del eje principal: **justify-content**: flex-start | flex-end | center, etc.
- Alineamiento a lo largo del eje secundario: **align-items:** flex-start | flex-end | center, etc.
- Dirección de los elementos (de izquierda a derecha, de arriba a bajo, etc.): **flex-direction:** column | row | row-inverse, etc.
- Por defecto, Flex intenta ajustar los elementos en una fila pero esto se puede modificar: **flex-wrap:** wrap | no-wrap | wrap-inverse, etc.



Grid Layout

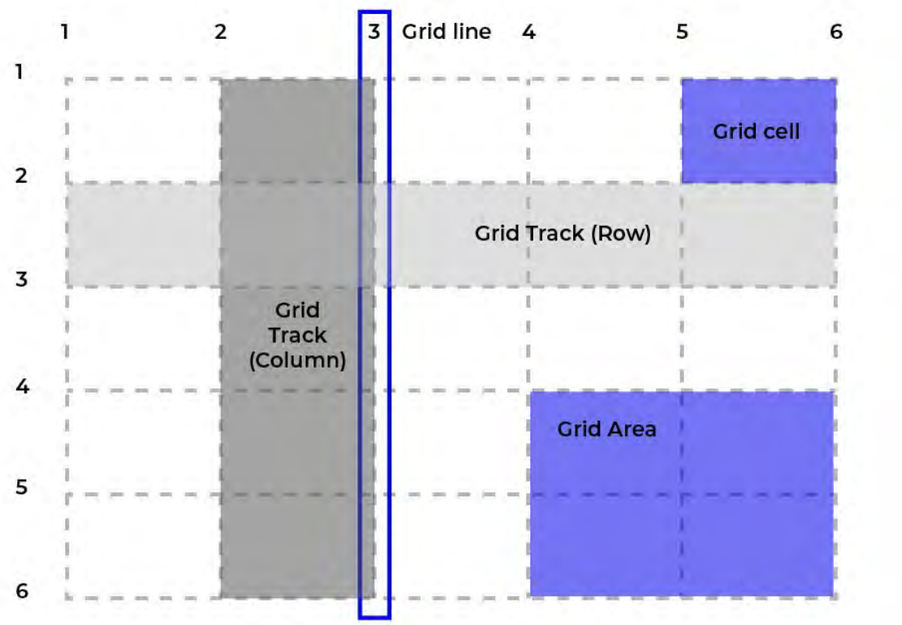
autentia

¿Qué es?

CSS Grid o Grid Layout, es un estándar de las Hojas de Estilo en Cascada que nos **permite maquetar contenido ajustándose a una rejilla** en dos dimensiones totalmente configurables mediante estilos CSS.

CONCEPTOS BÁSICOS

- **Grid Layout se compone de lineas** horizontales (para las filas) y verticales (para las columnas).
- El espacio delimitado entre dos líneas consecutivas se le llama **track**.
- Una vez que especificamos el número de filas y columnas, Grid Layout **numera las líneas automáticamente**.
- Una **celda** es el espacio que define la intersección de las líneas verticales y horizontales, teniendo el tamaño 1x1 en nuestra rejilla.
- Un **Grid** es el espacio que ocupa más de una celda en nuestra rejilla.



CARACTERÍSTICAS

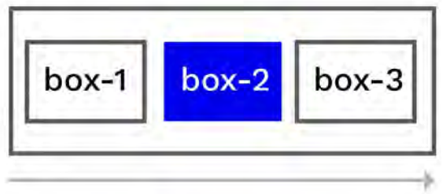
- Es parte de la especificación de CSS, por lo que **no hay problemas de incompatibilidades**.
- Nos permite **colocar los items sin tener que hacer trucos como** (margin:auto, position, etc.), ya que flexbox solo tiene una dimensión(columnas o filas).
- Los ítems cuya posición no se especifique **se colocaran solos** (de manera automática), gracias al algoritmo de **auto placement**, ya que Grid es una rejilla, **no es una tabla**.
- Nos ofrece una **sintaxis muy extensa** en su especificación.
- Nos facilita la creación de diseños complejos con layouts.
- **Grid Layout y Flexbox se pueden combinar**. Permitiéndonos contener dentro de un Grid una estructura hecha en Flexbox que sólo crece en una dirección.
- Un contenedor en Flexbox es el conjunto de ítems en una dirección, a diferencia de CSS Grid en el que **cada Grid es un contenedor**.

¿En qué consiste?

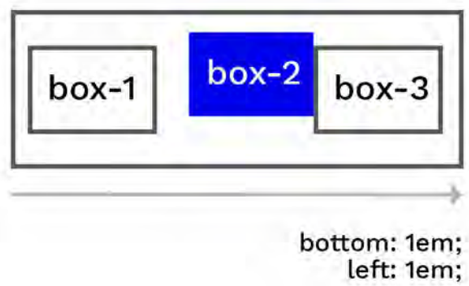
Es una propiedad de CSS llamada ***position***, cuya función determina cómo se posicionará un elemento en la página. Le acompañan además unas propiedades de desplazamiento que precisan con más detalle esta posición (*top*, *right*, *bottom*, *left* y *z-index*). **Un elemento *posicionado*** es aquel que tenga un *position* definido y cuyo tipo no sea *static*.

TIPOS

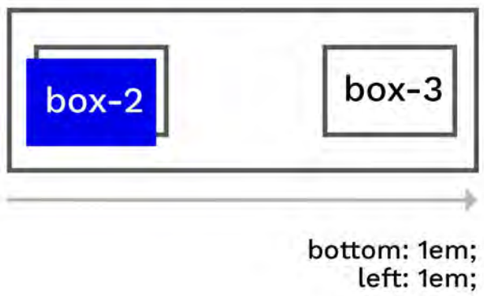
- **STATIC:** valor por defecto. Posiciona los elementos de acuerdo al flujo normal del documento y los elementos a su alrededor. Las propiedades de desplazamiento **no tienen efecto**.



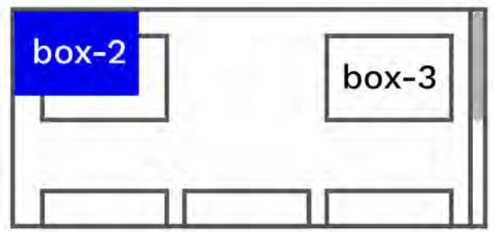
- **RELATIVE:** el elemento se posiciona como si fuese *static*, pero las distancias definidas en *top*, *right*, *bottom* y *left* desplazarán el elemento desde ese lado.

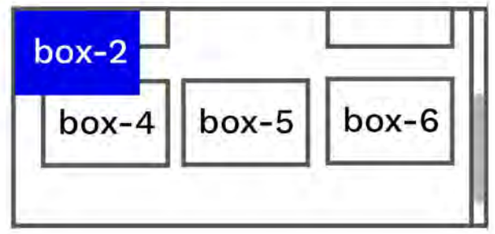


- **ABSOLUTE:** la posición del elemento se calcula con respecto al ancestro ***posicionado*** más cercano, o en su defecto al cuerpo del documento.

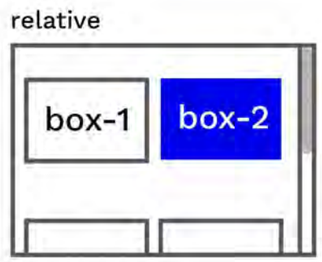


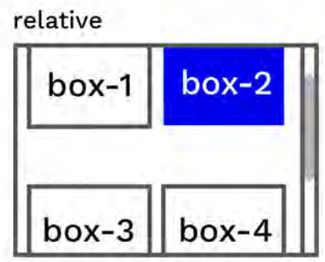
- **FIXED:** posiciona el elemento dentro del *viewport* inicial, fijándose en un sitio independientemente de la posición de los demás elementos.

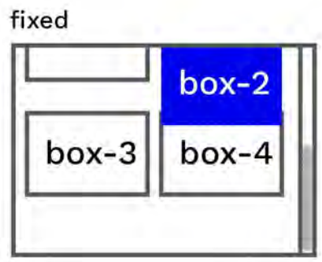




- **STICKY:** intercambia entre *relative* y *fixed*, basándose en la **posición de desplazamiento actual** de un contenedor.







¿Qué es?

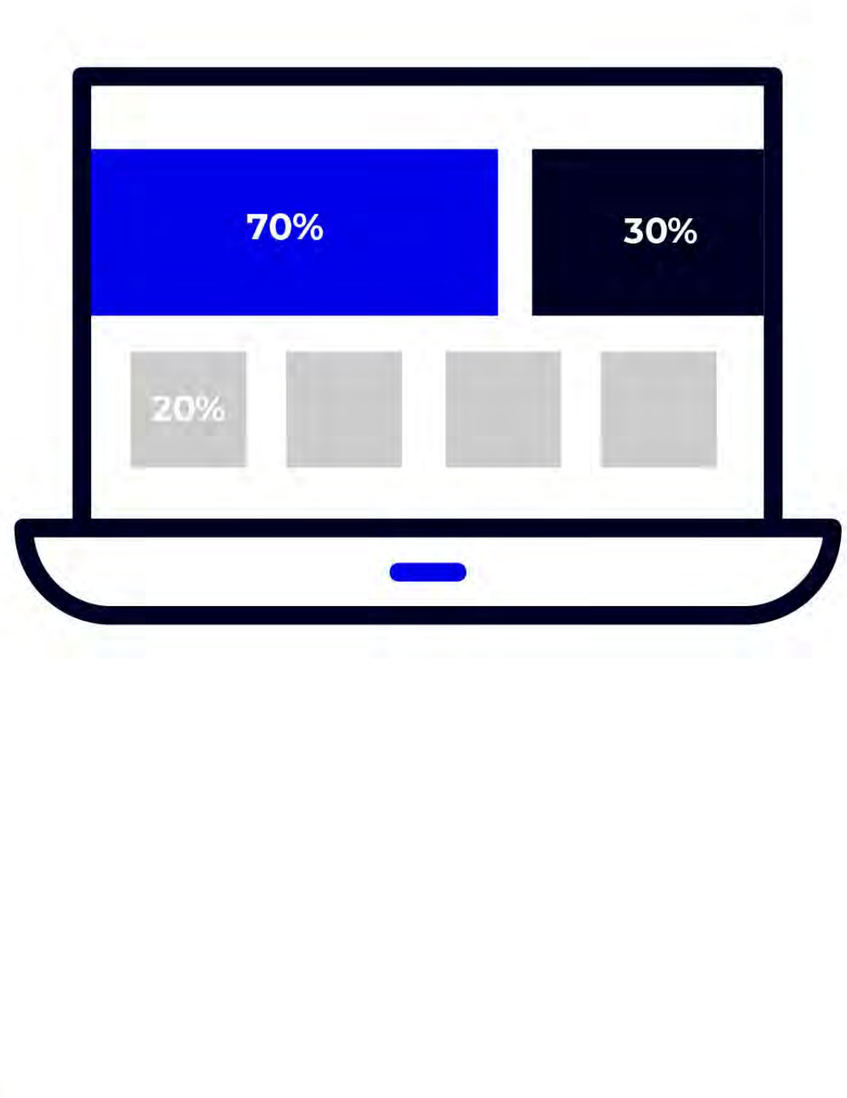
Se basa en la proporcionalidad a la hora de colocar los elementos a lo largo de la interfaz usando **porcentajes** o **em** en vez de píxeles, por lo que independientemente del tamaño de la pantalla, el porcentaje será igual para todos.

¿EN QUÉ CONSISTE?

Con la finalidad de que en distintas pantallas se visualice la información de manera muy parecida, el diseño fluido hace uso de los porcentajes para que tanto en dispositivos móviles como en pantallas grandes, los elementos se muestren de igual forma y siempre se llene el ancho de la página.

Esto puede acabar con una experiencia de usuario bastante desagradable ya que si tenemos la misma disposición de los elementos en todos los dispositivos, lo que se vea bien en una pantalla grande, se puede ver muy pequeño en un móvil.

Por ejemplo, en un monitor muy grande, las imágenes se podrían ver muy estiradas, mientras que en un móvil, la letra pueda llegar a ser demasiado pequeña.

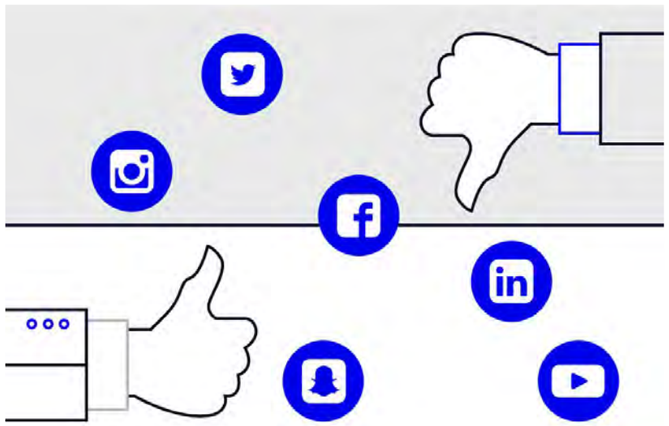


¿Qué es?

Es un **enfoque que se preocupa de desarrollar y diseñar sitios web** que puedan ajustarse a cualquier resolución, adaptando la fuente y las imágenes a cualquier dispositivo. Se intenta **que el usuario tenga una experiencia satisfactoria independientemente del dispositivo** que utilice para acceder.

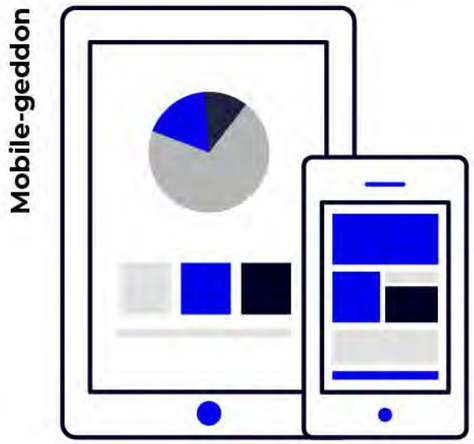
¿POR QUÉ LO NECESITAS?

- **Mejorar la experiencia del usuario**. Según Google, los usuarios de móvil tienen una tasa de rebote del 61% frente al 67% de conversión, si tienen una buena experiencia.



- **El acceso a contenidos desde el móvil está en auge**. El uso de Internet desde el móvil es de más del 75% a nivel global y sigue subiendo gracias a la mejora del ancho de banda y los dispositivos.

- **Google favorece el posicionamiento** de los sitios con Responsive Design en sus búsquedas, ya que aumenta de forma natural el tráfico orgánico. Esta **actualización en el algoritmo de Google** recibió el apodo de **Mobilegeddon** (Mobile + Armagedón). Así que, si no lo haces por los usuarios, hazlo por tu posicionamiento SEO.



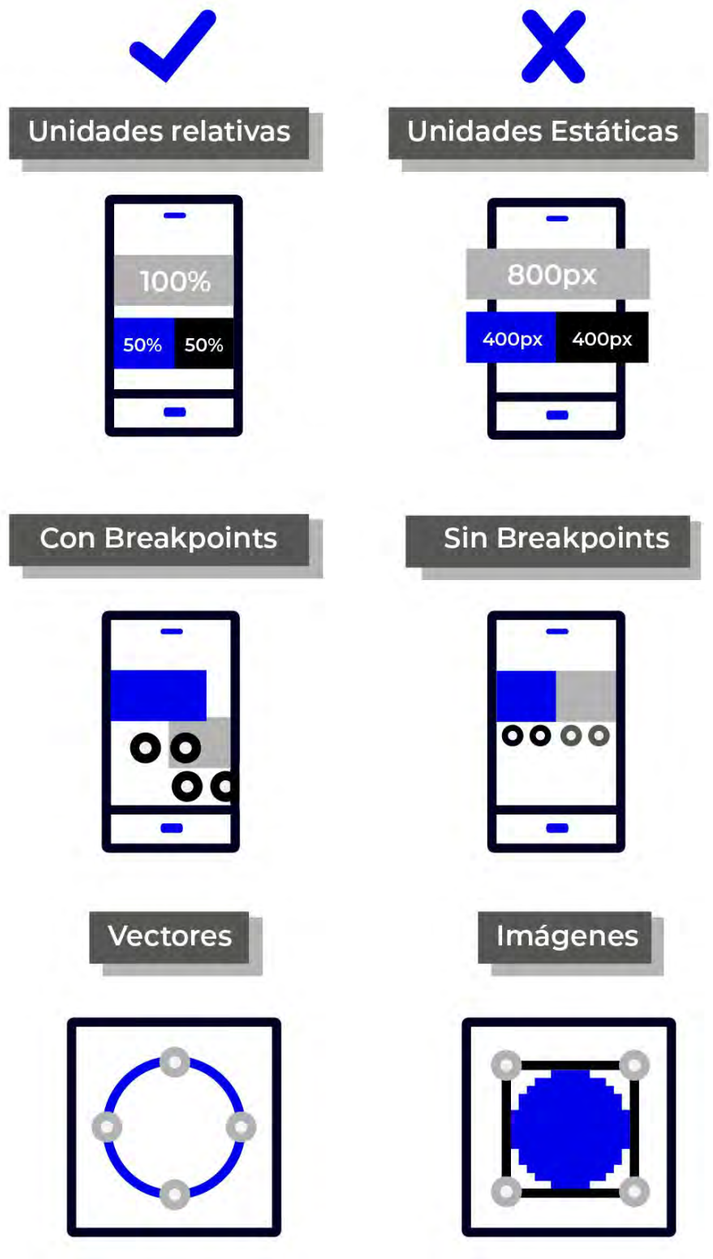
- **Aumenta la velocidad de carga**. Ya que son más ligeros y optimizados para móviles que una versión desktop.
- **Mejora la difusión por RRSS,** aumentando las ventas y la tasa de conversión.
- Responsive Web Design tiene un enfoque adaptativo, lo que nos **ofrece una ventaja competitiva al estar preparados para cualquier dispositivo**.

¿Cómo hacerlo?

Una aplicación o **sitio web *Responsive* debe tener un diseño flexible y capaz de adaptarse** a diferentes resoluciones de pantalla y dispositivos. Estas son **algunas técnicas que nos permiten adaptarnos mejor** a cada dispositivo para así ofrecer una experiencia satisfactoria a los usuarios finales.

TÉCNICAS BÁSICAS

- **Cambiar el tamaño de la caja,** de *box-sizing* a *border-box* para evitar que cada elemento añada sus propiedades de tamaño.
- **Usar la etiqueta meta** name="viewport" content="width=device-width initial-scale=1", **indica a la página el ancho de la pantalla en píxeles independientemente del dispositivo**. También se pueden establecer atributos como minimum-scale, maximum-scale, user-scalable.
- **Hacer uso de CSS Layouts que nos permiten la creación de diseños fáciles y flexibles** como Grid Layout, Flexbox o Multicol.
- **Definir puntos de ruptura**. Son expresiones condicionales que aplican diferentes estilos dependiendo del dispositivo. Esto se hace utilizando una herramienta llamada **Media Queries**.
- **Usar imágenes vectoriales SVG.** La imagen no pierde calidad al redimensionarse.
- **Envolver objetos en un contenedor**. Esto hace un diseño comprensible, limpio y ordenado.
- En el ciclo de desarrollo hay que tener presente que se va a acceder al sitio o la aplicación desde **diferentes dispositivos con distintos tipos de pantalla y resoluciones.**



Definición

La accesibilidad web significa que **personas con algún tipo de discapacidad podrán hacer uso de la Web** gracias a un diseño que va a permitir que estas personas puedan percibir, entender, navegar e interactuar con ella.

TIPOS DE ACCESIBILIDAD

Los tipos de accesibilidad se clasifican en función de la discapacidad que tenga el usuario:

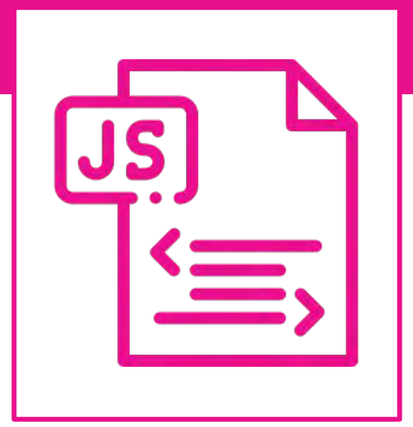
- **Sensorial**: permite el uso de la web a personas con problemas de sordera, ceguera completa/parcial o para distinguir colores como el daltonismo.
- **Motriz**: mejora el uso para gente que no puede utilizar correctamente un ratón, tienen el control motor delicado o tiempo de respuesta lento.
- **Cognitiva**: reúne una serie de técnicas para permitir que usuarios con un lenguaje de compresión y entendimiento limitado utilicen la web.
- **Tecnológica**: permite el uso de la web a usuarios que no disponen de los recursos suficientes para acceder a la web de manera eficiente, como por ejemplo, conexión lenta a la web o acceso a través de móvil y tablets.

TÉCNICAS

Existen una serie de técnicas para conseguir que nuestras web sean accesibles y pautas que podemos seguir:

- **Fundamentales**
  - Elementos sonoros o gráficos con información textual alternativa.
  - Diseño de la web independendiente del dispositivo.
  - Desactivar elementos visuales o sonoros para no interferir en la lectura.
  - Emplear un lenguaje sencillo.
  - Buena usabilidad de la Web.
  - Hardware y software actualizados.
- **CSS**
  - Diseño de páginas flexibles al tamaño de la interfaz, tamaño de fuente...
  - Color adecuado y alto contraste.
  - Tamaño de fuente grande y/o flexible.
  - Elementos de interacción fáciles de clicar.
- **HTML**
  - Descripciones detalladas para imágenes complejas.
  - Información alternativa para los marcos.
  - Tablas bien formadas (para su lectura secuencial).

Front:
JavaScript



¿Qué es?

Es un **lenguaje de scripting basado en ECMAScript** (ActionScript y JScript son otros lenguajes que implementan ECMAScript) que puede ejecutarse tanto en el lado del cliente como en el del servidor y permite crear contenido dinámico en una web.

¿PARA QUE SE USA?

Hoy en día, la mayoría de los navegadores vienen con motores para el renderizado de JavaScript, esto significa que se podrán ejecutar comandos en el documento HTML sin la necesidad de descargar un programa o compilador externo. Algunos usos muy comunes de JavaScript en el lado del cliente (UI) son la automatización de procesos para que el usuario no tenga que realizarlos de forma manual, como por ejemplo:

- Sugerencias o autocompletado de texto.
- Animaciones.
- Paso de imágenes en un carrusel de forma automática.
- Formularios interactivos.

A través de un script, podemos realizar este tipo de funcionalidades y muchas otras, permitiendo mejorar la experiencia del usuario en una web.

¿CÓMO AÑADIRLO?

Se puede añadir importando un fichero con extensión **.js** que contenga el código. También se puede añadir directamente en el HTML usando la etiqueta <script>código JS</script>, aunque este método está desaconsejado.

VENTAJAS

- **Un único lenguaje** para desarrollar front y back.
- Es un lenguaje **nativo en los navegadores web**.
- Al ser un lenguaje no tipado (si no integramos Typescript), su aprendizaje es muy **sencillo**.
- Dispone de **distintas librerías o frameworks** que facilitan el desarrollo de SPAs (Single Page Applications) en caso de no querer usar JavaScript nativo (VanillaJS).

DESVENTAJAS

- Dependiendo del navegador, se puede ejecutar de una forma u otra ofreciendo una experiencia de usuario distinta.
- Puede ser usado con fines **maliciosos** debido a que el código se ejecuta en el ordenador del usuario.
- Hay usuarios que desactivan JavaScript cuando navegan (por lo comentado en el punto anterior) afectando a la usabilidad de la web.

¿Qué es?

ECMAScript (ES), **es una especificación de lenguaje de scripting definido por Ecma International.** Su implementación más utilizada es la de JavaScript, de modo que ES se ha convertido en el estándar encargándose de regir como JavaScript debe ser interpretado y funcionar.

CAMBIOS RELEVANTES (I)

Aunque tiene versiones anteriores, es la versión nacida en 2015 la más relevante de todas y la que trajo los grandes cambios en JavaScript. Esta especificación es conocida como **EcmaScript 6 (ES6).**

Algunas de las **novedades** más relevantes introducidas por ES son:

- **Función Arrow:** conocidas como expresiones lambda en lenguajes como Java y C#, son abreviaciones de funciones utilizando el operador "=>"

```
let sum = (x, y) => x + y;
console.log(sum(1, 2)); // Imprime 3
```

- Operadores **spread, rest y destructuring**, que nos permiten operar con arrays/propiedades de objetos para declarar variables, pasarlas como parámetros:

```
let args = [0, 1, 2];
function f(x,y,z) {}
f(4, ...args); //x=5; y=0; z=1 => Spread
function f2(x, ...args) {console.log(args);}
f2(1, 2, 3); //Imprime [2,3] => Rest
let [x, y, z] = ...args; //x=0; y=1; z=2 => Destructuring
```

- Se introducen las variables de tipo **let** y **const**, recomendando su uso en detrimento de var. Las variables let sólo se encuentran definidas en el bloque en el que se declara, mientras que var permite utilizar la variable fuera del bloque, dando lugar a errores.

```
let x = 1; // Usar en vez de var
const y = 1; // No puede cambiar de valor
```

Sistemas de tipos

autentia

¿Qué son?

Definen un conjunto de reglas asignadas a una propiedad, clase, función y tienen como objetivo reducir los errores en un proceso de desarrollo. Esto puede ocurrir de forma estática (en tiempo de compilación) o de forma dinámica (en tiempo de ejecución).

NO TIPADO

Son aquellos lenguajes que no tienen definido un sistema de tipos, por lo que bastará con que nuestro código no tenga errores sintácticos para que compile correctamente. Este tipo de lenguajes no tienen ninguna de las ventajas de los tipados y sí todas sus desventajas.

TIPADO ESTÁTICO

Son aquellos lenguajes que definen los tipos **en tiempo de compilación** y en caso de equivocarnos, el compilador nos mostrará un error. Algunos ejemplos son Java, C, Go, C# y Typescript.

Algunas ventajas:

- Detección temprana de errores.
- Código más expresivo.
- Ayuda durante el desarrollo con el autocompletado.

Algunas desventajas:

- Vuelve más lento el proceso de desarrollo ya que hay que compilar el código.
- Mayor dificultad para alguien que se inicia en el mundo de la programación.

TIPADO DINÁMICO

Son aquellos lenguajes que definen los tipos **en tiempo de ejecución**. Podemos definir el tipo mal y no nos daremos cuenta del error hasta que estemos ejecutando la aplicación, ya que no tenemos un compilador que nos avise de este fallo. Algunos ejemplos son Javascript, Python, Ruby y PHP.

Algunas ventajas:

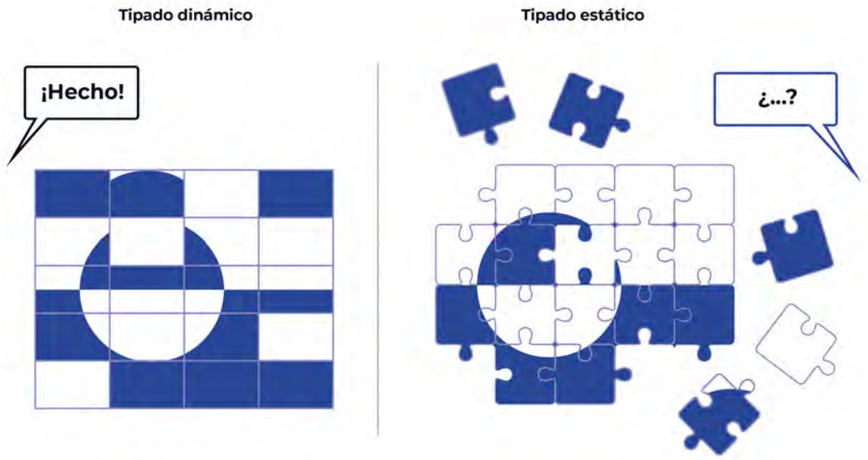
- El proceso de desarrollo es más rápido al no tener que compilar.
- Código más ‘flexible’.
- Aunque no haya compilador, hay herramientas que ayudan a prevenir errores (linters).

Algunas desventajas:

- Más propenso a errores humanos.
- Código menos expresivo (se debe inferir de qué tipo es cada variable, función, etc.)

TIPADO FUERTE O DÉBIL

Suelen definirse como aquellos lenguajes que tienen más restricciones (o menos) en su tipado. Por ejemplo, Javascript y Elixir son tipados dinámicos pero Javascript permite sumar 1 (como entero) + ‘1’ (como string) y da como resultado ‘11’ (hace una concatenación). En Elixir recibiremos un error en tiempo de ejecución.

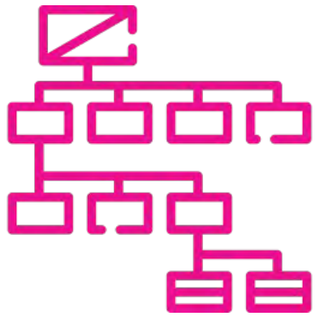


Un Resumen

Los tipos en JavaScript más comunes se resumen a continuación. Cualquier variable puede ser de alguno de estos tipos en cualquier momento.

TIPOS

- **String**: representa una cadena de caracteres. Se puede definir con comillas dobles o simples.
- **Number**: representa los números, tanto los enteros como los de punto flotante. Tiene una capacidad de 2^{53} - 1.
- **BigInt**: cuando un número es demasiado grande para ser representado como un Number, se utiliza BigInt. Se define con la letra 'n' al final del número.
- **Boolean**: este tipo se representa con las palabras *true* o *false*.
- **Undefined**: se utiliza para representar una variable que ha sido declarada pero no ha sido inicializada hasta el momento.
- **Null**: un tipo que representa un valor inexistente.
- **Object**: un tipo complejo que agrupa un conjunto de valores de distintos tipos para representar un concepto más abstracto. Cada valor de un Object tiene un nombre único asociado a él. Este nombre se conoce como *propiedad*. Las llaves '{' '}' definen a un Object cuyos elementos siempre se encuentran entre estos dos símbolos.
- **Array**: define una serie de valores que pueden ser de distintos tipos. Se diferencia del Object, principalmente, porque cada elemento del Array no tiene un identificador personalizado asociado, solo la posición en la que se encuentra dentro de ella. Los corchetes '[' ']' definen a un Array cuyos elementos siempre se encuentran entre estos dos símbolos.
- **Function**: simboliza un método, incluyendo su firma y sus instrucciones. Una variable definida como un Function se puede utilizar luego para invocar el mismo método varias veces o incluso, para incluirla como un parámetro de otra función. Se define como cualquier función de JavaScript.



¿Qué son?

Estructuras de datos complejas capaz de almacenar grandes cantidades de información y recuperar elementos específicos de forma eficiente.

MAP

Es un objeto que guarda parejas de **clave-valor** donde la clave y el valor pueden ser de cualquier tipo (string, number, boolean, incluso un Object). Los métodos más usados son:

- map.set(key, value)
- map.get(key)
- map.has(key)
- map.delete(key)
- map.clear()
- map.keys()/values()

SET

Es una colección de **valores únicos.** Aunque se intente añadir a través del método *add* el mismo valor, éste no hará nada y esta es la razón por la que los valores en un Set solo aparecen una sola vez. Los métodos más usados son:

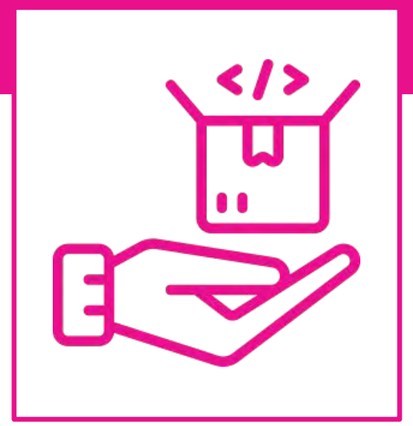
- set.add(value)
- set.delete(value)
- set.has(value)
- set.clear()
- set.values()

WEAKMAP

Igual que un Map pero con la diferencia de que **las claves solo pueden ser objetos y no primitivos** y no soporta métodos como *keys(), values()* o *size()*. Además, las referencias a los objetos de las claves son débiles, por lo que si no hay ninguna referencia a ese objeto, éste será eliminado de memoria y del propio mapa automáticamente por el Garbage Collector.

WEAKSET

Igual que un Set, pero **únicamente puede almacenar objetos y no primitivos.** Las referencias a los objetos son débiles (igual que un WeakMap) por lo que estos serán eliminados una vez sean inaccesibles. Tampoco soporta métodos que tengan que ver con los valores o el tamaño de la colección como *values(),* o *size()*.



¿Qué son?

Una promesa es un **objeto de JavaScript que puede producir un valor en el futuro**. A las promesas se le añaden funciones que se ejecutan cuando tiene éxito y también cuando fallan, pudiendo gestionar errores fácilmente.

DESCRIPCIÓN

Las promesas son unos objetos de JavaScript que nos ayudan a trabajar con código asíncrono. El valor del resultado de una promesa no se conoce cuando es creada, si no que la **promesa tiene una operación asíncrona que se tiene que resolver**.

Puede tener estos **tres estados:**

- **Pending**: estado inicial, cuando se crea el objeto.
- **Fulfilled**: resuelto con éxito.
- **Rejected**: resuelto pero ha fallado.

Un ejemplo, sería envolver una petición a una API Rest en una promesa. Si la petición tiene éxito queremos mostrar el resultado en la página y si ha fallado queremos mostrar un modal al usuario diciéndole que algo ha ido mal.

Usar promesas tiene muchos beneficios:

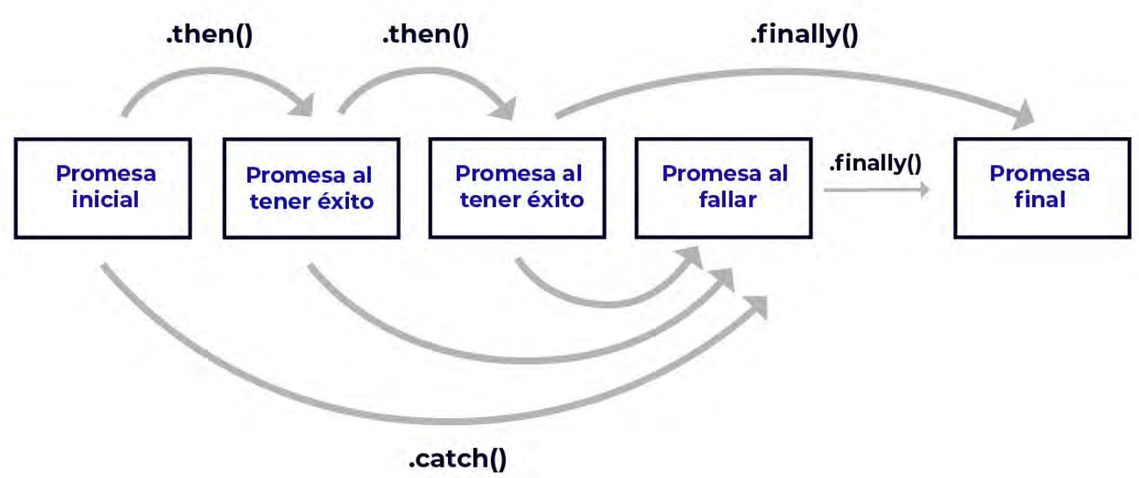
- Mejora la **claridad del código**. Al tener una sintaxis más parecida al código síncrono, es fácil de entender.
- Dan más control sobre la **gestión de los errores.**
- Definen una **estructura común** para trabajar con operaciones asíncronas.

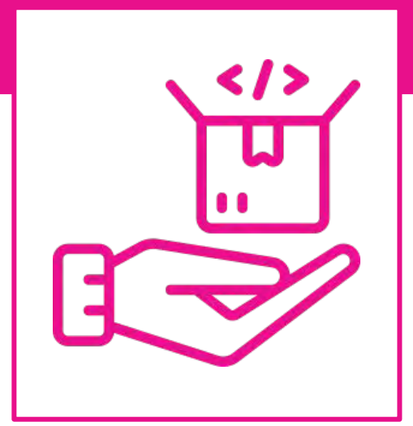
MÉTODOS

Los métodos promise.then(), promise.catch() y promise.finally() se usan para asociar acciones con el resultado de resolver la promesa. Estos métodos pueden devolver otra promesa, por lo que **pueden encadenarse distintas promesas.**

- then(): se ejecuta cuando la promesa se resuelve y ha tenido éxito (fulfilled).
- catch(): se invoca cuando ha fallado (rejected).
- finally(): agrega un método que se ejecuta cuando se resuelve, tanto si ha tenido éxito como si no.

Async/await es una alternativa más moderna a then/catch para gestionar la asincronía con promesas, proporcionándonos una forma más sencilla y limpia de trabajar.





Promesas - then/catch vs. async/await

autentia

¿Qué son?

Then/catch y async/await son dos formas distintas gestionar promesas, siendo la segunda la más moderna. Es recomendable usar siempre async/await frente a la otra forma, ya que facilita la lectura y evita problemas como el *callback hell*.

THEN / CATCH

Then/catch fue la primera forma que había de gestionar las promesas. Gracias a la introducción de las promesas, se solucionó el problema del *callback hell* (anidación). De esta forma, es posible encadenar promesas:

```
readDir("/folderpath")
  .then((files) => getFilesSize(files))
  .then((filesSize) => // Return another promise)
  .then((result) => // Return another promise)
```

Aún así, **todavía es posible que aparezca un *callback hell*** (si, por ejemplo, necesitamos el resultado que se obtuvo hace más de una promesa), como aquí:

```
connectToDatabase().then(db => {
  return getUser(db).then(user => {
    return getUserSettings(db).then(settings => {
      return enableAccess(user, settings);
    });
  });
});
```

ASYNC / AWAIT

El async/await **es la siguiente iteración que se introdujo en las promesas**. Básicamente, es un azúcar sintáctico que consigue hacer que el **código asíncrono se asemeje a un código procedural o síncrono**. Gracias a esto, también se soluciona del todo el problema del *callback hell*. Si volvemos a escribir el ejemplo previo con async/await, quedaría de la siguiente manera:

```
const db = await connectToDatabase();
const user = await getUser(db);
const settings = await getUserSettings(db);
await enableAccess(user, settings);
```

Hay que tener en cuenta que al utilizar algún *await*, habrá que utilizar la palabra *async* en la función que contenga ese código:

```
async function example() {
   const db = await connectToDatabase();
   return await getUser(db);
}
```

ES Modules

autentia

¿Qué es?

A partir de la especificación ES6, se estandarizó la importación de módulos. El objetivo es poder **importar una serie de funciones, variables, objetos de forma nativa en el navegador**, sin depender de herramientas de terceros para realizar este trabajo.

CARACTERÍSTICAS

Existen dos tipos de exportaciones:

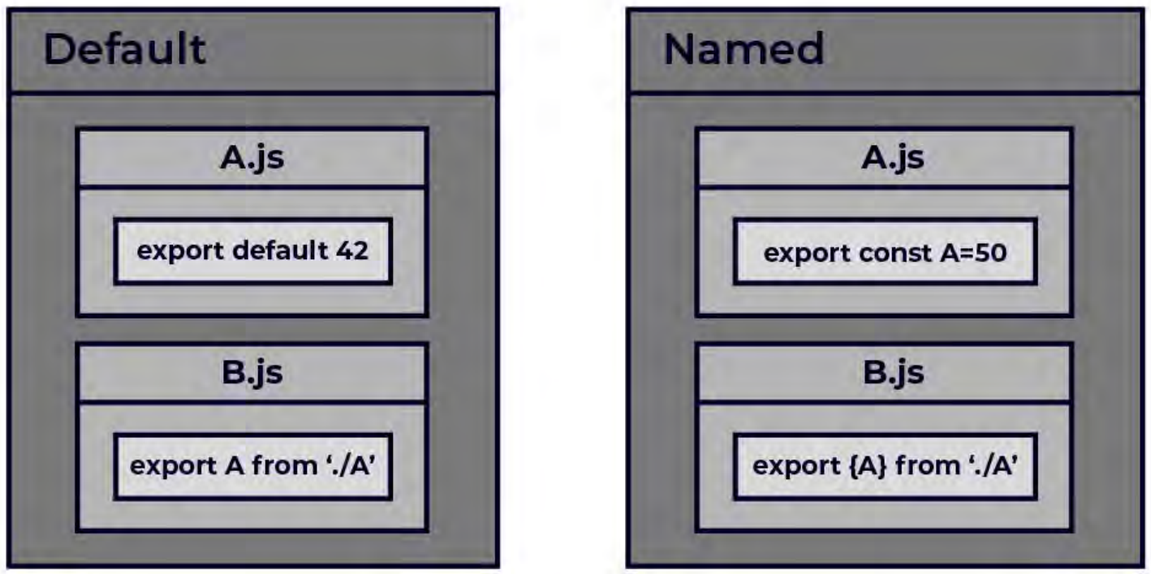
- **Named exports**: tienen un nombre ya asignado y es obligatorio importarlas con el mismo nombre con el que se exportaron.
- **Default exports:** se puede exportar sin tener que especificar un nombre y se podrá importar con el nombre que se desee. Importante saber que solo puede haber un *default export* por módulo.

Podemos combinar el uso de *named* y *exports* aunque no se recomienda usar *default* ya que damos la libertad al desarrollador a importar el módulo con distintos nombres en distintos ficheros y a la hora de hacer un refactor en el futuro, puede dar muchos dolores de cabeza.

Si queremos importar todos los named exports de una vez como un objeto, se puede usar *import * as AnyName from 'module_name'*

VENTAJAS

- ESM se ejecutan siempre en ***strict mode***.
- **Mejora la organización del código encapsulando funcionalidades** de un fichero e importándolas en otro.
- Son importaciones **nativas en el navegador** por lo que no hay necesidad de usar dependencias externas.



Front:
JavaScript
(Entorno)

¿Qué es?

Node.js es un entorno de ejecución multiplataforma basado en JavaScript, es de código abierto y principalmente se usa para servidores web.

CONCEPTOS BÁSICOS

Gracias a Node.js se puede **utilizar JavaScript fuera del navegador**, pudiendo usarse en cualquier plataforma como una aplicación más. Esto le da a JavaScript la capacidad de hacer las mismas cosas que otros lenguajes de scripting como Python.

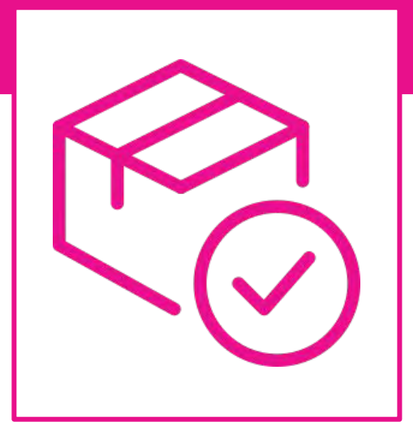
Uno de los usos más comunes de Node.js es el desarrollo de servidores web. En un servidor web tradicional se tendría un hilo por cada usuario. Con Node.js solo se tiene un hilo, pero su diseño hace que las tareas de I/O no bloqueen el hilo y pueda continuar con unas peticiones mientras espera a otras.

VENTAJAS

- Node funciona en un solo hilo. Usa un bucle de eventos para procesar las llamadas no bloqueantes de I/O de forma concurrente en un solo hilo. Esto tiene la ventaja de tener menos coste de memoria que si usara varios hilos.
- Para interpretar JavaScript utiliza el motor V8, creado para Chrome, que está muy optimizado.
- Los desarrolladores pueden crear paquetes y subirlos a un repositorio (llamado npm) para distribuirlos.

LIMITACIONES

- Cuando nos encontramos con tareas intensivas en CPU, Node.js tiene el módulo de Worker Threads para crear nuevos hilos. Cada hilo tiene su propia instancia de Node y del motor de JavaScript (para evitar problemas de concurrencia), por lo que tiene un impacto en la memoria.
- Calidad irregular en los módulos de npm. Existen paquetes muy estables y también otros que están poco probados y no tienen mucha documentación.



¿Qué es?

Npm es el **gestor de paquetes** por defecto de Node.js. Nos permite instalar dependencias, administrar módulos y gestionar paquetes de una manera sencilla. Npm también es la mayor **librería de software** del mundo.

¿CÓMO USAR NPM?

Las dependencias se gestionan desde el archivo **package.json**, ubicado en la raíz del proyecto. El fichero tiene la siguiente estructura:

```
{
  "name" : "NOMBRE_DEL_PROYECTO",
  "version" : "1.2.3",

  "scripts": {
    "start": "ng serve",
    "test": "ng test",
  },

  "dependencies": {
    "core-js": "3.6.4",
  },

  "devDependencies": {
    "prettier": "2.0.4",
    "stylelint": "13.0.0",
  }
}
```

ESTRUCTURA DE PACKAGE.JSON

Package.json tiene 3 secciones importantes:

- **Scripts:** aquí se especifican los comandos que se podrán ejecutar desde la línea de comandos. Para ejecutar un comando nos basta con hacer `npm run NOMBRE_DEL_COMANDO.`
- **Dependencies:** en esta sección se especifican las librerías y paquetes necesarios para que la aplicación funcione.
- **devDependencies:** aquí se especificarán las dependencias que se van a necesitar durante el desarrollo pero que no se necesitan para que la aplicación funcione como tal. Por ejemplo, las librerías que se utilicen para el testing.

¿Qué es?

Babel es una herramienta usada principalmente para **convertir código escrito en ECMAScript 2015 o superior en versiones retrocompatibles de Javascript.** De este modo, puede funcionar en entornos de navegadores actuales o más antiguos.

CARACTERÍSTICAS

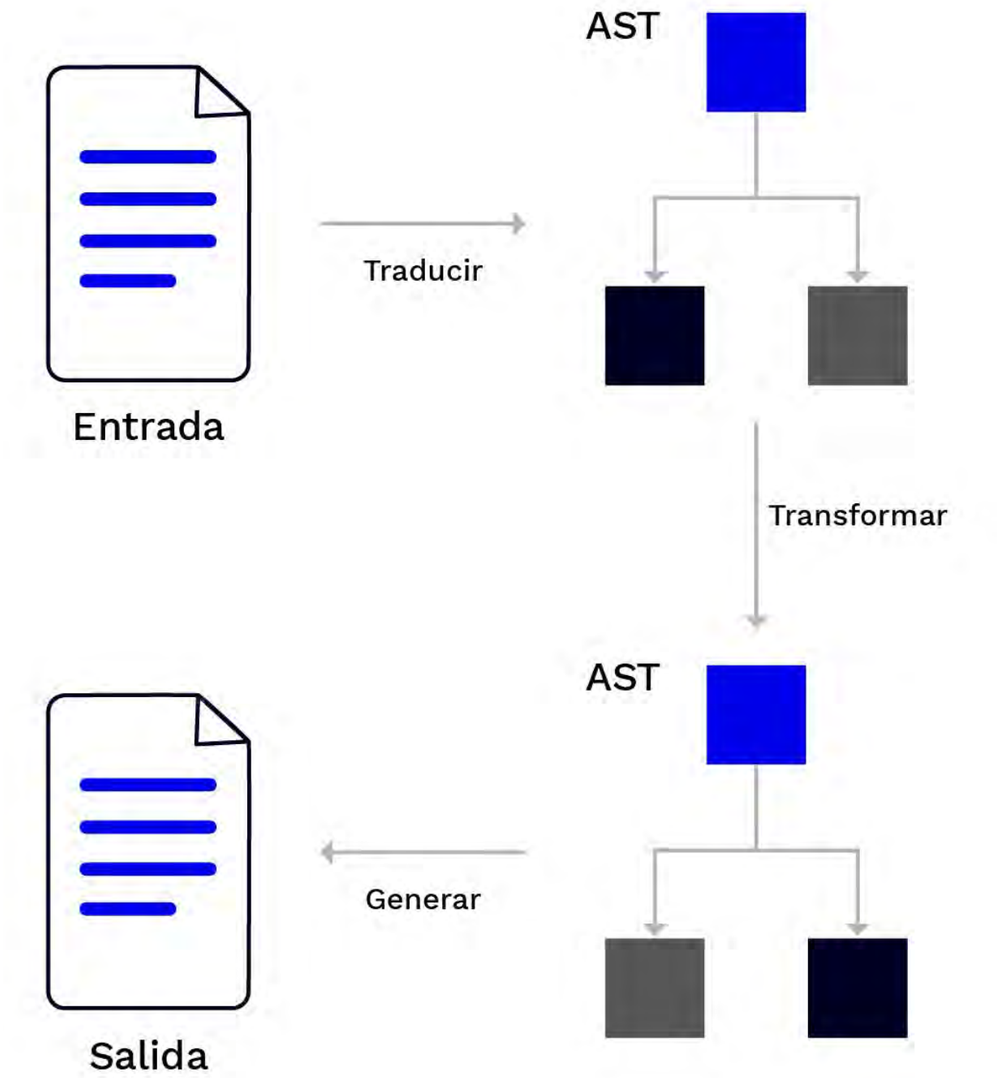
Babel coge un código de entrada y le hace una serie de transformaciones para entregar otro código como salida. Por defecto no hace ninguna transformación, **son los plugins los que hacen las transformaciones.** Entre las cosas que se pueden hacer con Babel están:

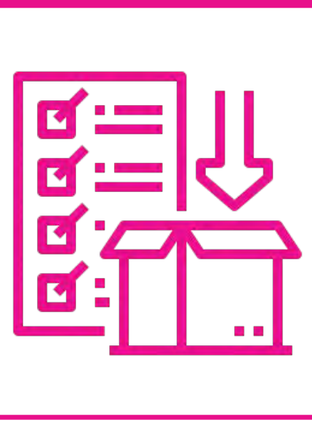
- Convertir la sintaxis entre versiones.
- Introducir Polyfills para funcionalidades más modernas.
- Automatizar refactorizaciones de código.

¿Cómo consigue hacer todas estas transformaciones? Babel transforma el código Javascript **en una representación de *Abstract Syntax Tree* (AST)** y luego, según los plugins configurados, le hace las transformaciones correspondientes.

Para hacer más fácil su uso, hay disponibles una serie de **presets** para usos comunes, como @babel/preset-typescript, que incluye todos los plugins necesarios para hacer una transpilación de Typescript a Javascript.

Además, los presets pueden tener en cuenta los navegadores y las versiones que tenemos como objetivo, de forma que si hay alguna transformación que ya es soportada por el navegador, no sería necesaria hacerla.





¿Qué es?

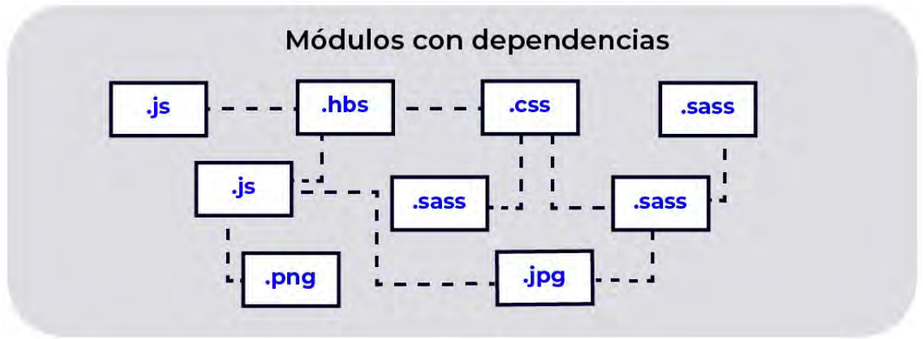
Herramienta Open Source cuya finalidad es la de empaquetar y optimizar los ficheros de un proyecto en uno o más paquetes o ficheros (normalmente uno). A las herramientas que realizan esta tarea se les conoce como **bundlers.**

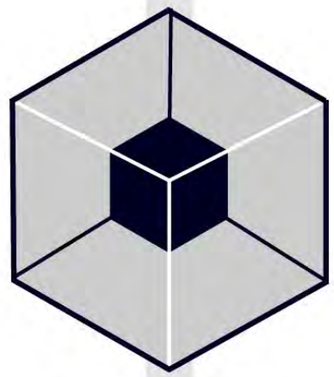
CARACTERÍSTICAS

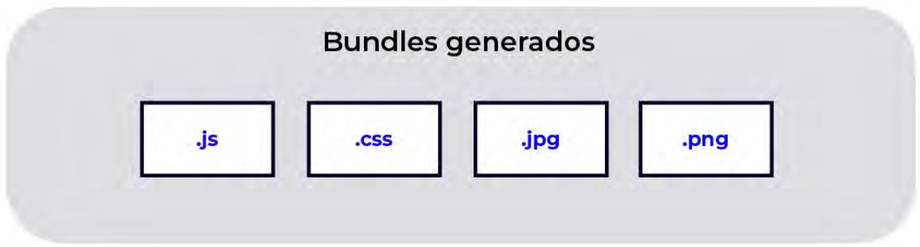
Webpack realiza muy bien su función de minimizado, ya que unifica todas las dependencias en una sola. Genera un archivo para el código JavaScript, otro para el CSS, etc. El resultado sería uno o varios ficheros listos para poner en producción y en los que todas las dependencias quedarían resueltas. Webpack también tiene en cuenta archivos como imágenes, tipos de letra (fonts), etc. y los convierte en dependencias de la aplicación.

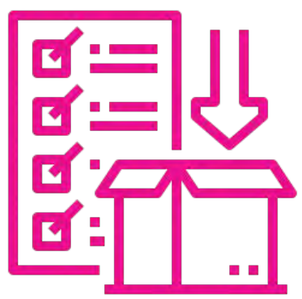
La configuración se suele hacer a través del archivo **webpack.config.js** que estará en la raíz del proyecto. Las propiedades más importantes de configuración son las siguientes:

- **Punto de entrada (entry):** punto desde donde webpack comenzará a analizar el código.
- **Punto de salida (output):** punto donde colocará los paquetes generados. Normalmente en el directorio **dist**.
- **Loaders:** por defecto, webpack solo convierte archivos .js o .json. Con los loaders podemos extender estas funcionalidades y convertir otro tipo de archivos. Un ejemplo, a través de un loader podemos convertir un archivo Typescript a JavaScript.
- **Plugins:** permiten extender funcionalidades que no se pueden hacer con los loaders como optimizar el código empaquetado, variables de entorno, etc.









¿Qué es?

Parcel es un '*bundler*' o empaquetador de aplicaciones. Ofrece un rendimiento rápido utilizando procesamiento multinúcleo, además de no requerir configuración.

CONFIGURACIÓN BÁSICA

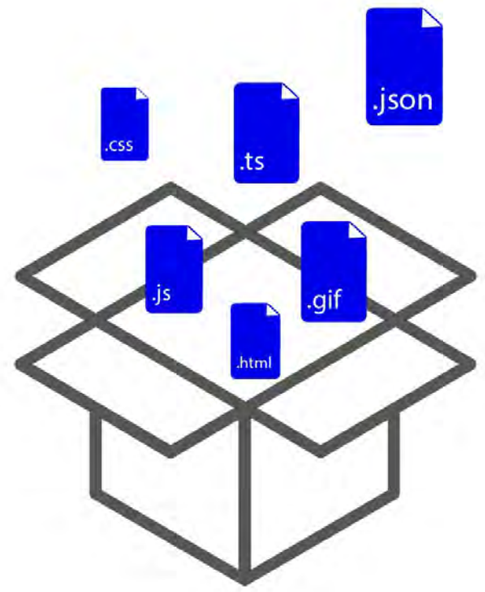
Parcel permite **empaquetar ficheros** Javascript, HTML, CSS, entre otros, **de una forma muy rápida y con cero configuración.**

Para instalar Parcel, tan solo debemos ejecutar `npm install --save-dev parcel-bundler`. Seguidamente indicamos qué fichero será el punto de entrada del proyecto y con el flag *-p* indicamos un puerto específico para el servidor que nos ofrece Parcel en caso de que queramos visualizar lo que se ha generado: `parcel -p 1234 index.html.` También podemos usar la propiedad *watch* en el anterior comando que permite live/hot reloading. Esto nos ayudará a ver los cambios instantáneamente sin tener que reiniciar el servidor para ello.

VENTAJAS

- **Tiempos de empaquetado muy bajos:** compilación multinúcleo y caché del sistema de archivos para reconstrucciones rápidas, incluso después de un reinicio.
- **Separación de código (Code splitting)**: a través de imports dinámicos.
- **Reemplazo de módulos en caliente (hot reloading).**
- **Conversiones automáticas:** a través de Babel, PostCSS, entre otros.
- **Resaltado de errores amigable.**

Bundler	Time
browserify	22.98s
webpack	20.71s
parcel	**9.98s**
parcel - with cache	**2.64s**



Snowpack

autentia

¿Qué es?

Herramienta para aplicaciones web modernas que **permite ejecutar tu aplicación sin empaquetar en desarrollo.** **Permite instalar dependencias actuales optimizadas** de tal forma que puedan correr nativamente en el navegador. Snowpack **no es un bundler, sino una herramienta que puede sustituir a un bundler** como Webpack o Parcel.

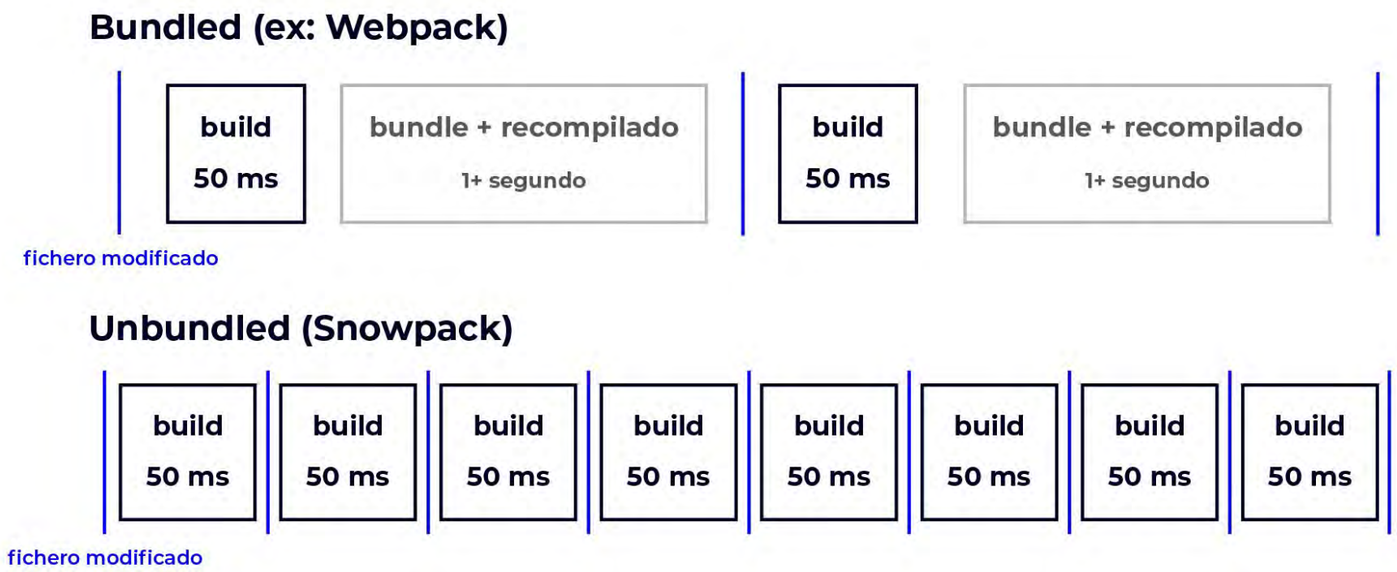
¿EN QUÉ CONSISTE?

Los bundlers normalmente, por muy eficientes que sean, pueden tardar unos segundos en volver a empaquetar nuestro código con los cambios realizados. Hoy en día, gracias a que los ES Modules se ejecutan de forma nativa en el navegador, no necesitamos un bundler para tal fin. Aun así, existen dependencias legacy que el navegador no puede entender y aquí es donde entra Snowpack, instalando esas dependencias modernas.

Al hacer el despliegue de la página o aplicación web, Snowpack es compatible con bundlers como Webpack o Parcel, lo que permite optimizar el código de producción. Además, durante el desarrollo, Snowpack servirá la aplicación sin empaquetar y sin depender del bundler usado en producción, por lo que los ciclos de desarrollo son más rápidos.

CARACTERÍSTICAS

- **Build time reducido:** gracias al uso de ES Modules tendremos un renderizado casi instantáneo en el navegador.
- **Cada archivo se construye individualmente y se almacena en caché indefinidamente**. El entorno de desarrollo nunca creará un archivo más de una vez, ni el navegador descargará un archivo dos veces, a menos que éste cambie.
- **Servidor de desarrollo:** sólo construirá un archivo cuando el navegador lo solicite. Por defecto soporta archivos .ts y .jsx, compilándolos a .js antes de enviarlos al navegador.



¿Qué es?

Sass es un preprocesador que extiende del CSS y le aporta funcionalidades extra, que luego traduce a CSS puro para que el navegador lo pueda entender.

FUNCIONALIDADES

- Variables (Sass ofrecía variables antes de ser estandarizados en CSS).
- Anidación de elementos.
- Notaciones más cortas para selectores y propiedades.
- Herencia (extiende los estilos de otro selector).
- Mixins (una especie de funciones que generan CSS en función de una variable introducida).

```
@mixin button($button-color, $text-color) {
    background-color: var($button-color);
    color: var($text-color);
    padding: varr(--s) var(--m);
    border-radius: 7px;
}

.blue-button {
    @include button(blue, white);
}
```

VENTAJAS

- Sass te permite escribir menos CSS, de forma limpia y sencilla.
- Por lo general, tiene menos código, por lo que tardas menos en escribir el CSS.
- Es más potente que el CSS puro, ya que es una extensión de este. Ofrece funcionalidades muy útiles.
- Es compatible con todas las versiones de CSS, por lo que se puede usar cualquier librería de CSS.

DESVENTAJAS

- Requiere tiempo aprender las funcionalidades extra que aporta Sass con respecto a CSS.
- El código ha de compilarse.
- La depuración se vuelve más compleja.
- Usar Sass puede dificultar el uso del inspector de elementos del navegador.

¿Qué es?

PostCSS es una herramienta para transformar los estilos de CSS con plugins de JavaScript. Estos plugins pueden lintear el CSS, soportar variables y mixins, transpilar sintaxis CSS futura, etc.

PLUGINS

PostCSS tiene más de 200 plugins, cada uno para una función específica. Estos son algunos de los ejemplos que muestran las posibilidades que ofrece PostCSS:

- **Autoprefixer**: este plugin añadirá prefijos de proveedor (vendor-prefixed properties) a las propiedades de CSS para que sean compatibles con distintos navegadores. De esta forma, reducimos la confusión en nuestro código.
- **PostCSS Preset Env**: con este plugin se puede utilizar sintaxis de CSS futuro (que todavía no haya salido) y el mismo plugin lo traducirá a CSS que los navegadores puedan entender. Tiene el mismo objetivo que Babel con JavaScript pero aplicado a CSS.
- **Stylelint**: este plugin nos indica errores que haya en nuestro código y nos fuerza a seguir un estándar de buenas prácticas a la hora de escribir código CSS. Además, soporta la última versión de sintaxis de CSS. A veces, no necesitamos que Stylelint nos indique todos los errores en el código, es por esto que nos permite habilitar o deshabilitar las reglas que más nos convengan para que se adapte a nuestro desarrollo.

¿EN QUÉ SE DIFERENCIA CON SASS?

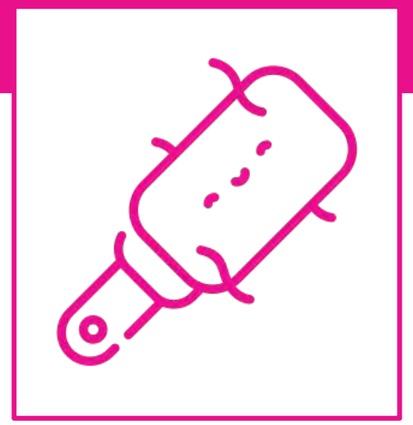
La principal diferencia entre PostCSS y preprocesadores como Sass, Less y Stylus es que PostCSS es modular y llega a ser incluso más rápido que el resto.

Con PostCSS puedes escoger qué funcionalidades utilizar, mientras que los demás preprocesadores incluyen un montón de funcionalidades que puedes necesitar o no.

AUTOPREFIXER

```
:fullscreen {
}
```

```
:-webkit-full-screen {
}
:-ms-fullscreen {
}
:fullscreen {
}
```



¿Qué es?

Se encarga de analizar el código en busca de errores programáticos y estilísticos, como por ejemplo, errores de sintaxis o nombres de variables mal escritos.

LINT VS. FORMAT

Format

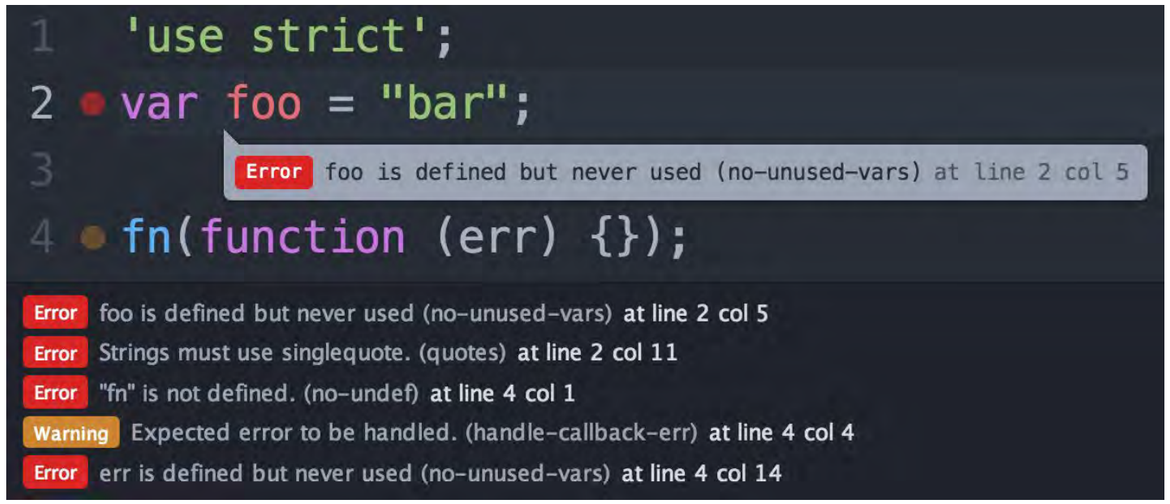
- Cubre la necesidad de que todo el mundo en un mismo proyecto utilice el mismo formato de código.
- Revisa y modifica los espacios, tabulaciones, etc.

Lint

- Cubre la necesidad de que todo el mundo en un mismo proyecto utilice las mejores prácticas para un código de buena calidad (por ejemplo, usar *let* y *const* en JavaScript en vez de *var*).
- Ayuda a utilizar las mejores sintaxis o nuevas funcionalidades de un lenguaje y atrapar posibles errores.

LINTERS MÁS UTILIZADOS PARA JS

- ESLint
- JSLint
- JSHint



Fuente: https://developer.com/linting-is-parenting-878b2470836a

¿En qué consiste?

Cada navegador implementa los estándares de manera distinta. Esto puede dar lugar a que los usuarios tengan una experiencia diferente en función del navegador que están usando.

CONCEPTO

La compatibilidad entre navegadores es un concepto a tener en cuenta a la hora de desarrollar aplicaciones web, debido a que para una misma web, **usar distintos navegadores puede resultar en una experiencia distinta**. Puede darse el caso extremo en el que exista incompatibilidad con un navegador, quedando limitada la audiencia de esa web.

Cuando el diseño se desajusta entre navegadores, puede ocurrir que el texto no quepa en la pantalla, que no sea visible la barra de scroll o que cierto código en JavaScript no se ejecute, etc. Como es inviable comprobar la compatibilidad entre todos los navegadores del mercado, merece la pena asegurarlo entre los que tienen más cuota de mercado, como Chrome, Firefox y Safari.

Hay distintas acciones que nos ayudan a asegurar esta compatibilidad, entre las que se encuentran:

- **Validar tanto el HTML como el CSS** de la web para que cumplan el estándar.
- **Resetear los estilos CSS**. Cada navegador tiene unos valores por defecto para ciertas propiedades, haciendo que algunos elementos se vean distintos.
- Usar **técnicas soportadas**. La web de Can I Use muestra la compatibilidad de funciones de la API de JavaScript para distintos navegadores.

DIFERENCIAS ENTRE NAVEGADORES

Hay dos piezas fundamentales en un navegador: por un lado el motor de renderizado (que analiza el código HTML y CSS) y por otro el motor de JavaScript.

Cada **navegador utiliza un motor distinto** que implementa los estándares con pequeñas diferencias. Además, esta implementación puede cambiar con las versiones y con el sistema operativo.

Es por esto que no todos los navegadores interpretan el HTML, CSS y JavaScript igual. Aunque a día de hoy las diferencias sean pequeñas, pueden hacer que un usuario no pueda ver correctamente la página.

¿Qué es?

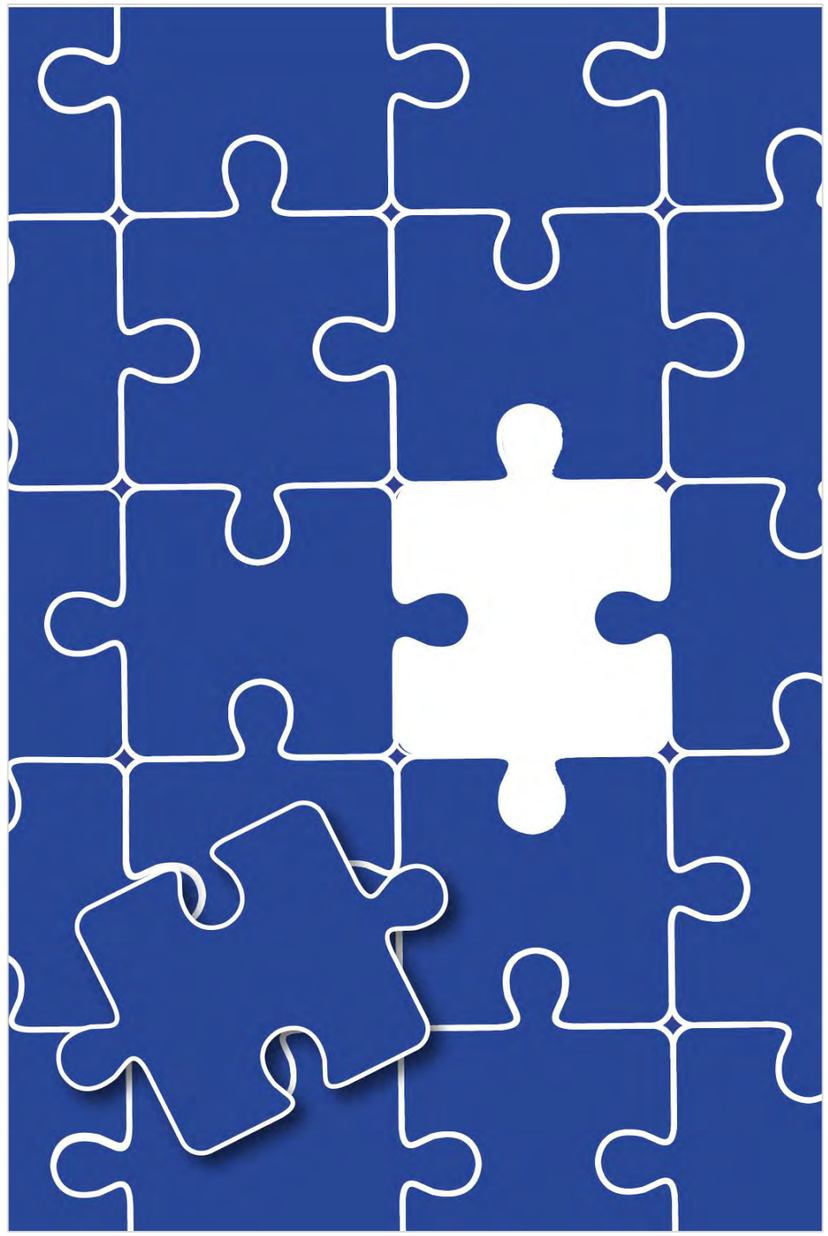
Un *polyfill* es un trozo de código que **proporciona una funcionalidad** en navegadores que no la soportan, dando a las aplicaciones web la posibilidad de tener cierta retrocompatibilidad. También son utilizados para implementar una funcionalidad propuesta o futura en navegadores actuales.

CARACTERÍSTICAS

Un polyfill **replica una funcionalidad en un navegador que no la soporta de manera nativa,** teniendo así soporte en navegadores antiguos. En general implementan múltiples formas de replicar una misma funcionalidad y así tener siempre una alternativa por si la primera no funciona.

Un ejemplo es la propiedad *sessionStorage* de la API Window que no es soportada en Internet Explorer 7. Para darle soporte existen varias técnicas: un almacenamiento local basado en cookies, usar la propiedad *localStorage*, un almacenamiento con Flash 8, etc. El polyfill se encarga, de forma **transparente para el desarrollador,** de comprobar si está soportada por el navegador y elegir la técnica óptima para dársela en caso de que no esté soportada.

La palabra Polyfill busca describir este concepto juntando *poly*, ya que siempre hay varias formas de replicar una funcionalidad, y *fill*, en cuanto a que llena los vacíos que existen en las tecnologías del navegador.



Web Storage

autentia

¿Qué es?

El navegador nos ofrece varias APIs para guardar información en la memoria del navegador. La capacidad varía según el navegador del usuario, siendo 5MB, 10MB e incluso ilimitado las capacidades más comunes

LOCAL STORAGE

Los valores guardados no tienen fecha de expiración y persisten incluso cuando el usuario cierra el navegador. Las únicas maneras de eliminar el valor guardado es a través de JavaScript, herramientas de depuración o limpiando la caché del navegador.

SESSION STORAGE

Los valores guardados se eliminan automáticamente cuando el usuario cierra la pestaña desde la que se guardó el valor o el navegador.

También se puede eliminar el valor usando JavaScript o limpiando la caché.

INTERFAZ

```
interface Storage {
  long length;
  DOMString? key(long index);
  getter DOMString? getItem(DOMString key);
  setter void setItem(DOMString key, DOMString
value);
  deleter void removeItem(DOMString key);
  void clear();
};
```

El valor que vamos a guardar tiene que ser un dato primitivo. Cuando no lo es, hay que formatearlo antes a string usando JSON.stringify(valor_complejo), y al recuperarlo JSON.parse(valor_string).

Lanza una excepción “QuotaExceededError” si el nuevo valor no puede ser guardado. (Si, por ejemplo, el usuario ha desactivado el almacenamiento para el sitio, o si se ha excedido la cuota).

COOKIE

Las cookies no forman parte de la API de Web Storage y no se deben usar para almacenar un gran volumen de datos, ya que se envían en cada petición que el navegador hace al servidor.

¿Qué son?

Son dos organizaciones encargadas de crear estándares para las aplicaciones WEB.

W3C

World Wide Web Consortium (W3C) es una organización que se encarga de recomendar y crear estándares que aseguren el crecimiento de la World Wide Web en base a 6 pilares:

1. Aplicaciones web.
2. Web móvil.
3. Voz.
4. Servicios web.
5. Web Semántica.
6. Privacidad y seguridad.

Página web oficial: https://www.w3.org

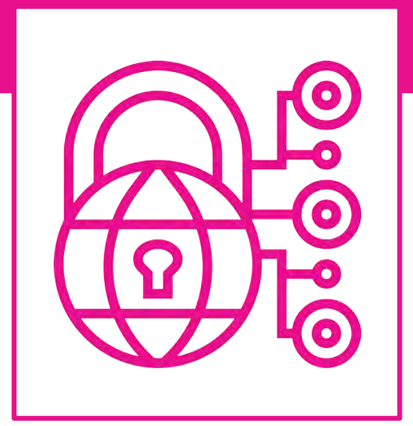
WHATWG

Web Hypertext Application Technology Working Group (WHATWG) es una organización que mantiene y desarrolla HTML y APIs para las aplicaciones Web. Se fundó en 2004 por antiguos empleados de Apple, Mozilla y Opera al haber un desacuerdo con la W3C.

Su propósito es dedicarse al desarrollo y mantenimiento de estándares HTML, prometiendo que el lenguaje HTML nunca va a desaparecer si no que va a evolucionar en el proceso.

En 2019 la W3C anunció que WHATWG sería la única que se encargaría de definir el estándar del HTML y el DOM. Pero la W3C revisará y aprobará ese estándar.

Página web oficial: https://whatwg.org



¿Qué es?

Es el área relacionada con la informática y la telemática que **se enfoca en la protección de la infraestructura computacional y todo lo relacionado con ésta** y, especialmente, la información contenida en una computadora o circulante a través de las redes de computadoras. En España, **INCIBE-CERT** es el centro de respuesta a incidentes de seguridad.

DECÁLOGO DE CIBERSEGURIDAD PARA EMPRESAS

1 **Política y normativa**
Definir, documentar y difundir una política de seguridad que determine cómo se va a abordar la seguridad mediante políticas, normativas y buenas prácticas.

2 **Control de acceso**
Implantar mecanismos para hacer cumplir los criterios que se establezcan para permitir, restringir, monitorizar y proteger el acceso a nuestros servicios, sistemas, redes e información.

3 **Copias de seguridad**
Garantizar la disponibilidad, integridad y confidencialidad de la información de la empresa, tanto la que se encuentra en soporte digital, como la que se gestiona en papel.

4 **Protección antimalware**
Aplicar a la totalidad de los equipos y dispositivos corporativos, incluidos los dispositivos móviles y los medios de almacenamiento externo.

5 **Actualizaciones**
Mantener constantemente actualizado y parcheado todo el software, tanto de los equipos como de los dispositivos móviles para mejorar su funcionalidad y seguridad.

6 **Seguridad de la red**
Mantener la red protegida frente a posibles ataques o intrusiones. Aplicar buenas prácticas a la configuración de la red WiFi. Hacer uso de una red privada virtual (VPN).

7 **Información en tránsito**
Establecer los mecanismos necesarios para asegurar la seguridad en movilidad de estos dispositivos y de las redes de comunicación utilizadas para acceder a la información corporativa.

8 **Gestión de soportes**
Desarrollar infraestructuras de almacenamiento flexibles y soluciones que protejan y resguarden la información y se adapten a los rápidos cambios del negocio y las nuevas exigencias del mercado.

9 **Registro de actividad**
Monitorizar el registro de actividad para detectar posibles problemas o deficiencias de los sistemas de información.

10 **Continuidad de negocio**
Considerar, desde un punto de vista formal, aquellos factores que pueden garantizar la continuidad de una empresa en circunstancias adversas.

Fuente: incibe.es

XSS (Cross-Site Scripting)

autentia

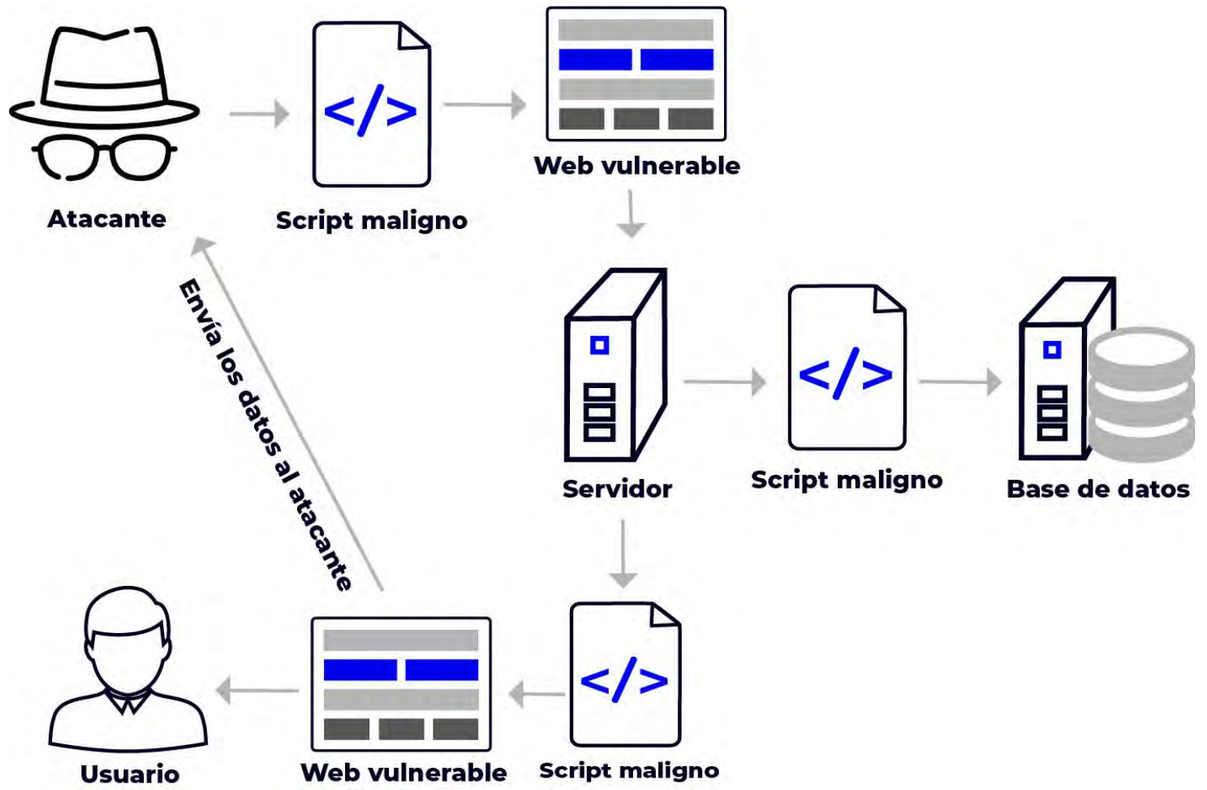
¿Qué es?

Consiste en conseguir **ejecutar código JavaScript en una página web** para tratar de acceder a datos confidenciales como las credenciales o las cookies, acceder a la cámara, etc.

VULNERABILIDADES

Existen **dos tipos** de vulnerabilidades:

- **Persistente:** el código JavaScript inyectado se queda almacenado en el servidor, por ejemplo formando parte de un comentario que aparece en un foro. Un navegador que cargue ese comentario ejecutará el código JavaScript persistido.
- **Reflejado:** el código JavaScript no se queda almacenado en el servidor, sino que el atacante de alguna manera consigue hacer que el usuario ejecute código JavaScript para robar datos sensibles.



SOLUCIÓN

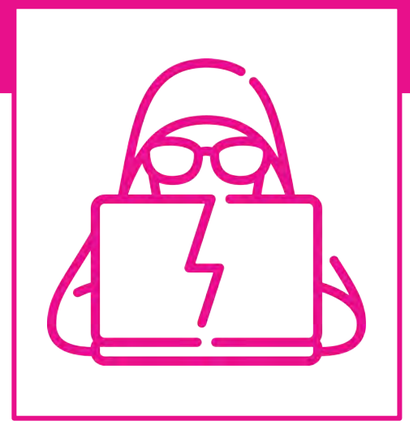
Cualquier valor que pueda ser introducido por el usuario en la aplicación debe considerase no fiable y ser procesado antes de utilizarlo.

La mayoría de los frameworks se protegen contra este ataque escapando automáticamente cualquier texto que se utilice dentro del HTML.

Es importante seguir las recomendaciones del framework que se utiliza para prevenir los ataques XSS.

CSRF (Cross-Site Request Forgery)

autentia



¿Qué es?

Consiste en hacer que **el usuario, sin saberlo, envíe una petición a algún servidor de algún sistema** en el que esté autenticado para hacer una transacción, modificar una contraseña, etc.

VULNERABILIDAD

Cuando el navegador envía una petición al servidor, envía también todas las cookies asociadas a ese dominio en la petición. Por este motivo no es necesario que el atacante tenga acceso a las cookies del usuario, solo necesita ser capaz de enviar la petición al servidor con la acción que desea y las cookies serán enviadas automáticamente por el navegador.

SOLUCIÓN

La técnica más utilizada para protegerse contra este ataque es generar un token CSRF en el lado del servidor que se envía al cliente. En cada petición HTTP que el cliente envié, el servidor espera recibir el token enviado. Si el token falta o el valor es incorrecto, la petición se rechaza.

Cabeceras HTTP más utilizadas : Seguridad

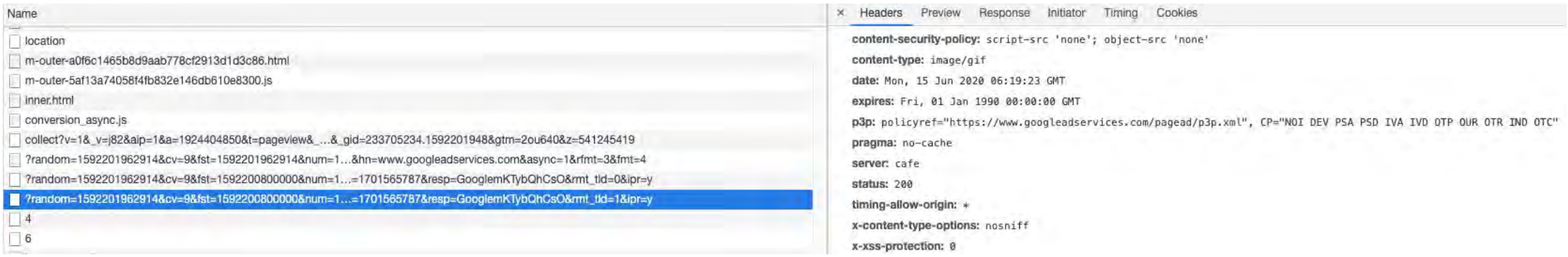
autentia

¿Qué son?

Las **cabeceras de seguridad** HTTP son una **parte fundamental para la seguridad de nuestra web**. Implementarlas dentro de nuestro sitio web **nos protege de ataques** como XSS, inyección de código o clickjacking, entre otros.

CABECERAS

- **HTTP Strict Transport Security (HSTS)**: solo se permite conexiones HTTPS. Esto evita incidentes de seguridad como el secuestro de cookies.
- **Content Security Policy (CSP)**: es una capa de seguridad adicional que ayuda a prevenir ataques de inyección de código.
  - Estableciendo whitelist de contenidos como js, images, font, etc.
  - Permite establecer políticas de navegación (a qué recursos se puede redirigir al usuario).
  - Podemos generar informes o limitar el uso de recursos como plugins.
- **Cross Site Scripting Protection (X-XSS):** habilita un filtro en los navegadores contra ataques XSS.
- **X-Frame-Options:** restringe el renderizado de objetos como <frame>, <iframe>, <embed> o <object> para prevenir los ataques de clickjacking.
- **X-Content-Type-Options:** esta cabecera evita que el usuario pueda determinar el tipo de archivo que espera el servidor e intente camuflar bajo otro tipo de archivos, por ejemplo, hacer pasar script de php como imágenes.



OWASP TOP TEN

autentia

¿Qué es?

OWASP Top 10 es un **documento de concienciación estándar** para desarrolladores y expertos en seguridad de aplicaciones web. Representa un amplio consenso sobre los riesgos de seguridad más críticos para las aplicaciones web.

TOP 10 RIESGOS DE SEGURIDAD DE APLICACIONES WEB - VERSIÓN 2017

1 **Inyección**
Los ataques de inyección, como SQL, NoSQL, comandos del S.O. o LDAP ocurren cuando se envían datos no confiables a un intérprete como parte de un comando o consulta.

2 **Pérdida de autenticación**
Las funciones de la aplicación relacionadas a autenticación y gestión de sesiones son implementadas incorrectamente, permitiendo a los atacantes comprometer usuarios y contraseñas, token de sesiones o explotar otros fallos de implementación para asumir la identidad de otros usuarios

3 **Exposición de datos sensibles**
Muchas aplicaciones web y APIs no protegen adecuadamente datos sensibles, tales como información financiera, de salud o Información Personalmente Identificable (PII).

4 **Entidades externas XML**
Muchos procesadores XML antiguos o mal configurados evalúan referencias a entidades externas en documentos XML.

5 **Pérdida de control de acceso**
Las restricciones sobre lo que los usuarios autenticados pueden hacer no se aplican correctamente.

6 **Configuración de seguridad incorrecta**
La configuración de seguridad incorrecta es un problema muy común y se debe en parte a establecer la configuración de forma manual, ad hoc o por omisión (o directamente por la falta de configuración).

7 **Secuencia de comandos en sitios cruzados (XSS)**
Los XSS ocurren cuando una aplicación toma datos no confiables y los envía al navegador web sin una validación y codificación apropiada o actualiza una página web existente con datos suministrados por el usuario, utilizando una API que ejecuta JavaScript en el navegador.

8 **Deserialización insegura**
Estos defectos ocurren cuando una aplicación recibe objetos serializados dañinos y estos objetos pueden ser manipulados o borrados por el atacante para realizar ataques de repetición, inyecciones o elevar sus privilegios de ejecución.

9 **Componentes con vulnerabilidades conocidas**
Los componentes como bibliotecas, frameworks y otros módulos se ejecutan con los mismos privilegios que la aplicación. Si se explota un componente vulnerable, el ataque puede provocar una pérdida de datos o tomar el control del servidor.

10 **Registro y monitoreo insuficiente**
El registro y monitoreo insuficiente, junto a la falta de respuesta ante incidentes permiten a los atacantes mantener el ataque en el tiempo, pivotar a otros sistemas y manipular, extraer o destruir datos.

Cabeceras HTTP más comunes

autentia

Http/2

Definición

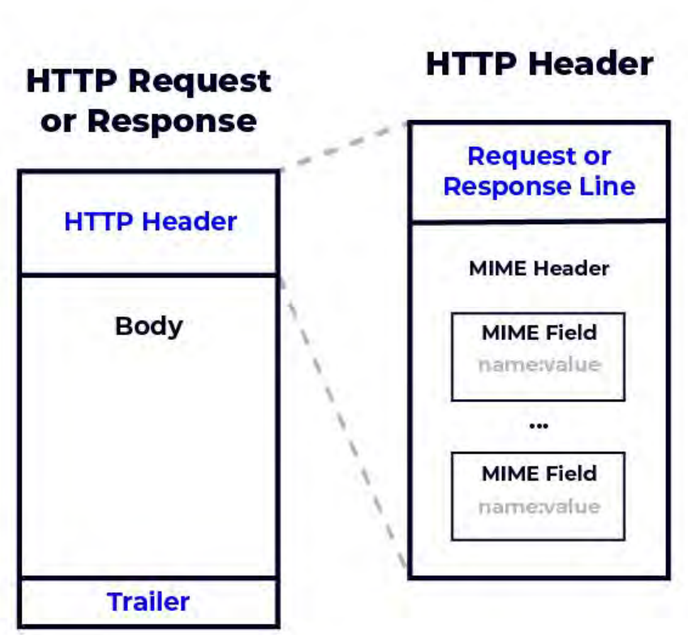
Los cabeceras HTTP **son parámetros opcionales**. Permiten tanto al cliente como al servidor **enviar información adicional**. Una cabecera consta de un nombre (sensible a mayúsculas y minúsculas), separado por ":", seguidos del valor a asignar. Por ejemplo "Host: www.example.org". Las cabeceras **varían dependiendo de si se trata de una request o una response**.

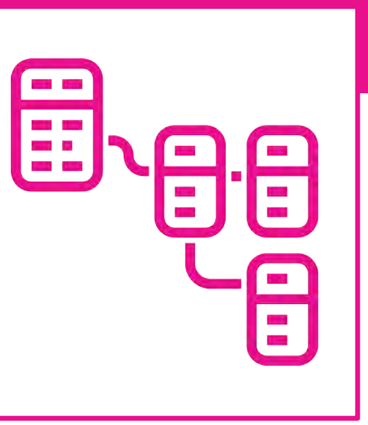
CABECERAS PRINCIPALES EN LA REQUEST

- **Cookie**: la cookie HTTP es un dato que permite al servidor **identificar las peticiones que viene de un mismo cliente**. HTTP es un protocolo sin estado. El servidor asigna una cookie a cada cliente y este las referencia usando la cabecera Cookie.
- **User-Agent**: identifica el tipo de aplicación, el sistema operativo, **la versión del navegador desde donde se hace la petición**, permitiendo al servidor bloquearla si la desconoce.
- **Host**: especifica el dominio del servidor, la versión HTTP de la solicitud y el número de puerto TCP en el que escucha (opcional).
- **X-Requested-With**: identifica solicitudes AJAX hechas desde JavaScript con el valor del campo XMLHttpRequest.
- **Accept-Language**: anuncia al servidor **qué idiomas soporta el cliente**.

CABECERAS PRINCIPALES EN LA RESPONSE

- **Content-Type**: determina el tipo de cuerpo (tipo MIME) y la codificación de la respuesta.
- **Content-Length**: indica el tamaño del cuerpo de la respuesta en bytes.
- **Set-Cookie**: es la cabecera que utiliza el servidor para enviar la cookie asignada al cliente. Con esta cookie, el servidor puede identificar y restringir el acceso a ciertas rutas al cliente. También se puede indicar la duración o fecha de vencimiento de la cookie.





¿Qué es?

Cross-Origin Resource Sharing (CORS) es un mecanismo que permite a los navegadores acceder a recursos de dominios diferentes al que están accediendo.

¿CÓMO FUNCIONA?

La mayoría de navegadores, al cargar recursos de un dominio, restringen las peticiones a recursos de otros dominios.

Para hacer la petición a estos recursos, por seguridad, el navegador utiliza CORS. Para ello sólo hace falta añadir el dominio de origen en el *header* Origin de la petición.

El servidor puede responder correctamente con algunos headers si la petición está autorizada o con error si no lo está.

Cuando las peticiones son complejas, los navegadores realizan una petición *preflighted*. En lugar de realizar directamente la petición, primero consultan mediante el método OPTIONS si es posible realizarla. En caso afirmativo, se lanza la petición real. Esto previene de cambios no deseados en el servidor.

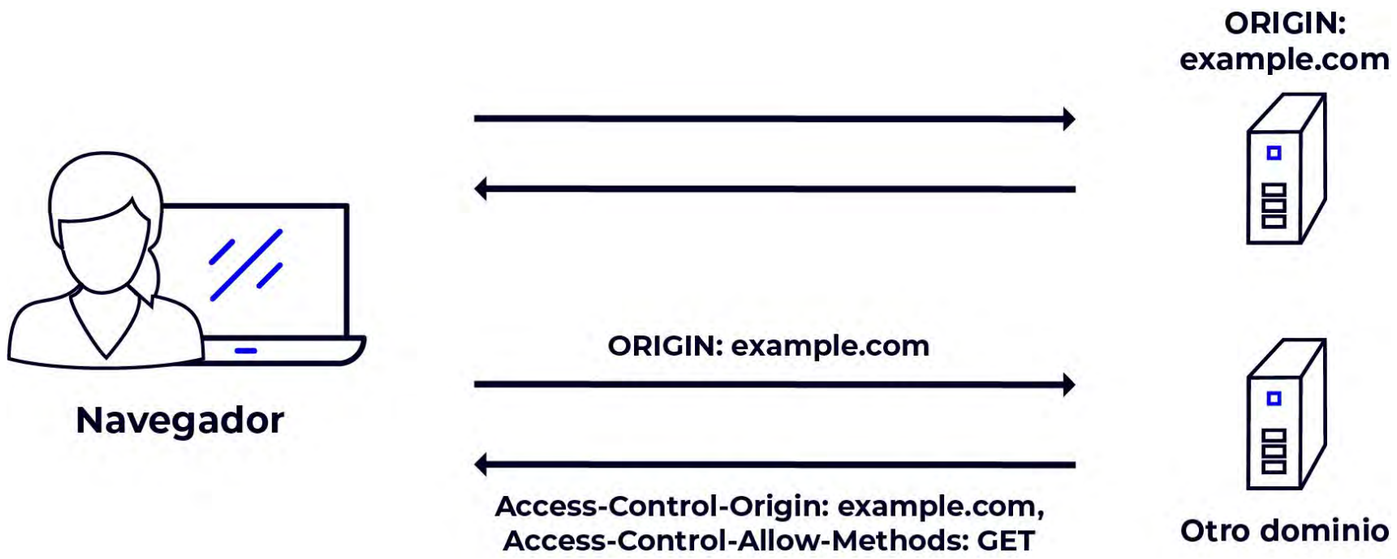
Habilitar las peticiones *cross-origin* **depende fundamentalmente del servidor.**

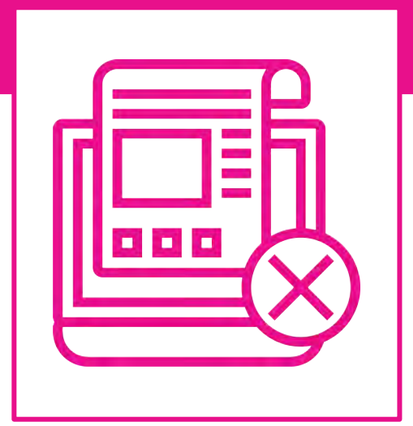
CONSEJOS DE SEGURIDAD

Es imprescindible establecer una configuración adecuada para CORS en nuestro servidor si queremos mantener nuestros datos a salvo y prevenir ataques basados en CORS.

Debemos utilizar orígenes específicos y restringir los métodos que pueden utilizarse desde cada uno de estos orígenes a los estrictamente necesarios.

El valor “*” para los orígenes permitidos sólo debería usarse cuando nuestra API sirva únicamente datos públicos que queremos que sean accedidos sin ninguna restricción.





¿Qué es?

Content Security Policy (CSP) es una capa de seguridad adicional que pretende evitar y mitigar algunos tipos de ataque como XSS o inyección de datos.

¿EN QUÉ CONSISTE?

CSP permite que el servidor ofrezca una lista de los dominios de donde pueden ser cargados distintos tipos de recursos, incluidos scripts, imágenes, etc. También se pueden definir los protocolos permitidos.

Está diseñada para ser totalmente retrocompatible. Si el navegador no soporta CSP o el servidor no lo habilita, simplemente se utiliza la política de CORS.

¿CÓMO SE USA?

Al llegar una petición al servidor, este manda una respuesta con el *header* Content-Security-Policy. El valor de este *header* consiste en una o varias directivas separadas por punto y coma.

Las directivas aceptan como valores dominios, la cadena 'self', diferentes patrones o la cadena “*”, entre otros.

EJEMPLO

```
Content-Security-Policy:
    default-src 'self' *.trusted.com;
    img-src *;
    media-src media.com;
    script-src 'none';
```

En este ejemplo sólo se permite cargar contenido desde el origen del documento y desde cualquier subdominio de trusted.com a excepción de:

- Las imágenes se pueden cargar desde cualquier sitio.
- Media solo se carga desde media.com (pero no desde sus subdominios).
- No se ejecutará ningún script.

¡OJO!

- Si se especifican default-src, style-src o script-src, entonces los estilos y scripts incluidos *inline* serán ignorados.
- ‘self’ permite cargar todo el contenido que provenga del mismo origen pero excluye sus subdominios.