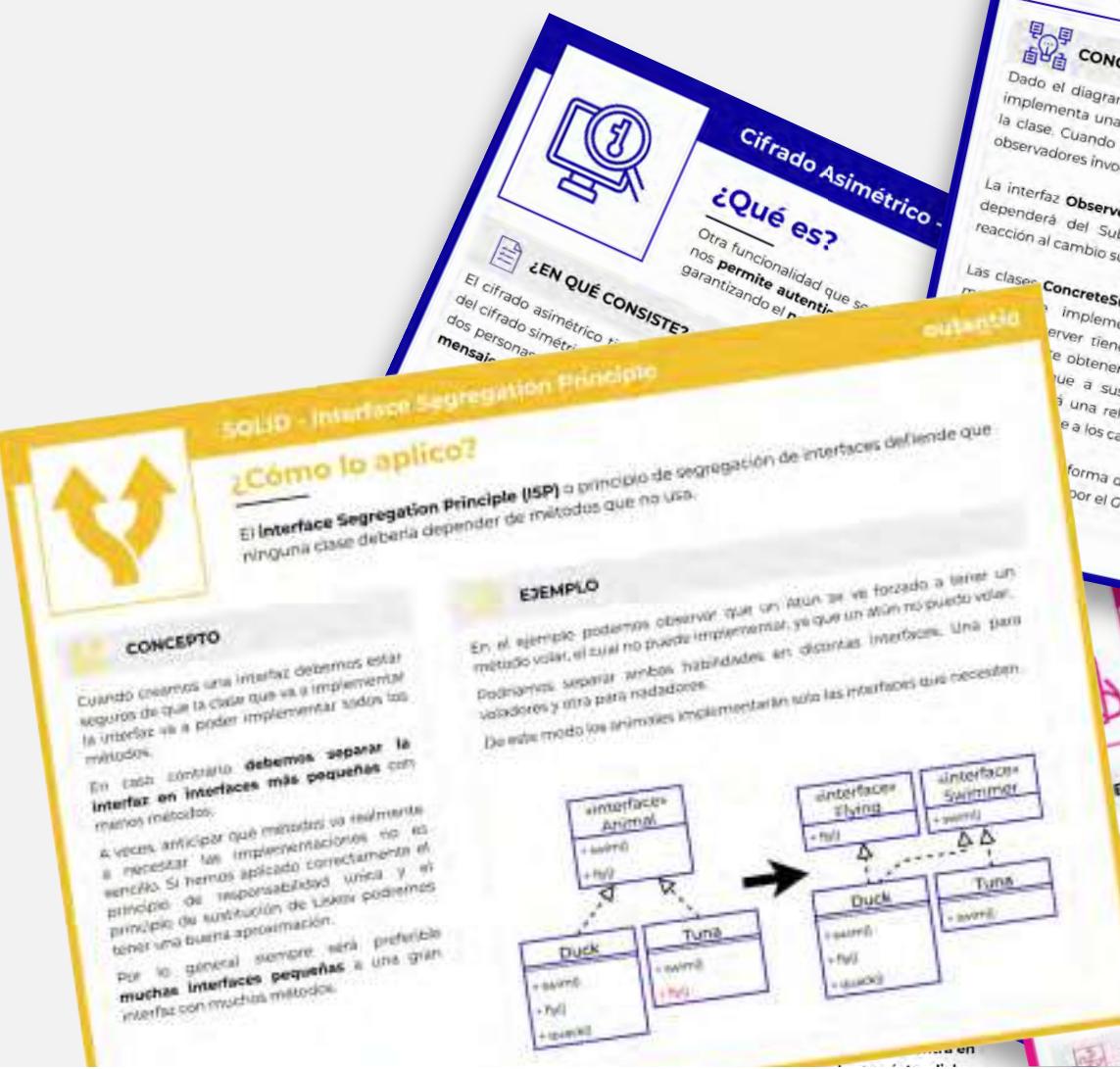


Fichas Onboarding

#KnowledgePills

autentia

V.2



Comportamiento - Observer

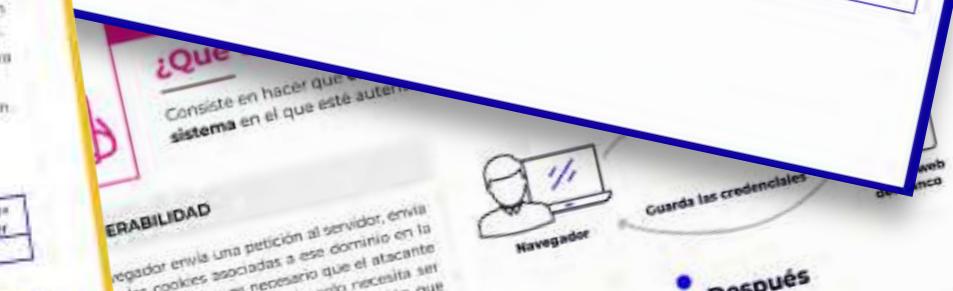
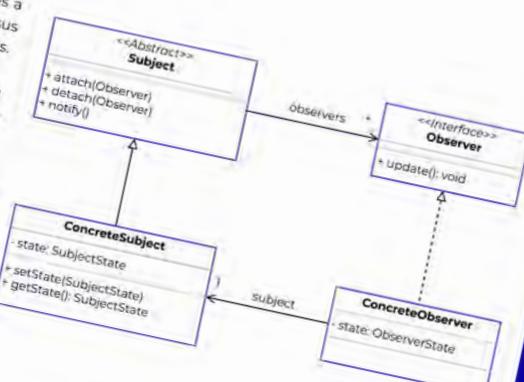
Reaccionando a cambios de estado

El patrón observador (*observer* en inglés) describe una solución que se asemeja al manejo de eventos. Principalmente es utilizado para permitir que ciertos objetos puedan reaccionar a los cambios que suceden en un momento dado en otros objetos.

Dado el diagrama, tenemos una clase abstracta **Subject**, que implementa una serie de métodos para enlazar observadores a la clase. Cuando el estado del subject cambie, notificará a sus observadores invocando el método *update* en cada una de ellas.

La interfaz **Observer** expone un solo método, cuya invocación dependerá del Subject. La lógica de este método es una reacción al cambio sucedido en Subject.

Las clases **ConcreteSubject** y **ConcreteObserver** denotan una implementación del patrón. Se observa que el observer tiene una referencia al ConcreteSubject, lo obtiene su estado. De esta manera, cuando el observer, esta implementación en forma directa al Subject para poder a los cambios sucedidos.



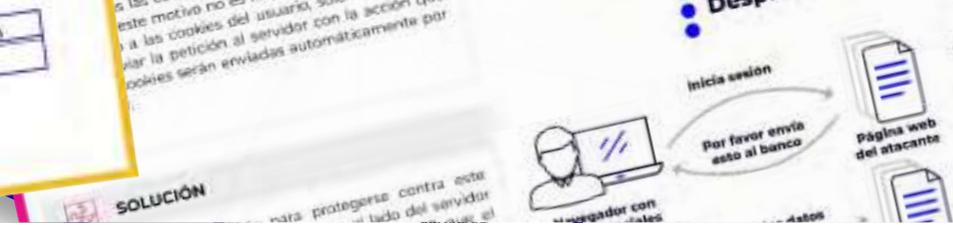
Después

Guarda las credenciales

Navegador

Servidor

Atacante



Después

Inicia sesión

Por favor envíe esto al banco

Página web del atacante

Transferir el dinero

Atacante

Guarda las credenciales

Navegador

Servidor

Atacante

Guarda las credenciales

Navegador

Índice

Estrategia tecnológica

DevOps: Cultura de valor

DevOps: Piezas básicas de la infraestructura

Front: HTML y CSS

Front: JavaScript

Front: JavaScript (Entorno)

Software Design: Principios y patrones del desarrollo de software

iOS

Back: Introducción al backend y Java

Como en anteriores ocasiones, Autentia se embarca de nuevo en la publicación de contenidos didácticos en forma de fichas. Nuestra experiencia anterior con las fichas Agile ha sido y continúa siendo un éxito de aceptación en la comunidad.

Esto nos ha impulsado a realizar una nueva publicación con una **recopilación de buenas prácticas, patrones, principios y tecnologías para el desarrollo de software** que han ido componiendo la serie de publicaciones sobre guías técnicas que usamos en el propio On-boarding de Autentia.

Las guías para directivos y técnicos que Autentia lleva publicando desde finales de primavera, se han concebido ya con la incorporación de fichas que aunque integradas totalmente en cada uno de los documentos de la serie, permiten también agruparlas por temas y manejarlas de manera independiente haciendo de ellas un activo muy valioso que se puede consultar de una manera cómoda y rápida.

En la primera versión de esta publicación se incorporan las fichas que han aparecido en los primeros títulos de las guías publicadas hasta ahora, *DevOps: cultura de valor*, *DevOps: Piezas básicas de la infraestructura*, *Front: HTML y CSS*, *Front: JavaScript*, *Front: JavaScript (Entorno)* y *Software Design: Principios y patrones del desarrollo de software*; que además las acompañamos con un bloque de fichas a las que denominamos *Estrategia Tecnológica*.

Estas fichas exponen **conceptos, herramientas y procesos** muy utilizados tanto a nivel de negocio como a nivel tecnológico pero todas con un interés y un valor apropiado que estamos seguros que la comunidad volverá a valorar.



Estrategia tecnológica



¿Qué es?

El plan estratégico de sistemas **debe ayudar a la organización a alcanzar los objetivos estratégicos**. Para ello es necesario definir los objetivos específicos del área y las directrices de la tecnología; identificar y calificar las iniciativas con respecto a su impacto/valor y esfuerzo/coste requerido y finalmente establecer el modelo de información futuro de alto nivel (datos, procesos, aplicaciones e infraestructura) que ha de soportar estas prioridades.



RESULTADOS DEL PLAN ESTRATÉGICO DE SISTEMAS

1 | Resumen para la dirección

Resumen ejecutivo que recoge las conclusiones más relevantes del plan.

2 | Otros resúmenes o productos para audiencias determinadas

Contiene información de valor para las diferentes áreas de la organización adecuando los elementos visuales y el lenguaje.

3 | Objetivos estratégicos del negocio

Reflejan la misión, aspiraciones y el contexto de la organización que servirá de base para trazar el plan de sistemas.

4 | Análisis de los sistemas actuales

Consiste en una revisión económica y organizativa del área para obtener un estado general de la situación.

5 | Análisis del mercado

La información contenida en este análisis deben ser: iniciativas, tecnologías y prácticas que se están usando y son tendencias local y globalmente. Además, las comparativas relacionadas con la inversión y gasto en el sector y los principales indicadores de gestión.

6 | Análisis del gap

Analizar el gap nos permitirá identificar oportunidades y fuentes de ventaja competitiva, definir mejoras para corregir debilidades y trazar un camino para lograr estar donde queremos.

7 | Aspiraciones del plan SI/TI

Queremos saber los objetivos y líneas estratégicas sobre las cuales tomar acción, lo cual implica identificar las áreas que queremos abordar y establecer ideas para hacerlo

8 | Iniciativas estratégicas

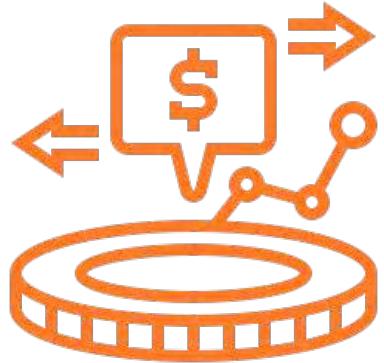
Para cada iniciativa estratégica buscaremos tener una descripción general, el objetivo que persigue y su valoración e impacto sobre el negocio.

9 | Implicaciones del plan

Debemos entender que nuestro plan se relaciona y afecta a otras partes, por tanto, necesitamos identificarlas para luego definir cómo mitigar estas implicaciones.

10 | Plan de implantación

Debemos identificar las acciones a ejecutar en corto, medio y largo plazo, las relaciones y prelaciones técnicas, los líderes de las iniciativas estratégicas, la organización, gestión y seguimiento del plan, el plan de comunicación y la motivación de las personas que participarán en este proceso.



¿Qué son?

Son dos **conceptos financieros** muy utilizados a nivel empresarial, ya **que ayudan a tener una visión más profunda de los gastos financieros de la empresa**. Es muy importante que se sepan identificar y diferenciar ya que nos **van a permitir tener un control financiero, así como** también, **definir y analizar los KPIs del negocio**.



CAPEX (CAPITAL EXPENDITURE)

CAPEX hace referencia a los gastos de capital. **Son todos aquellos gastos e inversiones asociados con bienes físicos.** O lo que es lo mismo, todos aquellos bienes comprados por la empresa.

Un buen ejemplo de ello son todos los ordenadores portátiles que se ponen a disposición de los empleados, las impresoras de la oficina, el coche de la empresa, etc.

Los gastos de capital (**CAPEX**) normalmente suelen tener varios años en su ciclo de vida. Es por ello que se **debe usar la amortización y depreciación para redistribuir el coste**.

Los gastos de capital (**CAPEX**) **se pueden distribuir de forma diferente a los OPEX** dependiendo del tipo de empresa. **En una empresa cuyos gastos de capital sean elevados**, por ejemplo, una empresa basada en los movimientos de áridos dispondrá de una flota considerable de camiones y tractores. Inversión en CAPEX mayor que en OPEX.



OPEX (OPERATIONAL EXPENDITURES)

OPEX hace referencia a todos los gastos operativos. **Son todos aquellos gastos relacionados con las operaciones y servicios.**

Como ejemplo tenemos todos los gastos en material de oficina, consumibles de las impresoras, gastos derivados del uso del coche de empresa (gasolina, parking, seguro, etc.)

Los gastos operativos (OPEX) **se pueden deducir de los impuestos durante el año fiscal en curso**. Pueden volverse excesivos a medio plazo sin ofrecer ningún rendimiento financiero por lo que se deben de controlar.

Normalmente **la distribución en gastos operativos (OPEX) será mayor en aquellas empresas de servicios y/o consultorías así como en Startups**. En esta última, debido principalmente a que normalmente se dispone de poca inversión inicial y un gasto elevado en CAPEX puede suponer el fracaso del negocio, además de ser innecesario.



Concepto de patrimonio

Según la Real Academia patrimonio hace referencia a: “**conjunto de bienes pertenecientes a una persona natural o jurídica, o afectos a un fin, susceptibles de estimación económica**”.

Tomando esto como punto de partida, se tiene que el patrimonio tecnológico de una empresa son los activos (procesos, tecnología, instrumentos) que ha adquirido la empresa y aportan un valor.



¿PARA QUÉ?

Conocer el patrimonio tecnológico de la empresa ayuda a tener conocimiento de cómo la tecnología apoya las actividades de la cadena de valor de la empresa.

Permite identificar debilidades y fortalezas de la empresa y las oportunidades de mejora que deben ser incorporadas.

El patrimonio tecnológico apoya la toma de decisiones en la estrategia tecnológica y permite realizar auditoría tecnológica.



¿CÓMO LO IDENTIFICAMOS?

A grandes rasgos podemos identificar dos tipos de tecnología dentro de la empresa:

1. La **tecnología de núcleo** está orientada a aportar valor a los objetivos de la empresa mediante el desarrollo de productos y servicios.
2. La **tecnología de apoyo** son las que no intervienen directamente en los objetivos de la empresa. Por ejemplo: herramientas informáticas, ERP, etc.



TIPOS DE TECNOLOGÍA NÚCLEO

1

Tecnologías clave: permiten a la empresa una mayor diferenciación y una mayor competitividad del producto.

2

Tecnologías básicas: son conocidas por todos y por tanto no ofrecen ventajas competitivas. Estas tecnologías son necesarias para fabricar y con el paso del tiempo las tecnologías clave se convierten en básicas.

3

Tecnologías incipientes: define a las tecnologías que se encuentran en un estado inicial de desarrollo y podrían convertirse en futuras tecnologías clave.

4

Tecnologías emergentes: son tecnologías en estado inicial de desarrollo con un impacto potencial desconocido.

Matriz ADL



¿Qué es?

Es una herramienta creada en los años 70 por la consultora de Arthur D. Little (ADL) para el análisis de la posición competitiva (situación del producto vs. situación del mercado) de una empresa/producto, el cual resulta importante para la planificación estratégica.

La matriz ADL propone el estudio desde dos ejes: la posición competitiva y la madurez de la industria.



POSICIÓN COMPETITIVA

- Dominante:** el producto es lanzado al mercado y hay poca o ninguna competencia. Se caracteriza por ser un monopolio o un mercado protegido.
- Fuerte:** se tiene una cuota de mercado fuerte y estable y no influye lo que la competencia esté haciendo.
- Favorable:** existen ventajas competitivas en determinados segmentos del mercado y se necesita protección.
- Sostenible:** existe una pequeña posición en el mercado global basada en un nicho o alguna forma de diferenciación del producto.
- Débil:** se experimentan pérdidas de participación de mercado y la línea es incapaz de mantener la rentabilidad.



ETAPAS DE MADUREZ (DEL MERCADO)

- Etapa Embrionario:** nace el mercado y crece rápidamente con la introducción de un nuevo producto, hay poca competencia y precios altos.
- Etapa de crecimiento:** el mercado continúa creciendo, las ventas aumentan y empieza a haber pocos competidores.
- Etapa de madurez:** el mercado y las cuotas de mercado son estables, el precio se reduce debido a que crecen los competidores. Se hace necesario un elemento diferenciador para mantener y conquistar el mercado.
- Fase de envejecimiento:** el producto deja de ser de interés, la demanda del producto disminuye y las empresas empiezan a abandonar el mercado. La competencia es agresiva.

MATRIZ ADL

	Embrionario	Crecimiento	Madurez	Envejecimiento
Dominante	Ampliar posición	Mantener posición	Crecer con el mercado	Mantener posición
Fuerte	Buscar mejor posición	Buscar mejor posición y ampliar la participación del mercado	Mantener la posición, buscando crecimiento relacionado con el sector.	Intentar encontrar nuevos mercados
Favorable	Mejorar la posición	Buscar mejor posición y ampliar la participación del mercado	Identificar el nicho y mantener la posición	Encontrar otros mercados o abandonar el mercado actual
Sostenible	Identificar nichos donde se pueda mejorar	Intentar mejorar la posición en determinados nichos	Encontrar un nicho y mantenerlo o salir del mercado	Abandonar
Débil	Mejorar o salir del mercado	Encontrar un nicho y protegerlo	Salir del mercado	Abandonar



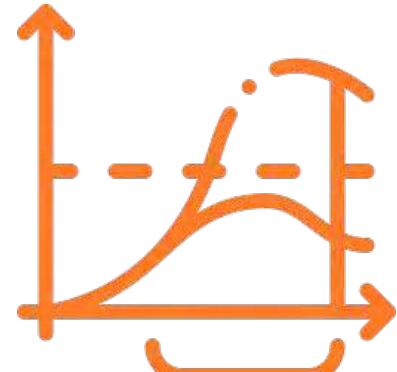
Desarrollo natural: los recursos deben ser usados para potenciar el negocio.



Desarrollo selectivo: el uso de los recursos no debe ser absoluto sino selectivo.



Abandono: negocio/producto que no son sostenibles.



Curva S de Foster

Origen

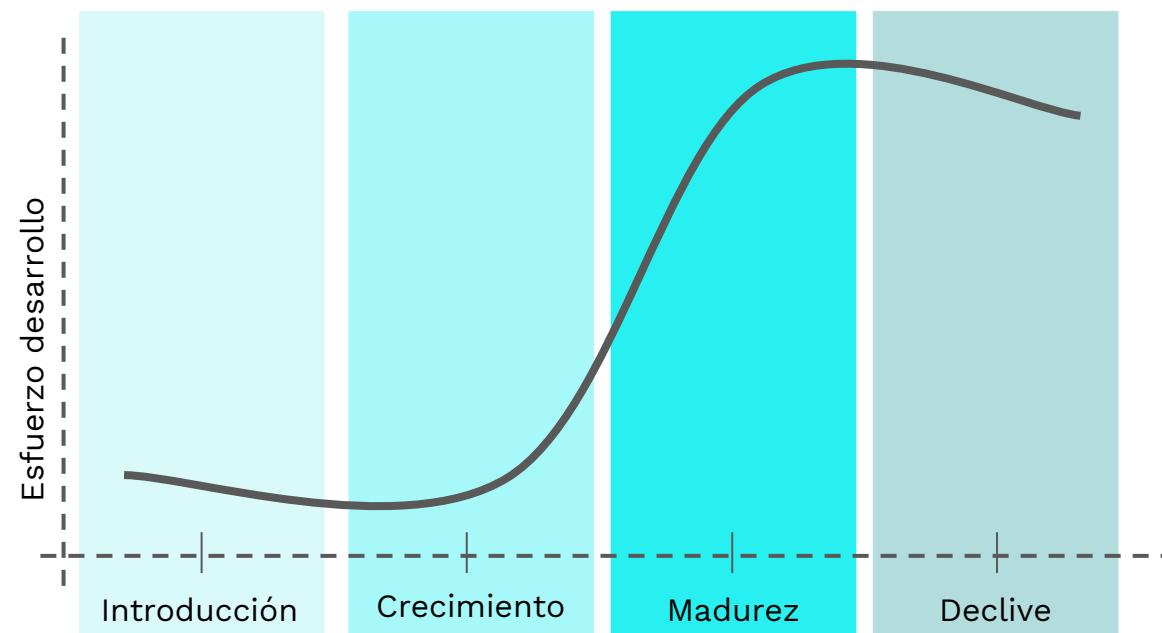
Fue creada en 1987 por Richard N. Foster. Ayuda a visualizar la evolución de la tecnología en el tiempo relacionada con el esfuerzo que implica cada etapa de madurez. Establece una relación entre esfuerzo de desarrollo y resultado obtenido, lo cual evidencia que a mayor nivel de madurez, mayor será el esfuerzo para mantener los resultados esperados.

La curva S está dividida en cuatro etapas, correspondientes al ciclo de vida de una tecnología: desde su introducción hasta el declive.

1. 2. 3
... 6. 7

ETAPAS DEL CICLO DE VIDA DE UNA TECNOLOGÍA

- Introducción:** se introduce la tecnología al mercado, el desarrollo es incipiente y el mercado es monopolístico u oligopolístico.
- Crecimiento:** se realizan mejoras en las características de la nueva tecnología, se logra una apertura del mercado con presencia de nuevos competidores y por tanto se debe aumentar el posicionamiento.
- Madurez:** se estabiliza la tecnología en el mercado, la competencia se consolida, los costos e inversiones son menores y se limitan a mejorar los atributos para adaptarla al mundo donde se encuentra sin dejar de lado la calidad por ser un factor clave para la competencia.
- Declive:** el mercado prefiere tecnologías con mejores rendimientos. No todas las tecnologías llegan a esta etapa por mantenerse en constante procesos de innovación basada en las necesidades del mercado.



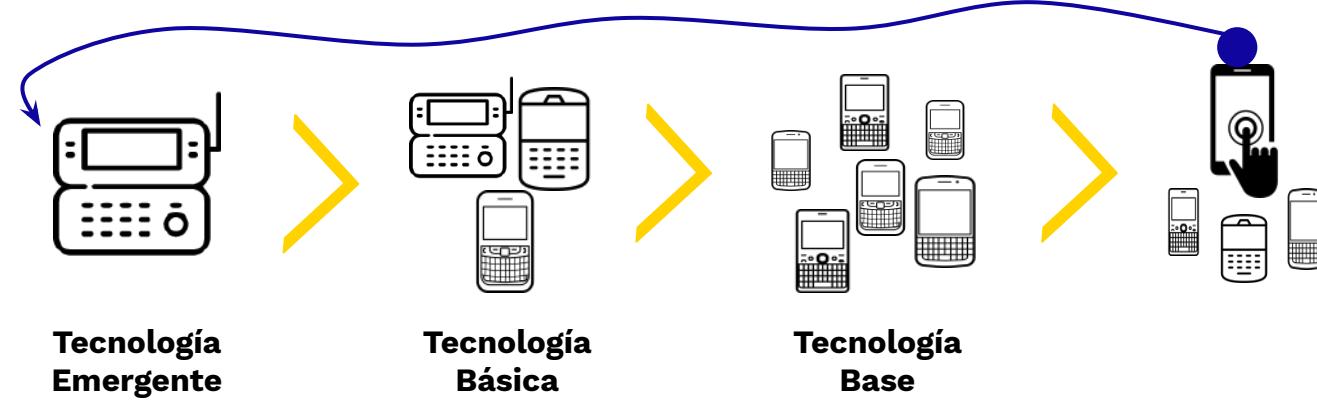
NIVEL DE MADUREZ DE LA TECNOLOGÍA

Tecnología emergente: tiene gran nivel de incertidumbre y por ser nueva posee gran potencial de desarrollo. El reto de este tipo de tecnología es incentivar al mercado a probar y adoptar su uso. Este tipo de tecnología está asociado a la primera etapa del ciclo de vida de la tecnología.

Tecnología básica: es aquella que ha sido utilizada por el mercado y ha creado una necesidad. Empiezan a aparecer competidores por lo cual necesita crecer, para ello debe aportar nuevos atributos que permitan diferenciarse. La tecnología básica se relaciona con la etapa de crecimiento.

Tecnología base: en este punto la tecnología se ha convertido en esencial en el mercado, no existen grandes diferencias entre competidores y las oportunidades de mejora son escasas. Estas tecnologías pasarán a ser reemplazadas, se asocian a la etapa de madurez de la curva S.

Teléfonos con teclados QWERTY





Coste total de propiedad

¿Qué es?

El **coste total de propiedad** (proveniente del término anglosajón Total Cost of Ownership o TCO), es un **herramienta diseñada para ayudar a los usuarios y a los gestores empresariales a determinar los costes directos e indirectos y los beneficios de un producto o sistema a largo plazo**. Se usa específicamente para la compra de equipos o programas informáticos y de modo creciente para el cálculo económico de soluciones energéticas sostenibles. *Ej: cuando se compra un software puede requerir servicios de soporte asociados, gastos en licencias anuales recurrentes, etc.*



EJEMPLO

En el siguiente ejemplo mostramos el **análisis de los costes** asociados con la compra de un vehículo con diferentes tecnologías y los elementos que incluyen como parte del TCO:

Se trata de un ejemplo real donde se han comparado los coches:

- **Gasolina:** Renault Zoe, 90 Cv y 5 puertas
- **Eléctrico:** Renault Clio Zen 1.5 DCI Energy,0 con 90 Cv y 5 puertas

Los **incentivos y fiscalidad** tenidos en cuenta son los aplicables en el PLAN MOVES 2020.

Se ha considerado un **kilometraje anual** de 15.000 Kms. El **consumo** es de 45 kWh (5'56 l) cada 100 Kms para el coche de gasolina y 13 kWh (1,6 l) para el coche eléctrico.

Los **gastos de mantenimiento** son anuales e incluyen para el coche de gasolina: aceite del motor, caja de cambios, filtros del aceite, aire, carburante, correa de distribución, embrague y revisión anual. Para los coches eléctricos, los controladores eléctricos.

Fuente: https://es.wikipedia.org/wiki/Coste_total_de_propiedad

	COCHE DE GASOLINA	vs.	COCHE ELÉCTRICO
Precio de compra	16.937 € con descuento aplicado		16.600 € con descuento aplicado
Incentivos y Fiscalidad	2053 €		4500 €
Seguro a todo riesgo	553 € anuales		571 € anuales
Equipamiento	Pantalla táctil de 7" y altavoces BOSE 1900 €		Pantalla táctil de 7" y altavoces BOSE 0 €
Precio combustible	0,033 € el Km		0,020 € el Km
Punto de recarga	0 €		660 € subvención incluida
Mantenimiento y Reparaciones	1000 €/anuales		400 €/anuales
Cuidado del Medio Ambiente	2,3 Kg de CO2 por litro + Impuesto Vehículo TM		0,108 Kg de CO2 por inicio de motor
TOTAL	20.944 € año 1		18.531 € año 1
TOTAL en 5 años	29.392 € año 5		23.615 € año 5



¿Qué es?

Es un término acuñado por **Clayton M. Christensen** en su libro *The Innovator's Dilemma*, publicado en 1997. Según su definición la Innovación Disruptiva consiste en **Innovar para crear un producto capaz de generar un nuevo mercado y desestabilizar a la competencia, que antes dominaba el escenario**.



CLAVES PARA LA INNOVACIÓN DISRUPTIVA

1. Mejor “**Learning by Doing**” que “**Paralysis by Analysis**”. La planificación estratégica ha muerto; sólo preparación (en base a sus capacidades) y rapidísima adaptación al entorno.
2. Mejor “**Act to Think**” que “**Think to Act**”. Gana continuas nuevas ventajas posicionales, por pequeñas que sean.
3. **Aprende a desaprender**. Las claves del éxito del pasado son la garantía del fracaso del futuro. Desarrolla ideas para los nuevos mercados.
4. Si quieras innovar de forma incremental, escuche a sus clientes. **Si quiere innovar de forma radical obsérvalos**.
5. Si quieras innovar radicalmente, **utiliza el tech-push** que impulsa nuevos productos o tecnologías desde el interior de la organización, sin que lo pida el mercado.
6. Un plan de empresa construido sobre conceptos de innovación radical es un **plan para aprender**, un plan para ejecutar una estrategia predefinida.
7. **Actúa según las circunstancias**, no te obsesiones a seguir un plan preconcebido. Adáptate y fluye.
8. **Permite tomar decisiones** en cada momento **a quién más información tiene**, no a quien más jerarquía ostenta
9. **Realiza marketing expedicionario**. Programa pequeñas y poco costosas incursiones en el incipiente mercado para explorarlo.
10. **Utiliza maquetas, pruebas de concepto y prototipos**. Falla pronto y de forma barata.
11. La gestión de empresas es una disciplina científica. **Observa la realidad de forma empírica**, como un investigador cuando intenta validar una teoría científica.



¿Qué es?

La cadena de valor **es una herramienta de análisis estratégico que ayuda a determinar la ventaja competitiva de la empresa.** Con la cadena de valor se consigue examinar y dividir la compañía en sus actividades estratégicas más relevantes a fin de entender cómo funcionan los costos, las fuentes actuales y en qué radica la diferenciación. Fuente: <https://economipedia.com/definiciones/cadena-de-valor.html>



ACTIVIDADES PRIMARIAS

Las actividades primarias: Un grupo de acciones enfocadas en la elaboración física de cada producto y el proceso de transferencia al comprador.

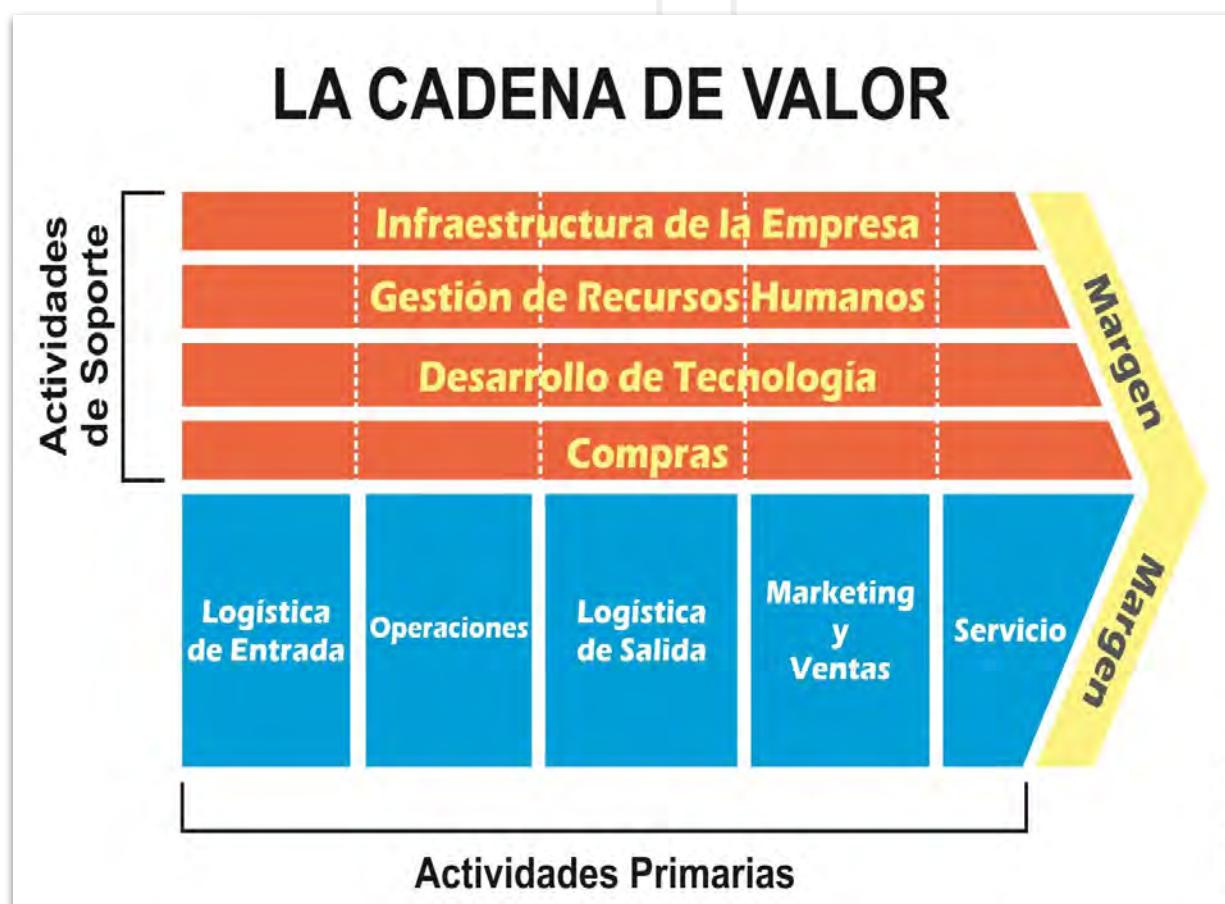
- **Logística interna:** Comprende operaciones de recepción, almacenamiento y distribución de las materias primas.
- **Operaciones (producción):** Procesamiento de las materias primas para transformarlas en el producto final.
- **Logística externa:** Almacenamiento de los productos terminados y distribución del producto al consumidor.
- **Marketing y ventas:** Actividades con las que se publicita el producto para darlo a conocer.
- **Servicio:** de post-venta o mantenimiento, las actividades de las que se encarga están destinadas a mantener, realzar el valor del producto y aplicar garantías.



ACTIVIDADES DE SOPORTE

Las actividades de apoyo: Son un soporte de las primarias y en ellas se incluye la participación de los recursos humanos, por ejemplo. Se distinguen las siguientes:

- **Infraestructura de la organización:** Actividades que prestan apoyo a toda la empresa, como la planificación, contabilidad y las finanzas.
- **Dirección de recursos humanos:** Búsqueda, contratación y motivación del personal.
- **Desarrollo de tecnología, investigación y desarrollo:** Generadores de costes y valor.
- **Compras:** Es todo aquello cuyo objetivo es abastecer y almacenar materias primas o materiales para producir.





¿Qué es?

El análisis **DAFO** son siglas que representan el estudio de las **Fortalezas, Oportunidades, Debilidades y Amenazas**, de una empresa, un mercado, o sencillamente a una persona, este acróstico es aplicado a cualquier situación, en la cual, se necesite un análisis o estudio interno y externo. **Fuente:** www.analisisfoda.com.



¿PARA QUÉ SIRVE EL DAFO?

Para **desarrollar una estrategia de negocio que sea sólida a futuro**, además, el análisis DAFO es una herramienta útil que todo gerente de empresa o industria debe ejecutar y tomarla en consideración.

Cabe señalar que, si existiera una situación compleja el análisis DAFO puede hacer frente a ella de forma sencilla y eficaz. **Enfocándose así a los factores que tienen mayor impacto en la organización o en nuestra vida cotidiana** si es el caso, a partir de allí se tomarán eficientes decisiones y las acciones pertinentes.

Además, el DAFO **ayuda a tener un enfoque mejorado**, siendo competitivo ante los nichos de los mercados al cual se está dirigiendo la empresa, teniendo mayores oportunidades en el mercado que se maneje creando estrategias para una eficaz competencia.

DEBILIDADES

Las debilidades se refieren a todos aquellos elementos, recursos de energía, habilidades y actitudes que la empresa ya tiene y que **constituyen barreras para lograr la buena marcha de la organización**.



AMENAZAS

Las amenazas son situaciones negativas, externas al programa o proyecto, que pueden atentar contra éste, por lo que llegado al caso, puede ser necesario diseñar una estrategia adecuada para poder sortearlas.



FORTALEZAS

Para realizar el análisis interno de una corporación deben aplicarse diferentes técnicas que permitan identificar dentro de la organización qué atributos le **permiten generar una ventaja competitiva sobre el resto de sus competidores**.



OPORTUNIDADES

Las oportunidades son aquellos **factores positivos que se generan en el entorno y que, una vez identificados, pueden ser aprovechados**.





¿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).

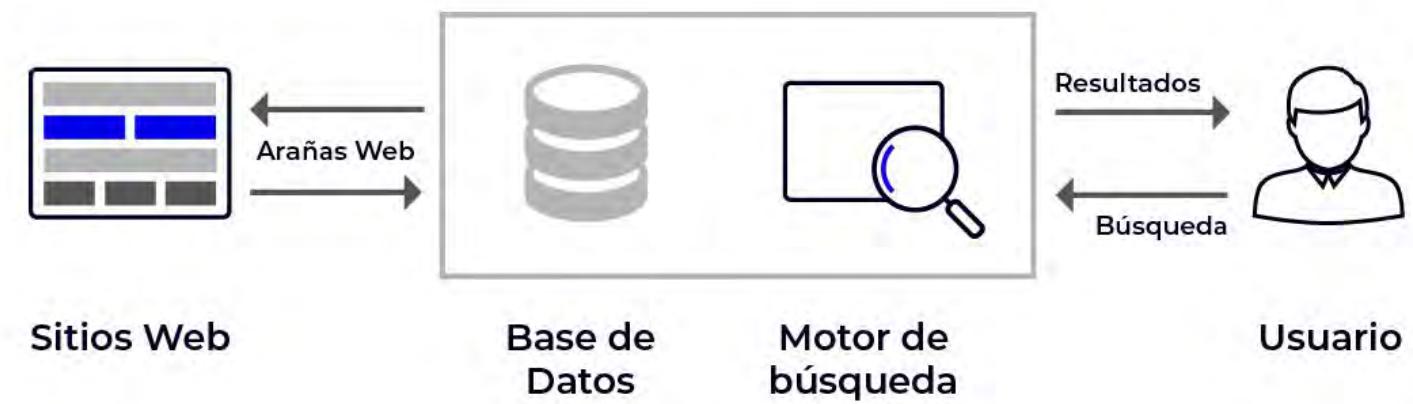


¿CÓMO FUNCIONAN LOS BUSCADORES?

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 es?

Es un **conjunto de procesos que permiten a la organización recuperarse tras un incidente grave en un plazo de tiempo que no comprometa su continuidad**. Este posibilita a las empresas estar preparadas para prevenir, protegerse, y reaccionar ante incidentes de seguridad que puedan afectarles y que podrían impactar en sus negocios. Este proceso implica las siguientes fases y documentos:

FASE 0. Determinación del alcance	Si nuestra empresa presenta cierta complejidad organizativa, abordar un proceso de mejora de la continuidad puede suponer emplear un número de recursos y un tiempo excesivo. Por tanto, es recomendable comenzar por aquellos departamentos o áreas con mayor importancia y progresivamente ir ampliando la continuidad a toda la organización.
FASE 1. Análisis de la organización	Durante esta fase recopilamos toda la información necesaria para establecer los procesos de negocio críticos, los activos que les dan soporte y cuáles son las necesidades temporales y de recursos. Se deberán elaborar en esta fase el Análisis de Impacto sobre Negocio (BIA) y el Ánalisis de Riesgos .
FASE 2. Determinación de la estrategia de continuidad	Conocidos los activos que soportan los procesos críticos, debemos determinar si en caso de desastre, seremos capaces de recuperar dichos activos en el tiempo necesario. En aquellos casos en los que no sea así, debemos establecer las diversas estrategias de recuperación. Como resultado de dicho proceso determinaremos las estrategias de recuperación más adecuadas a cada caso en el documento Estrategias de Continuidad de Negocio .
FASE 3. Respuesta a la contingencia	A partir de las estrategias de recuperación escogidas, se realiza la selección e implantación de las iniciativas necesarias, y se documenta el Plan de Crisis y los respectivos documentos Plan Operativo de Recuperación de Entornos y los Procedimientos Técnicos de trabajo .
FASE 4. Prueba, mantenimiento y revisión	A partir de la infraestructura tecnológica de nuestra empresa, desarrollaremos el Plan de pruebas y el Plan de Mantenimiento .
FASE 5. Concienciación	Además del análisis y la implantación, es necesario que tanto el personal técnico como los responsables de nuestra empresa conozcan qué es y qué supone el Plan de Continuidad de Negocio así como qué se espera de ellos.



¿Qué es?

Un KPI es una cuantificación de un desafío importante de negocio. Por tanto, la naturaleza de un KPI, es un indicador, un valor numérico en el que subyace la importancia de la conexión entre este y los retos de negocio.



¿POR QUÉ UN KPI NO ES CUALQUIER MÉTRICA?

Un KPI es una métrica, pero esto no implica que cualquier métrica sea un KPI. KPI son las siglas de **Key Performance Indicator**, siglas que definen la función de un KPI:

- **Indicator**, debe mostrar un número. *Por ejemplo, el nº de visitantes mensuales a la web es una métrica estupenda pero no un KPI. Un posible KPI válido puede ser el nº de visitantes procedentes de motores de búsqueda por una palabra clave en concreto.*
- **Performance**, debe estar conectado a un desempeño. *Por ejemplo, el nº de salas de reuniones de la organización es una métrica pero no está conectada con el rendimiento empresarial. No esperas que incrementando las salas de reuniones vaya a incrementar las ganancias.*
- **Key**, debe ser importante para tu negocio, departamento, equipo o cualquiera que sea tu escala.



UN TEST RÁPIDO: MÉTRICA VS KPI

Digamos que tenemos una métrica y necesitamos saber si es otra métrica que contribuye al desempeño del negocio de alguna manera o es un KPI. Házte la pregunta:

“¿Pagaría una cantidad significativa de dinero para obtener el valor de esta métrica dos veces mayor (o menor)?”

Si la respuesta es “no”, estamos hablando de una métrica y no de un KPI.

Por ejemplo: *El importe medio de la compra de un cliente puede ser o no un buen KPI. Dependerá de tu estrategia de negocio.*

Si tu equipo descubre una forma de hacer que el importe medio de la compra se incremente en 10% este podría ser un buen KPI.



¿POR QUÉ NECESITAMOS LOS KPI?

Porque **proporcionan una cantidad importante y valorable de información que nos ayuda en la toma de decisiones**. Un KPI no sólo te cuenta cómo está yendo el negocio en tiempo real, y por tanto, te permiten reaccionar de forma adecuada

Por ejemplo: *Si estamos utilizando un KPI para medir nuestra posición como empresa en los motores de búsqueda por determinadas palabras clave, los resultados pueden servirnos de señal de alerta y en función de ellos tomar una serie de decisiones.*



Definición

Es una técnica que consiste en hacer experimentos de forma rápida e iterativa que combinan marketing digital y creatividad tecnológica para encontrar mecanismos de crecimiento aportando valor al producto con el mínimo gasto y esfuerzo posible. Las iteraciones son análogas al método científico:

Observación -> Hipótesis -> Experimentación -> Análisis -> Conclusión



CONOCIMIENTOS NECESARIOS

Un equipo de growth hacking necesita dominar los siguientes ámbitos:

- **Marketing:** Para saber los canales y técnicas más adecuados de tu producto.
- **Analítica:** Para saber no sólo medir, si no qué medir en cada iteración y actuar en consecuencia.
- **Programación:** Para conocer las posibilidades y limitaciones técnicas de una idea.
- **Usabilidad y Experiencia de usuario:** Para saber las mejores formas de representar una idea.
- **Desarrollo de producto:** Para conocer los límites y flexibilización de tu producto.
- **Creatividad:** Para encontrar maneras nuevas de aportar valor a tu producto.
- **Psicología:** Para conocer y/o predecir el comportamiento de tus potenciales clientes.



IMPLEMENTACIÓN

En las iteraciones se han de llevar una serie de pasos que pueden variar en función a tu producto, pero en términos generales serían:

1. **Observación:** Hacer market fit, entender bien tu producto, su tono, estilo de comunicación y el cliente al que va enfocado.
2. **Elaboración de una hipótesis:** Enfocada al comportamiento de tu cliente, esto se traduce en una o varias **ideas con una meta bien definida y unos canales específicos de despliegue**. Las ideas han de ser priorizadas, por ejemplo mediante el método ICE:
 - **I:** Impacto
 - **C:** Confianza
 - **E:** Facilidad (easy)
3. **Experimento:** Se ejecuta la idea con los criterios y canales establecidos.
4. **Análisis:** Mediante las herramientas de análisis, ver que ha funcionado, como y porqué.
5. **Conclusiones:** Estudiar el análisis para ver cómo fidelizar, escalar y hacerlo viral.



Uso

A día de hoy existen múltiples ejemplos de implementación de Growth Hacking, no obstante, la propia naturaleza de esta técnica consiste en ser creativo e idear formas nuevas de encontrar el crecimiento de nuestro producto.



TÁCTICAS

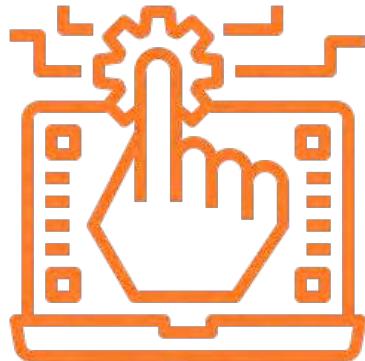
Existen numerosas estrategias de implementación de growth hacking:

- **Gamificación:** Aplicación de marketing con mecánicas de juego, ofreciendo una serie de incentivos a los usuarios.
- **Modelo freemium:** Permitir un acceso a tu producto con limitaciones que son anuladas con pago o suscripción.
- **Invitaciones exclusivas:** Creando hype al ser la única forma de acceso a nuestro producto/servicio.
- **Plataformas de terceros:** Aprovechar el posicionamiento de terceros para conseguir visibilidad compartiendo contenido relacionado con tu producto.
- **Crowdfunding:** Mediante el cual consigues financiación y conocimiento profundo de tu cliente.
- **Remarketing / Retargeting:** Enfocado al cliente que ya ha mostrado interés en tu producto para interactuar con él.
- **Venta urgente:** Consiste en ofrecer tu producto con un precio especial, con límite de tiempo o límite de unidades.
- **Skyscraper:** Analiza el producto de la competencia y mejorarlo.



EJEMPLOS PRÁCTICOS

- Crea una lista de espera de usuarios o una landing con una cuenta atrás para cuando vayas a lanzar tu producto.
- Desarrolla el look and feel de tu producto en base a una encuesta en redes sociales con N posibles aspectos, preguntando cuál es el más atractivo.
- Cuando un usuario fidelizado deje una compra sin terminar, envíale un correo/push de recordatorio.
- Haz webinars/tutoriales en relación a tu producto, es un buen punto de entrada de posibles clientes.
- Usa pop-ups inteligentes, por ejemplo, tras finalizar una venta a un cliente, solicita un email para enviar algo en lo que tenga interés.
- Haz concursos o encuestas en redes sociales en relación a tu producto que sean viralizables.
- Regala tu producto a influencers para que te den visibilidad.
- Ofrece un registro a través de redes sociales para facilitar el Engagement.
- Mostrar el número de personas interesadas en tu producto en las últimas N horas.
- Ofrece recompensas por compartir contenido tuyo en redes sociales.



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?

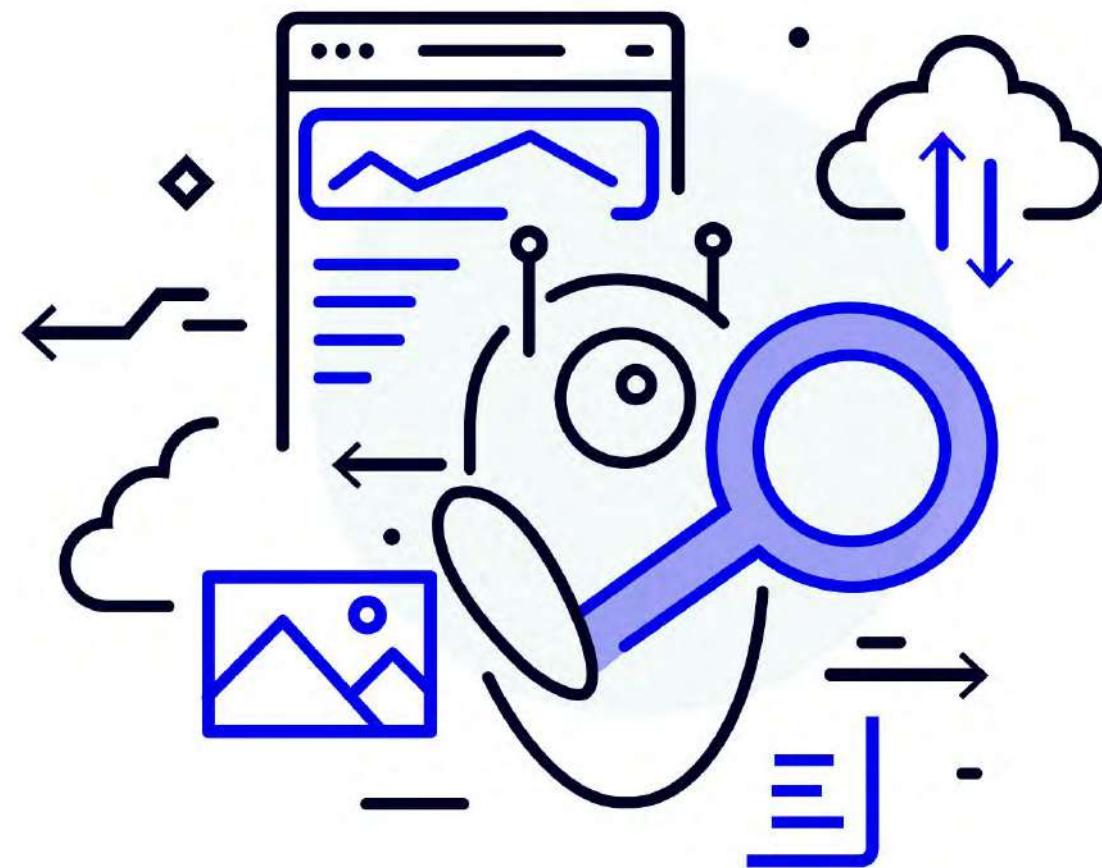
SRE trata los problemas de operaciones como si se tratase de un problema de software. **Su misión es proteger la estabilidad del sistema sin dejar de agregar y mejorar el software** detrás de todos los servicios, **vigilando constantemente la disponibilidad, latencia, rendimiento y consumo de recursos**.

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 usuarios 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 reanudarlos.





¿Qué es?

Un **CTI** es un sistema informático cuyo fin es interactuar entre una llamada telefónica y un sistema informático, coordinando esta interacción.



DEFINICIÓN

Aunque **CTI** son las siglas de Computer Telephony Integration, el término se ha ampliado para **abrirse a otros canales de comunicación** entre la empresa y sus clientes como pueden ser el email, web, aplicaciones, etc.

En sus inicios cualquier interacción telefónica con la empresa se hacía de forma manual; para ello se empleaban registros inmensos donde se tomaban todo tipo de anotaciones sobre la llamada (origen, duración, fecha, etc.). El CTI ha tomado hoy en día el relevo y **realiza estas tareas de forma automatizada**, haciendo todo el trabajo más eficaz y fluido repercutiendo en una mejora sustancial de la productividad.

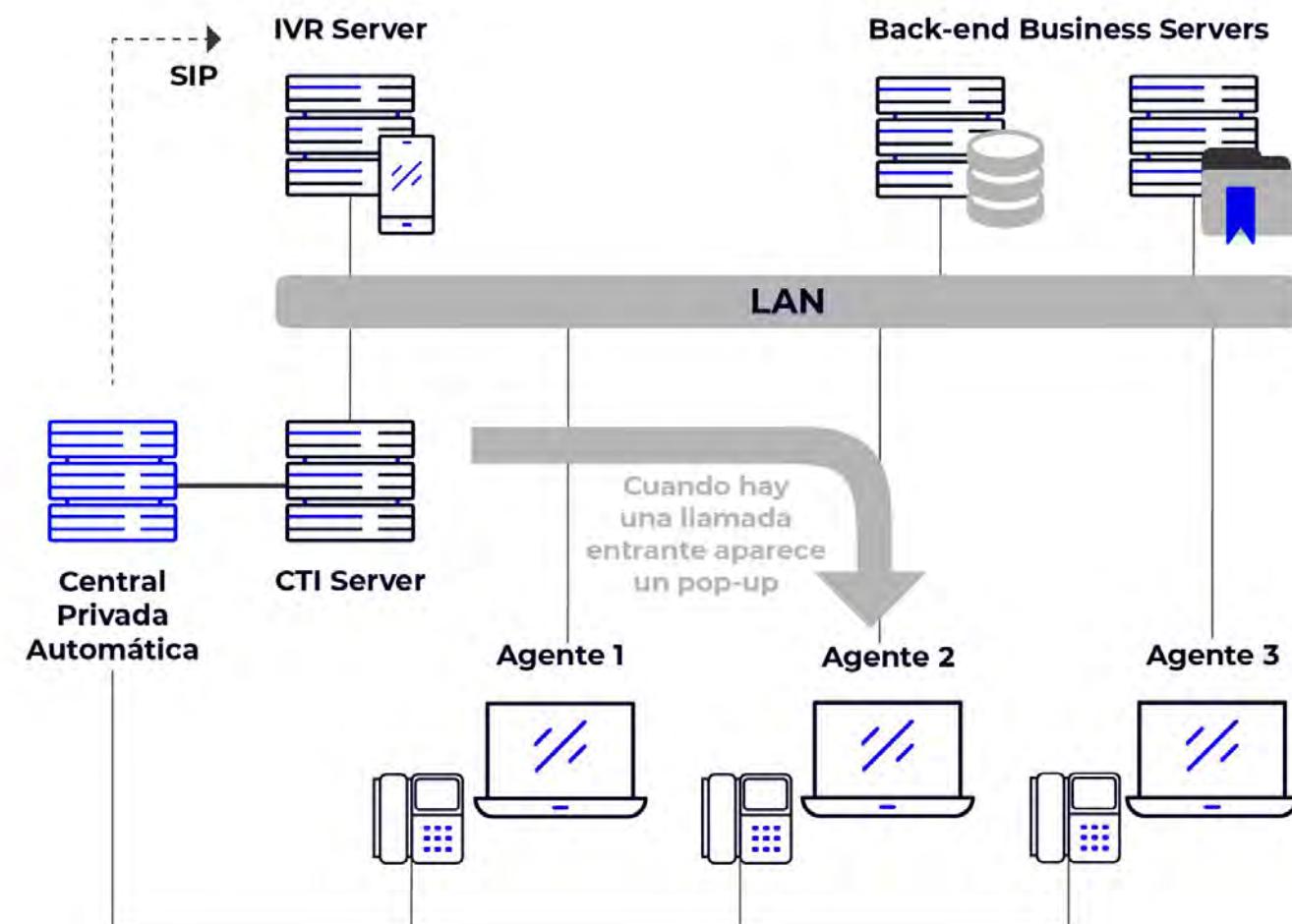
Disponer de un CTI en la empresa ofrece grandes **ventajas** entre las que destacan:

- Respuesta automatizada de llamadas por IVR las 24h.
- Identificador de llamadas y de cliente.
- Marcación automática.
- Funciones avanzadas: enrutamiento, funciones de informes, automatización de procesos y mezcla multicanal.
- Monitorización de la calidad del servicio y grabación de llamadas.



INFRAESTRUCTURA

Por otro lado, **IVR** (Interactive Voice Response) consiste en un sistema telefónico que es capaz de recibir una llamada e interactuar con el humano a través de grabaciones de voz y el reconocimiento de respuestas simples, como «sí», «no» u otras.



Balanced Scorecard



¿Qué es?

Es una **metodología utilizada para definir y hacer seguimiento a la estrategia de una organización**. Sus autores son Robert Kaplan y David Norton y presentaron el concepto en 1992 en la revista Harvard Business Review.



DEFINICIÓN

También llamado Cuadro de Mando Integral, se basa en un correcto equilibrio entre los elementos de la estrategia global y los elementos operativos de la misma:

Elementos Globales	Elementos Operativos
<input type="checkbox"/> Misión	<input type="checkbox"/> KPIs
<input type="checkbox"/> Visión	<input type="checkbox"/> Iniciativas Estratégicas
<input type="checkbox"/> Valores Centrales	
<input type="checkbox"/> Perspectivas	
<input type="checkbox"/> Objetivos	

Los propósitos de Balanced Scorecard son:

- Describir y comunicar la estrategia.
- Medir la estrategia.
- Hacer un seguimiento de las acciones que se están tomando para mejorar los resultados.

El Balanced Scorecard sugiere que evaluemos el desempeño de la estrategia de la organización desde cuatro perspectivas.



PERSPECTIVAS

1. **Perspectiva financiera**: consiste en conocer los ingresos reales de la organización y su capacidad presupuestaria.
2. **Perspectiva del cliente**: evalúa varios factores que influyen en la experiencia del cliente.
3. **Perspectiva de procesos**: permite optimizar el funcionamiento interno para garantizar agilidad y eficacia.
4. **Perspectiva de aprendizaje y crecimiento**: permite analizar la infraestructura de la organización y así crear valor futuro.

El Balance Scorecard propone un balance entre los indicadores de cada una de estas perspectivas y permite tener una visualización de todo lo que sucede en la organización en términos estratégicos, logísticos presupuestarios, etc.

Los beneficios de su implementación son entre otros: ayuda a mantener la estrategia siempre visible y como foco de la creación de estadísticas, transforma la visión de la organización en acciones reales, ayuda a alinear todas las áreas y las actividades de la organización en función del cumplimiento de la visión, estimula la transformación organizacional a través de la estrategia y produce mejoras en los procesos organizacionales.

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”



PROPOSITOS 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.
- **Testable** - 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?

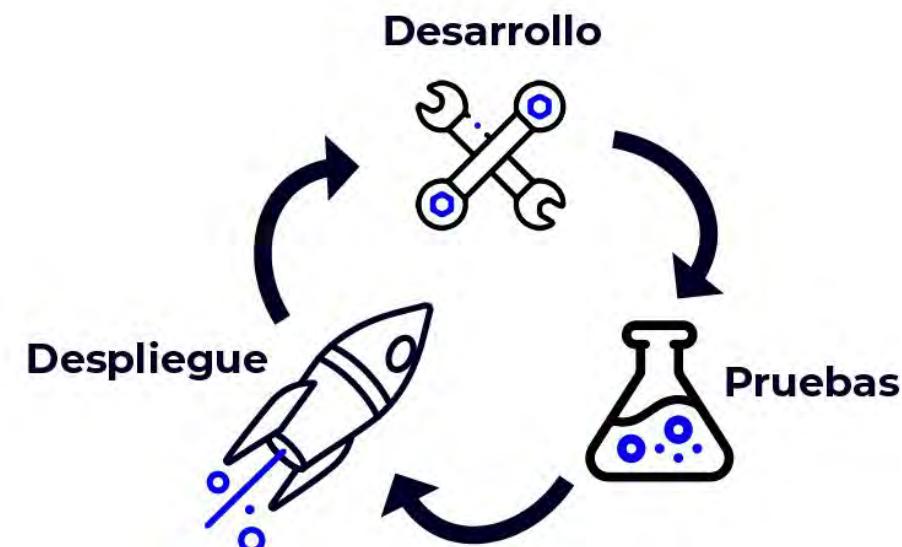
Durante el proceso de integración continua se pasan ciertas fases o tareas a ejecutar como por ejemplo instalar las dependencias necesarias, lanzar los tests, entre otras. En caso de que alguien del equipo decida hacer un Pull Request/Merge Request, comprobamos que dicho pipeline ha pasado correctamente 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**.



Despliegue continuo



¿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 continua.

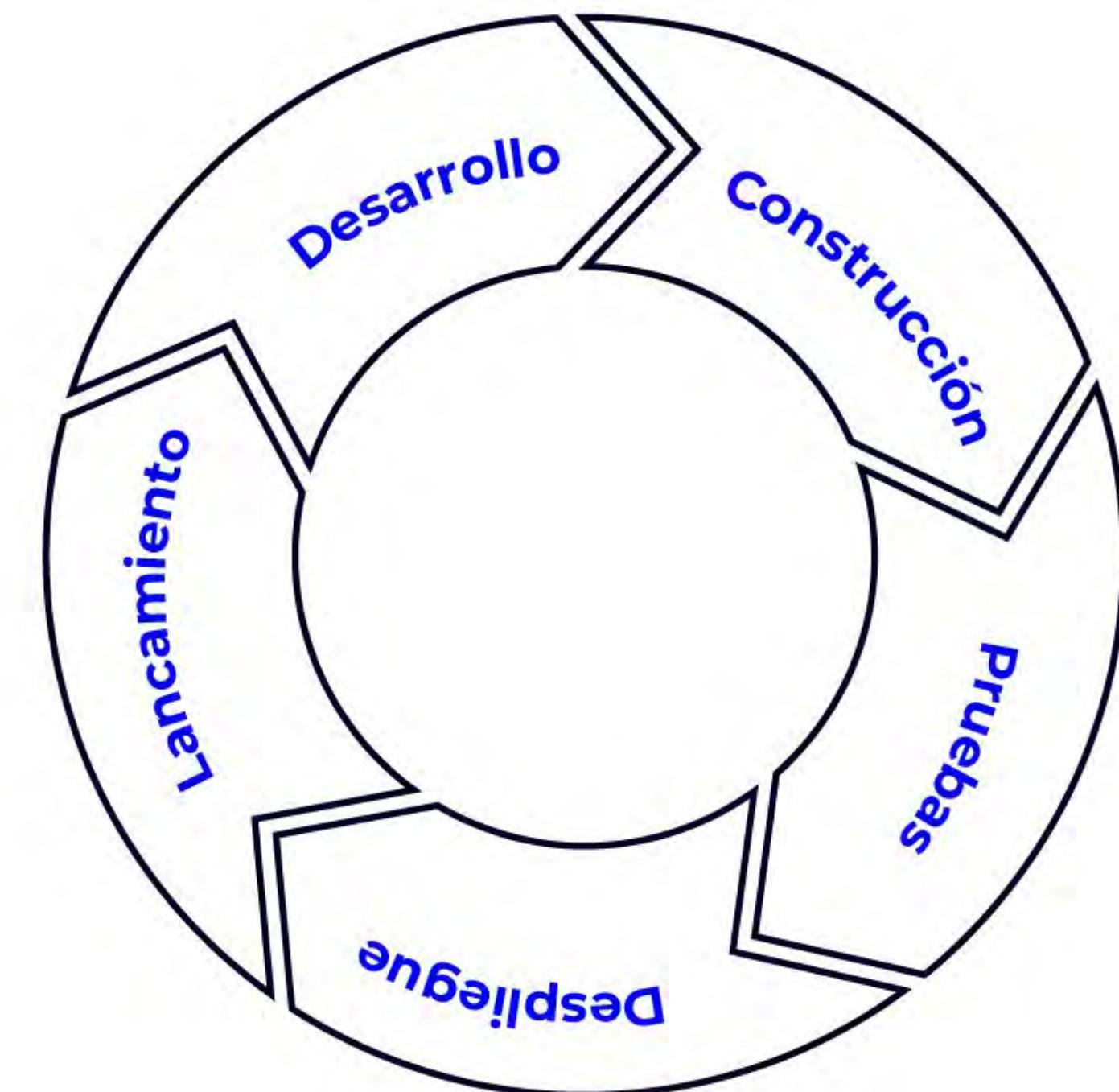


¿EN QUÉ CONSISTE?

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 períodos cortos para obtener feedback rápido del cliente.





¿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.



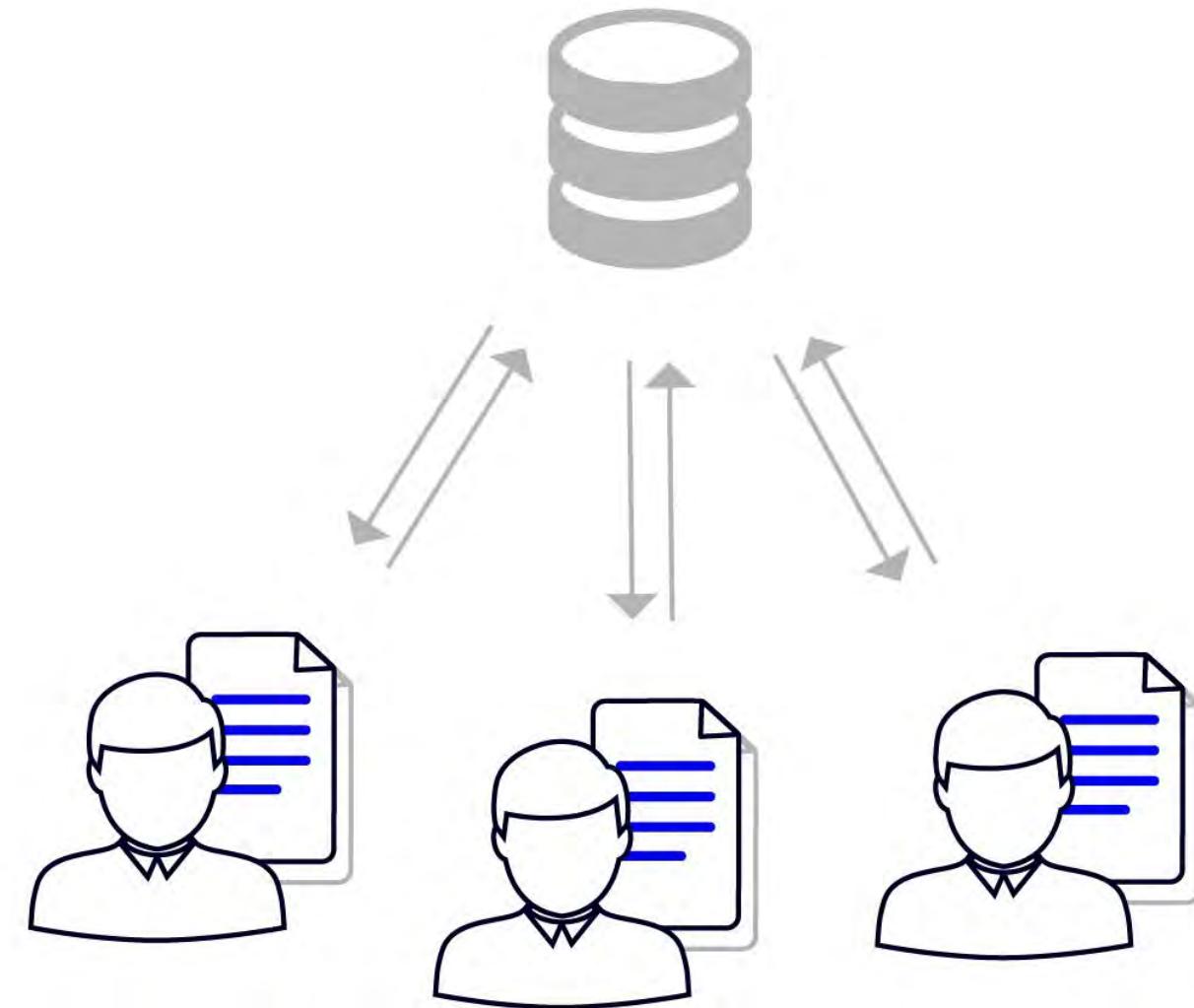
¿EN QUÉ CONSISTE?

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

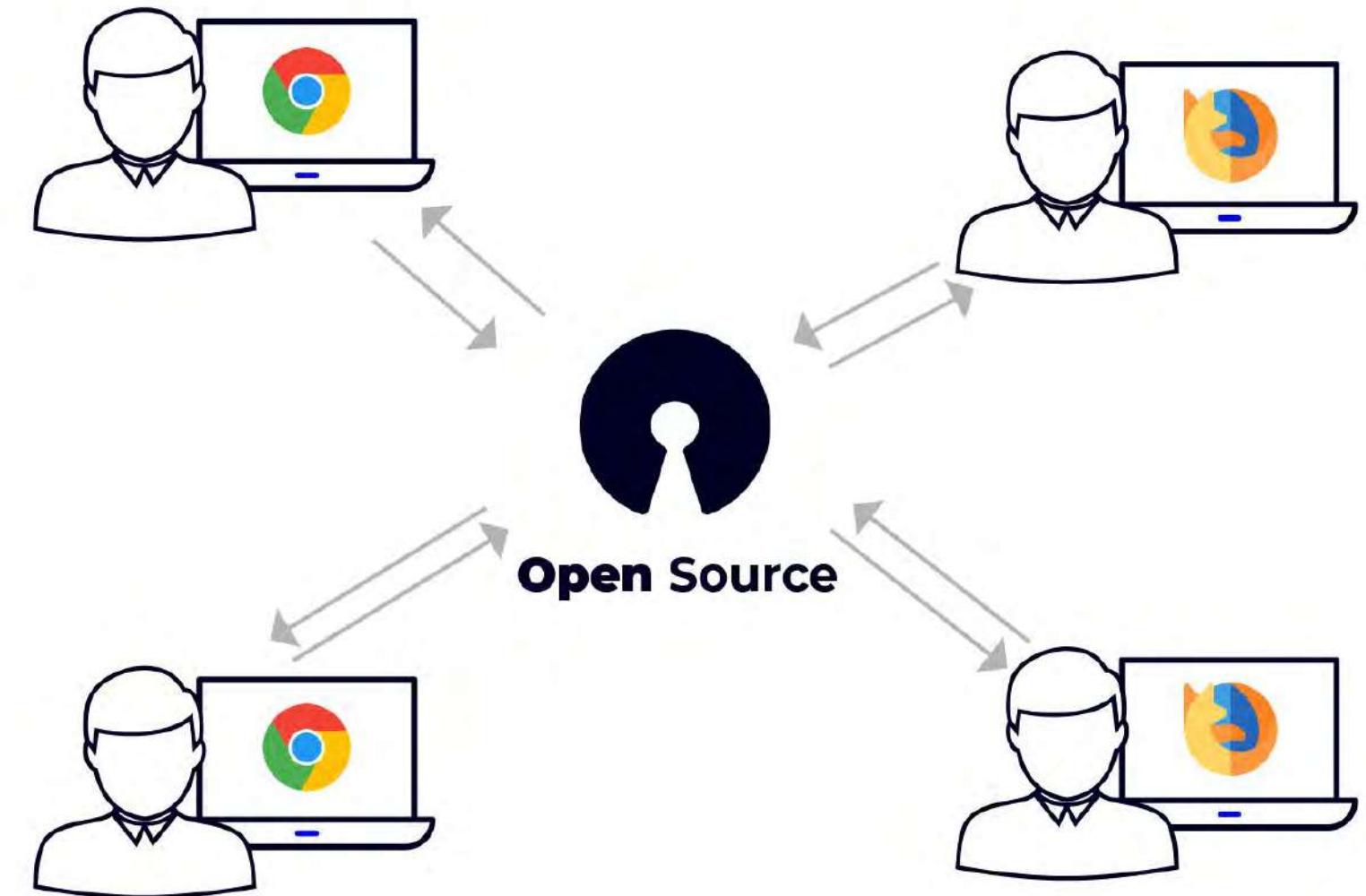
Feedback: soporte continuo para apoyar y mejorar una solución que beneficia tanto a la empresa como a la comunidad aumentando la fiabilidad en cada fase del desarrollo gracias al respaldo de todos los colaboradores.

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**.

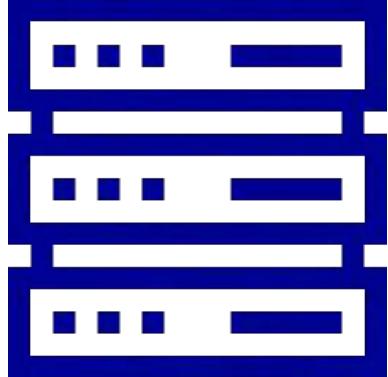
Incremento del número de versión **MAJOR**, **MINOR** o **PATCH** por cada nueva release del artefacto software en función de la naturaleza del cambio. Dada una release con número de versión X.Y.Z y una nueva versión a partir de ella:

- **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?

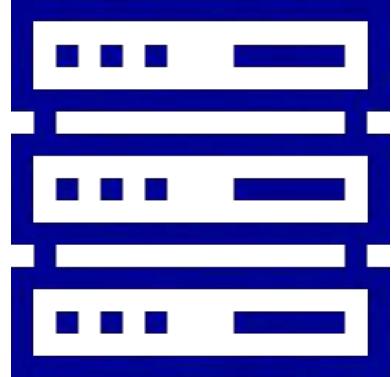
Una base de datos es una especie de “almacén” que nos permite guardar grandes cantidades de información para su posterior **recuperación, análisis y/o transmisión**. Podemos encontrar diversos tipos de bases de datos y se clasifican de acuerdo a las necesidades que busquen solucionar en dos grandes bloques.



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



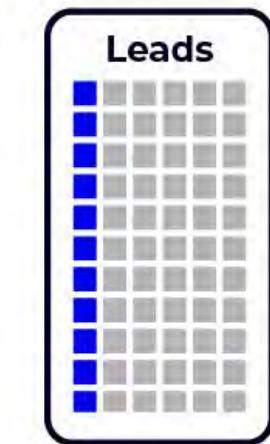
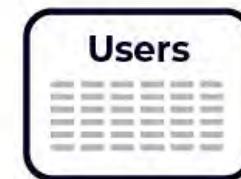
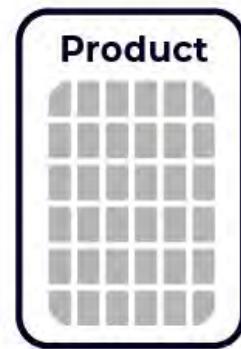
¿Qué es?

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.

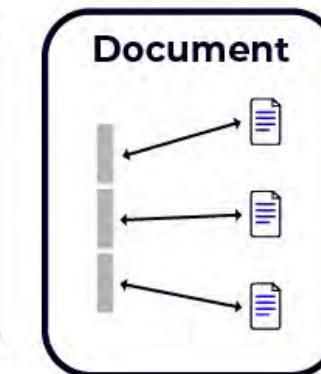
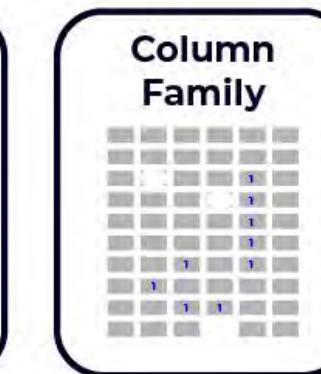
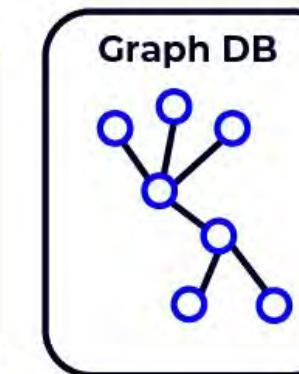
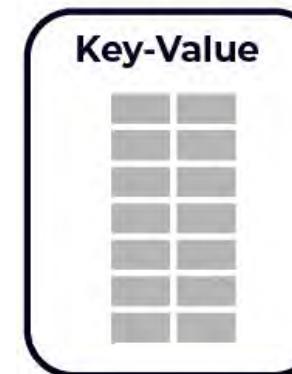


TIPOS

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 Gráficos	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



SQL



NO SQL



¿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

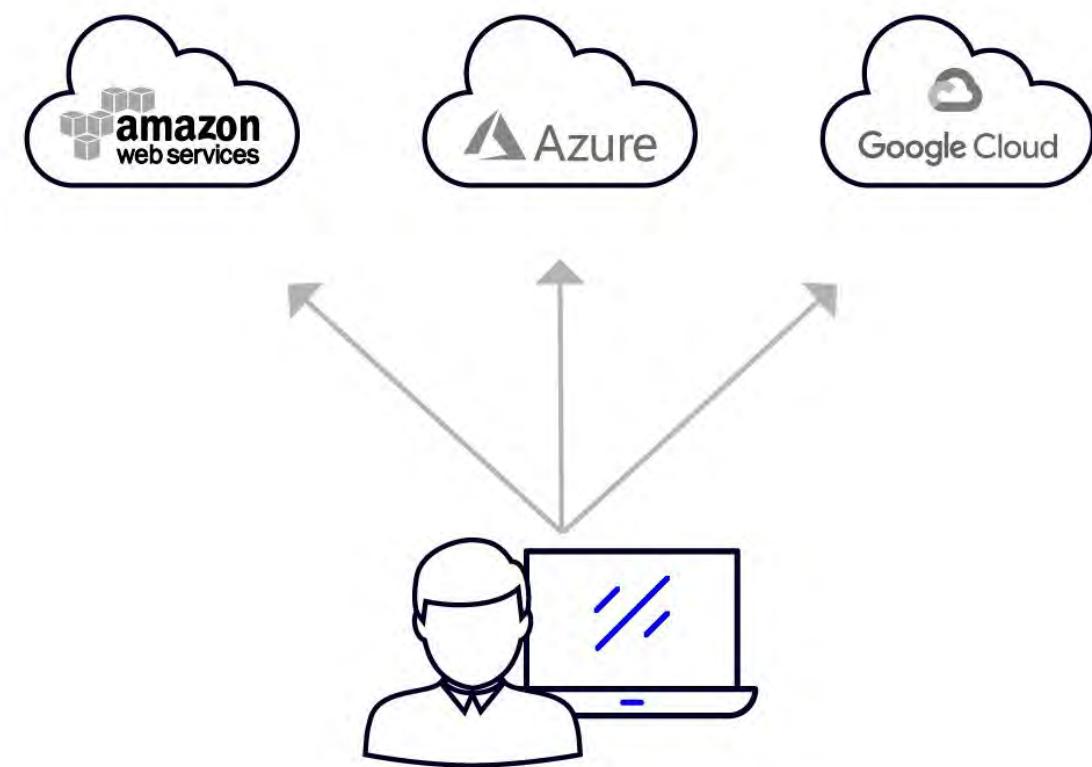
Los entornos en la nube son manejados por terceros , por lo que nosotros no tenemos acceso a las máquinas físicas. Los servicios más comunes son a la hora de montar entornos son PaaS o IaaS. Estos servicios suelen implementar sistemas basados en pagar a medida que se necesiten más recursos (servidores, 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

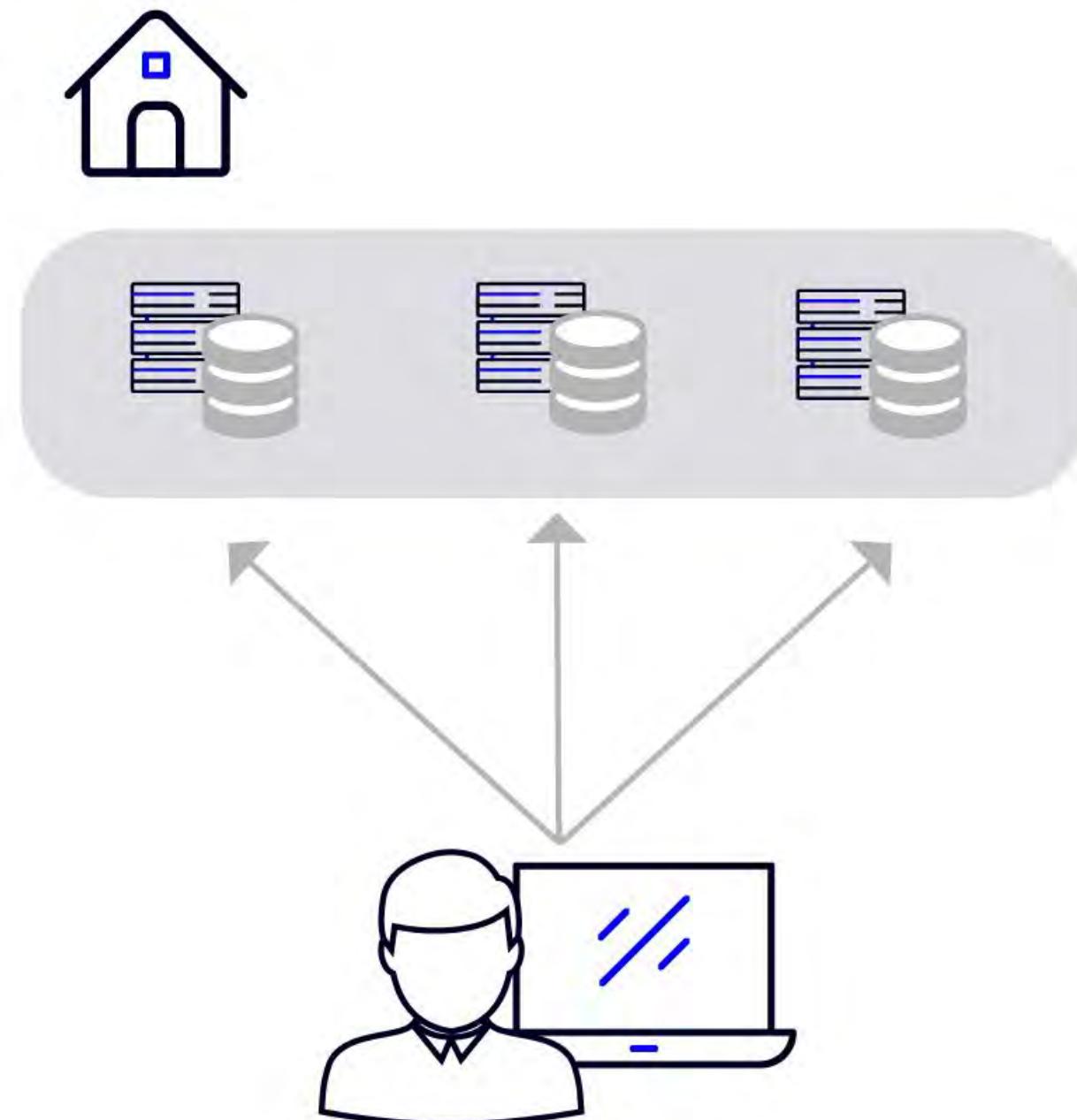
Existen distintos tipos de entornos de producción que según las necesidades de negocio, se podría optar por una opción u otra. Estos entornos pueden ser tanto **on-premise**, como en la **nube**, como **híbridos**.

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**).





API gateway

¿Qué es?

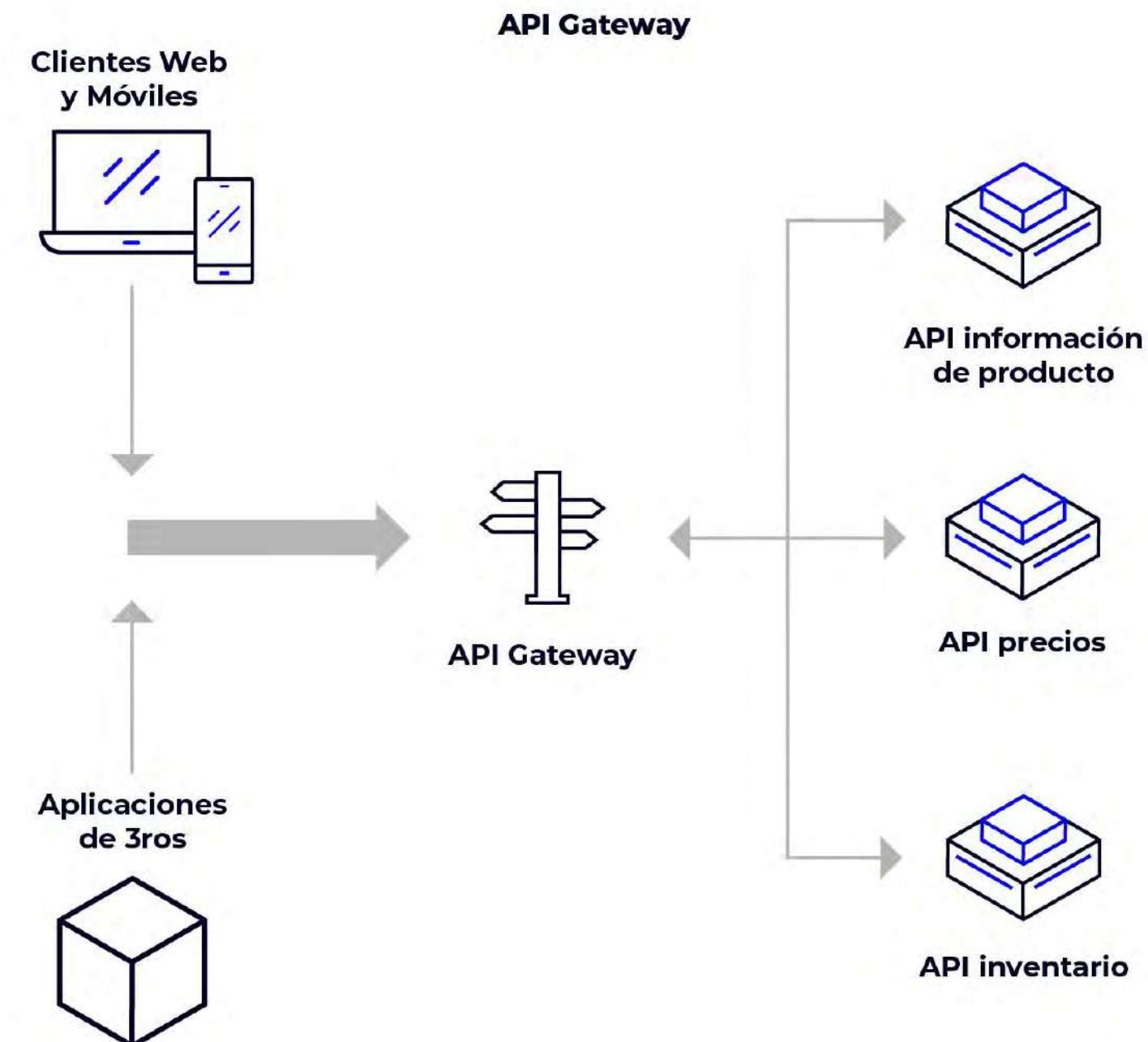
Sistema intermediario que proporciona una interfaz para hacer de enrutador entre los servicios y los consumidores desde un **único punto de entrada**. Es similar al patrón estructural Facade.



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.**



Alta disponibilidad



¿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.



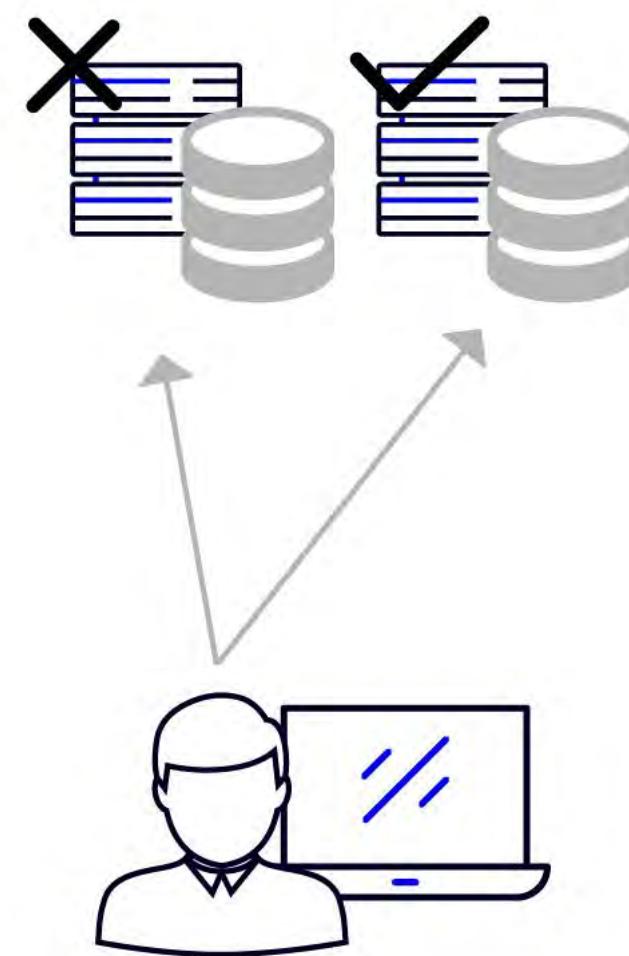
¿EN QUÉ CONSISTE?

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:

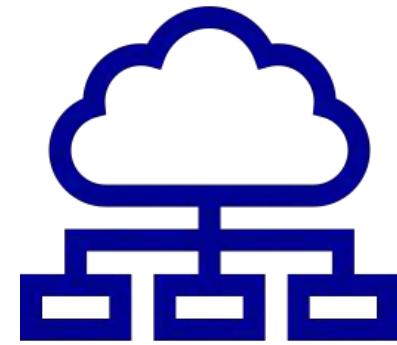
- **Desastres naturales** (terremotos, inundaciones, incendios).
- **Cortes de suministro eléctrico.**
- **Errores operativos.**

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

Definición

SaaS, PaaS e IaaS son tres tipos de servicio comúnmente asociados a 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

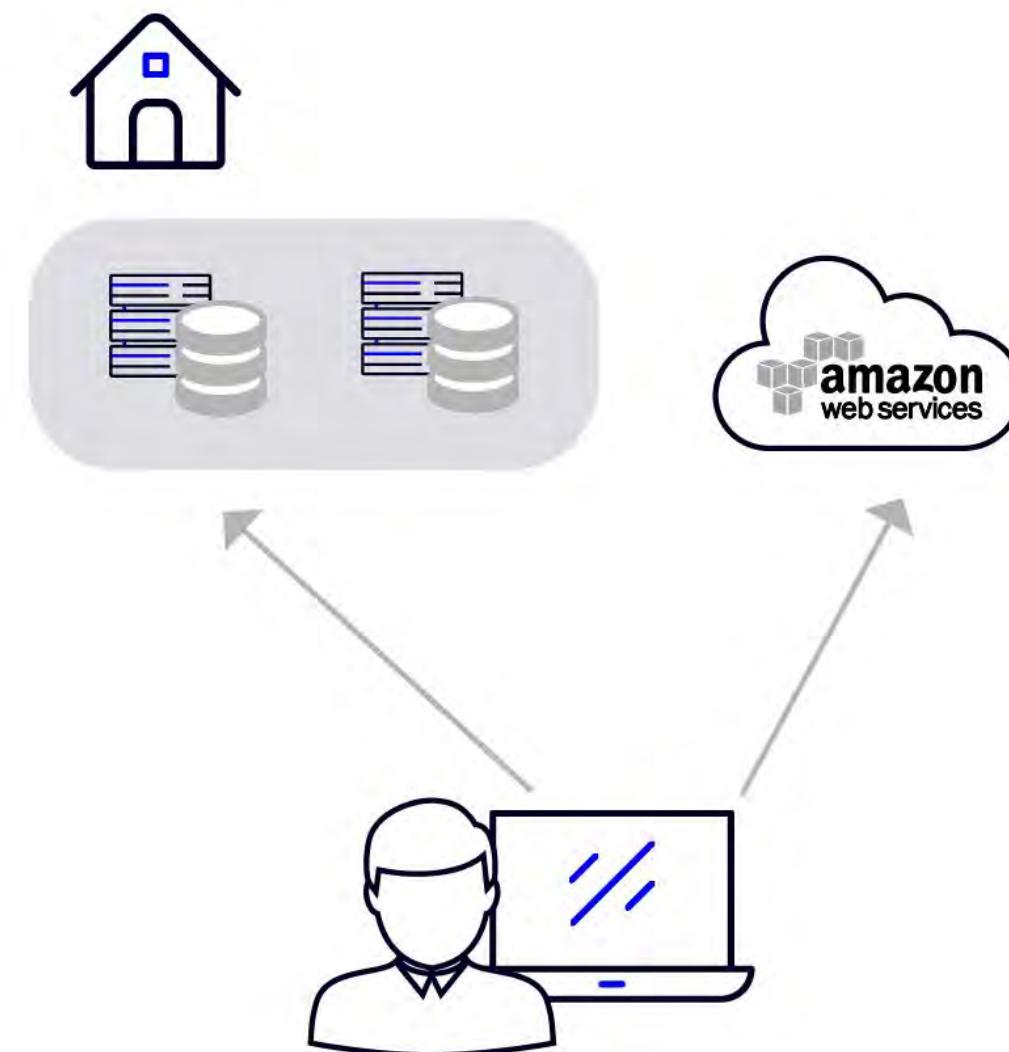
Los sistemas híbridos pueden tener todas las ventajas e inconvenientes de los otros dos entornos. Para gestionar esas desventajas, tendremos que **optimizar los servicios** que irán en la nube para maximizar las ventajas, y minimizar los costes. Lo mismo debemos hacer para las instalaciones on premise, 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.





HTTPS

¿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 encryptado 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**.



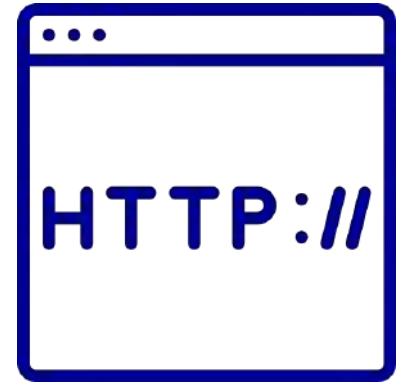
A TENER EN CUENTA

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.**

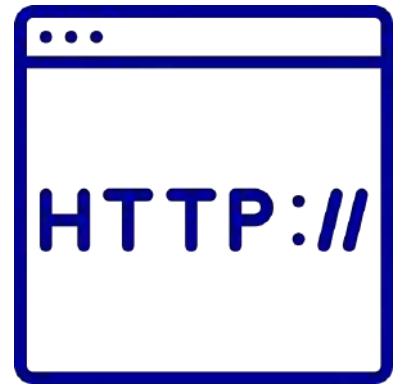


CARACTERÍSTICAS

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



¿Qué es?

HTTP es el **protocolo que utilizan los navegadores entre el cliente y el servidor**.

Especificamente 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/resuestas 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.



HTTP 2.0



¿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.

Cabeceras HTTP más comunes



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

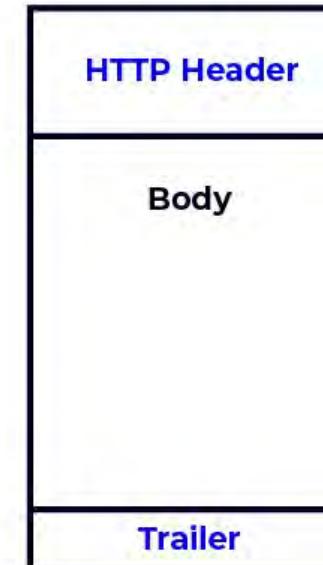
- **Cookie:** la cookie HTTP es un dato que permite al servidor **identificar las peticiones que viene de un mismo cliente**. HTTP es un protocolo sin estado. El servidor asigna una cookie a cada cliente y este las referencia usando la cabecera Cookie.
- **User-Agent:** identifica el tipo de aplicación, 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 Request or Response



HTTP Header





¿Para qué se utilizan?

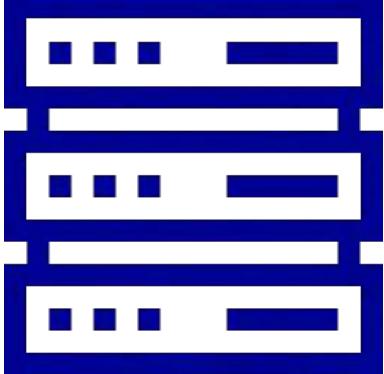
Las cabeceras de caché se utilizan para especificar *las políticas de almacenamiento de caché* que debe **usar el navegador tanto para las request como las response**. Con estas políticas podemos especificar tanto el cómo almacenar un recurso así como donde se almacena y su vigencia.



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 mientras se valida en origen.
 - **stale-if-error:** especifica tiempo extra de un recurso caducado sin validar.

Name		x Headers	Preview	Response	Initiator	Timing	Cookies
<input type="checkbox"/> 314309							
<input type="checkbox"/> 314309							
<input type="checkbox"/> 314309							
<input checked="" type="checkbox"/> 314309							
<input type="checkbox"/> 314309							
<input type="checkbox"/> 314309							
<input type="checkbox"/> 314309							
<input type="checkbox"/> login?service=http%3a%2f%2fcv.uoc.edu%2fUOC%2fmc-icons%2ffotos%2fobenjumea.jpg							
<input type="checkbox"/> jmurgui.jpg							
<input type="checkbox"/> login?service=http%3a%2f%2fcv.uoc.edu%2fUOC%2fmc-icons%2ffotos%2facastellanosr.jpg							
<input checked="" type="checkbox"/> restevan.jpg							
<input type="checkbox"/> login?service=http%3a%2f%2fcv.uoc.edu%2fUOC%2fmc-icons%2ffotos%2fjmurgui.jpg							
<input type="checkbox"/> acastellanosr.jpg?ticket=ST-395095-zzeBgQaQtg3ZSsyaeokr-cv.uoc.edu							
<input type="checkbox"/> obenjumea.jpg?ticket=ST-395094-0QrKcEZuj27C75fzzpPc-cv.uoc.edu							



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.

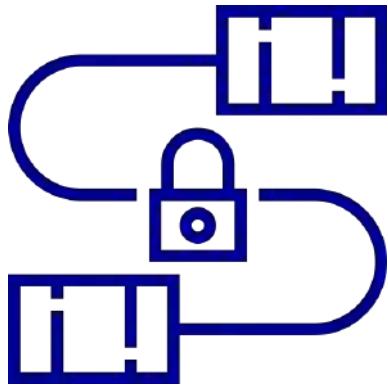
La primera se suele utilizar para guardar resultados de operaciones que son **poco pesados**. La segunda, en vez de utilizar la memoria de la misma máquina, se dedica un **conjunto** de máquinas explícitamente para este fin. Una de las **desventajas** de una caché distribuida es que su acceso es **más 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.





TLS

¿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.



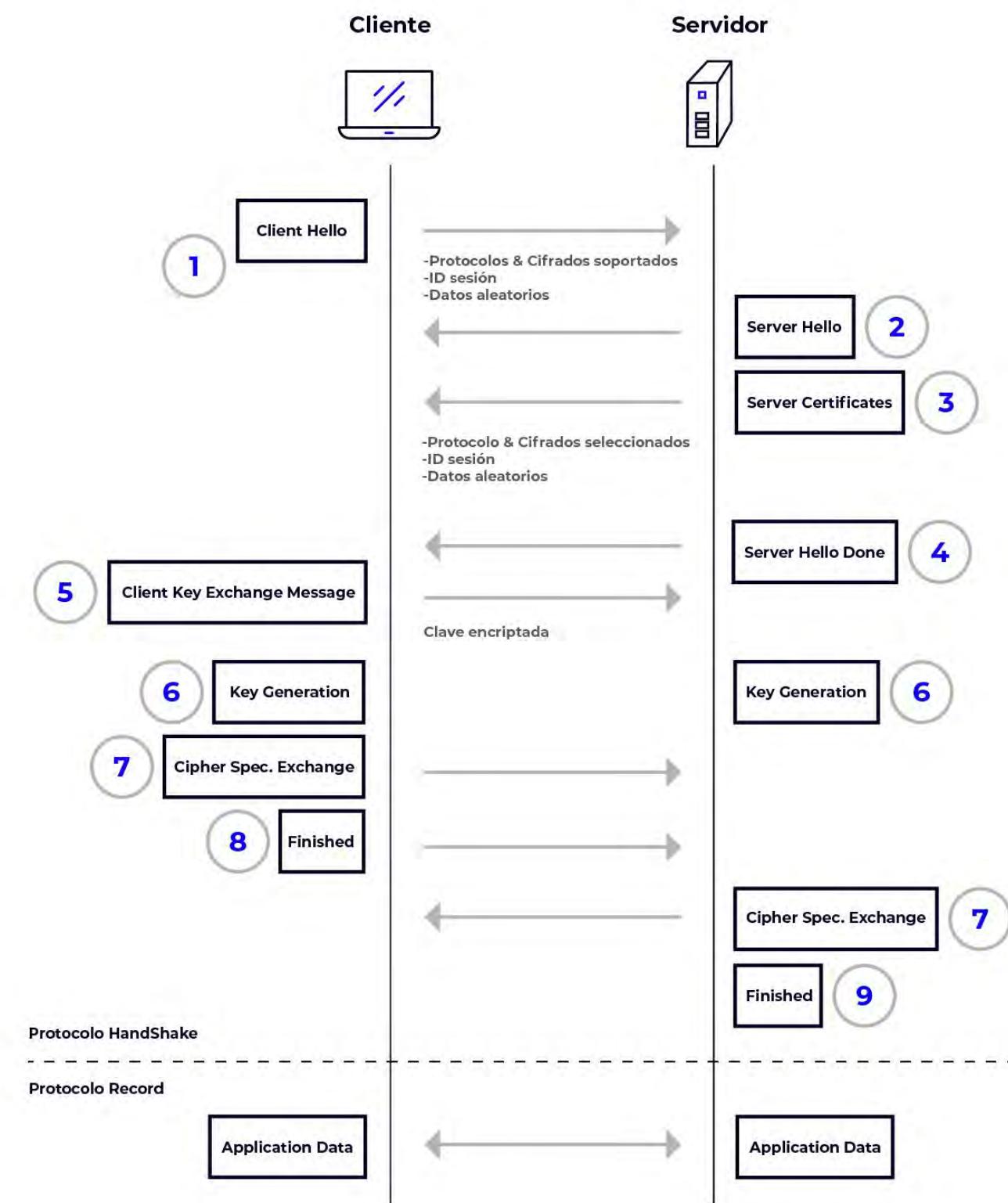
¿EN QUÉ CONSISTE?

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.





Cifrado simétrico

¿Qué es?

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



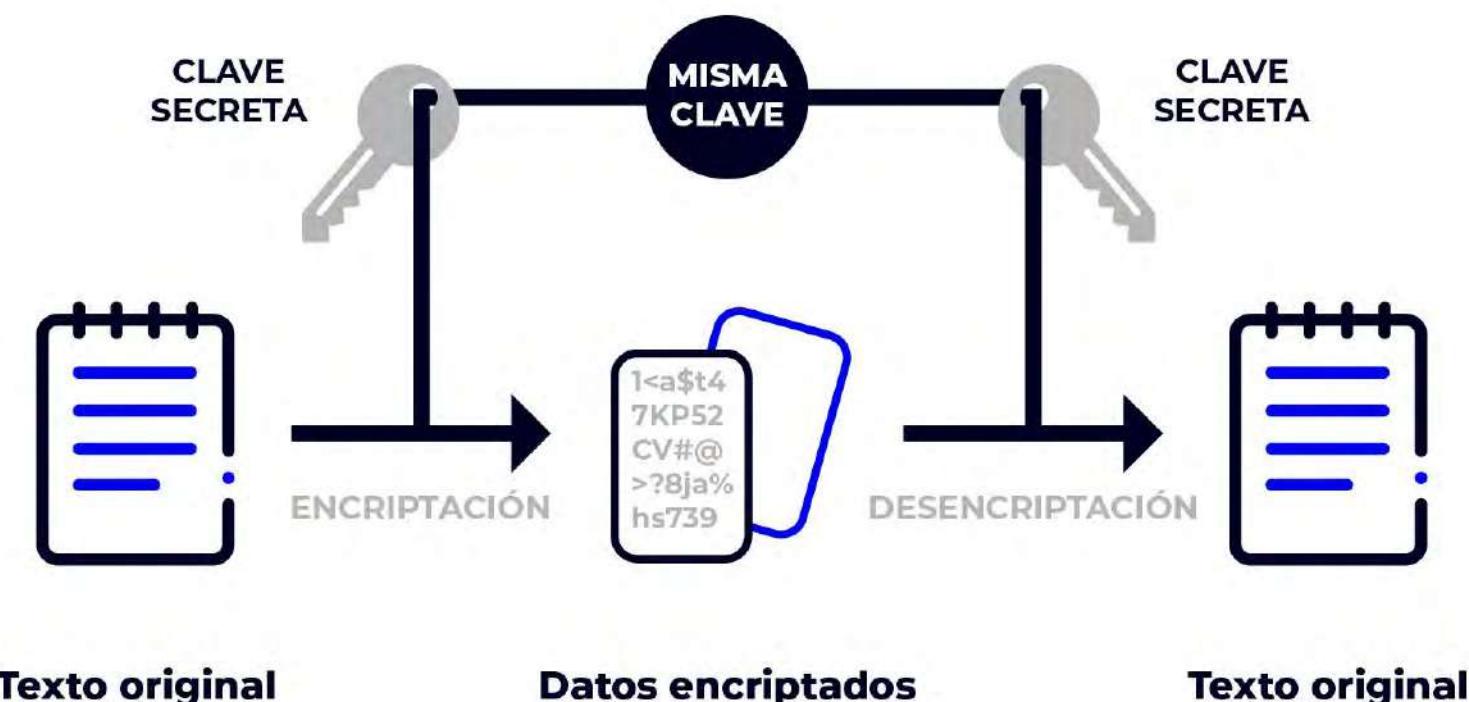
¿EN QUÉ CONSISTE?

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 aleatorios generada por un software especializado en generar este tipo de claves.

ENCRYPTACIÓN SIMÉTRICA





Certificado

¿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.

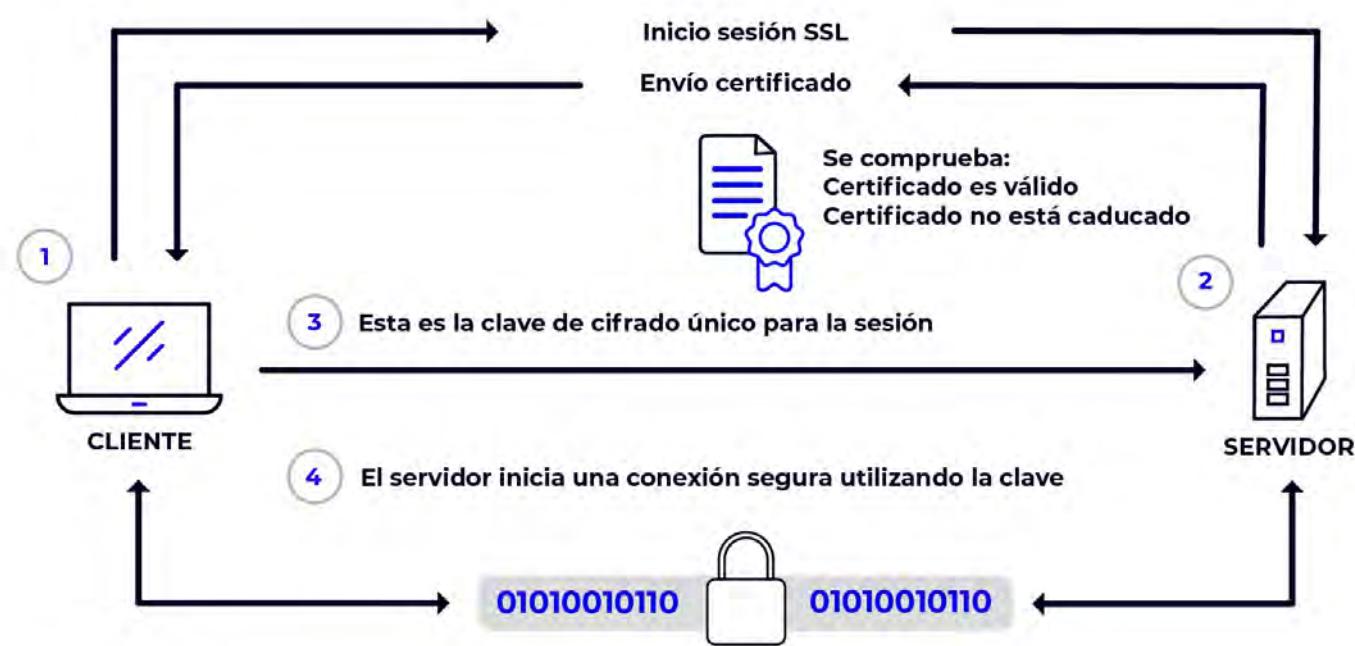
¿Qué ventajas nos ofrecen los certificados?

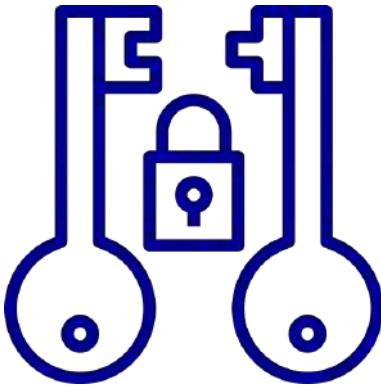
- Ahorro de espacio, tiempo y dinero.
- Confidencialidad y seguridad en las comunicaciones.



¿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

¿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:

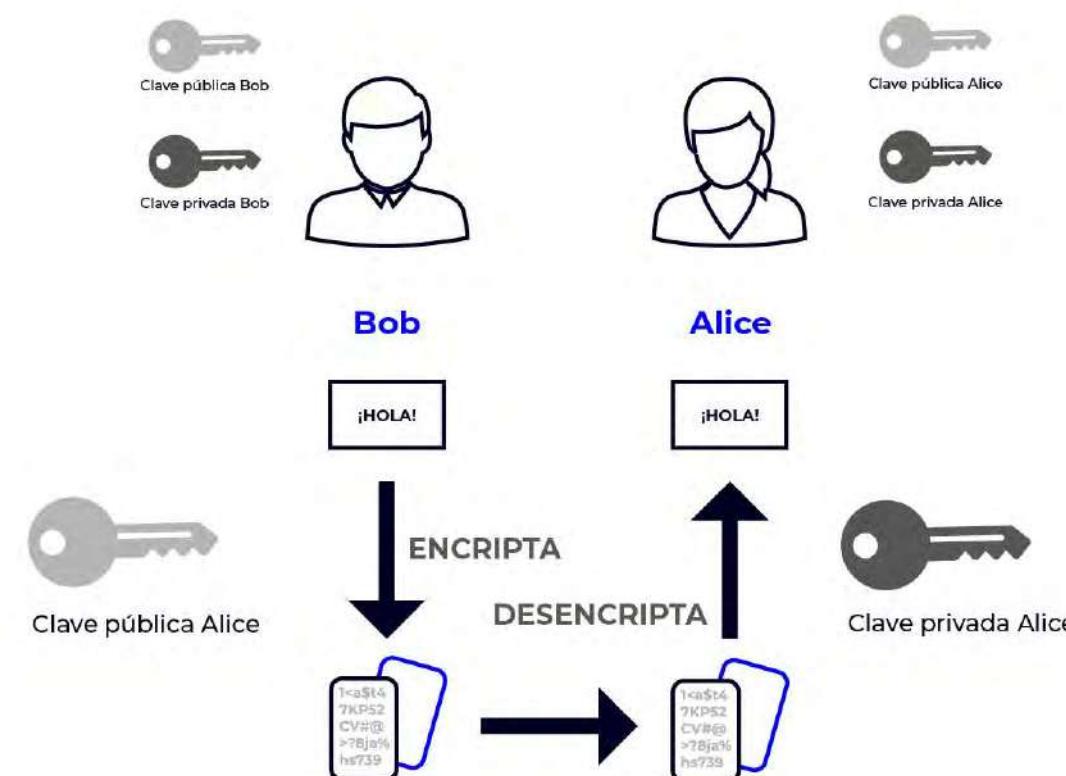
- Bob dispone de la clave pública de Alice, que sirve para encriptar (las claves públicas las puede ver cualquiera). **Encripta un mensaje con esa clave pública.**
- **Manda el mensaje encriptado a Alice.**
- Los mensajes que se hayan encriptado usando la clave pública de Alice sólo se pueden desencriptar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

ENCRIPCIÓN ASIMÉTRICA





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.

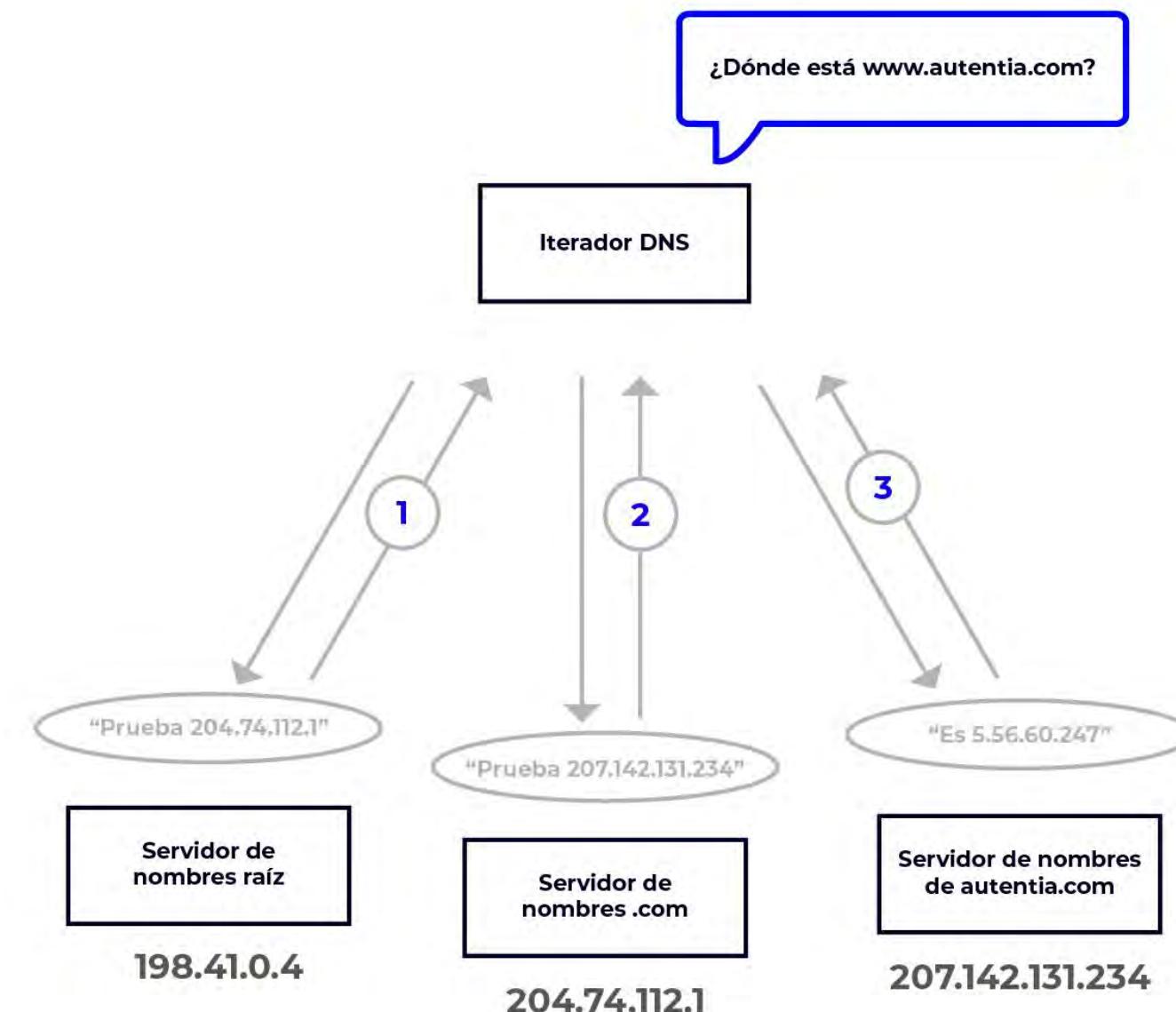
Cuando intentas acceder a www.autentia.com comienza un proceso de consultas iterativo para resolver la IP del sitio.

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.

¿Dónde está www.autentia.com?





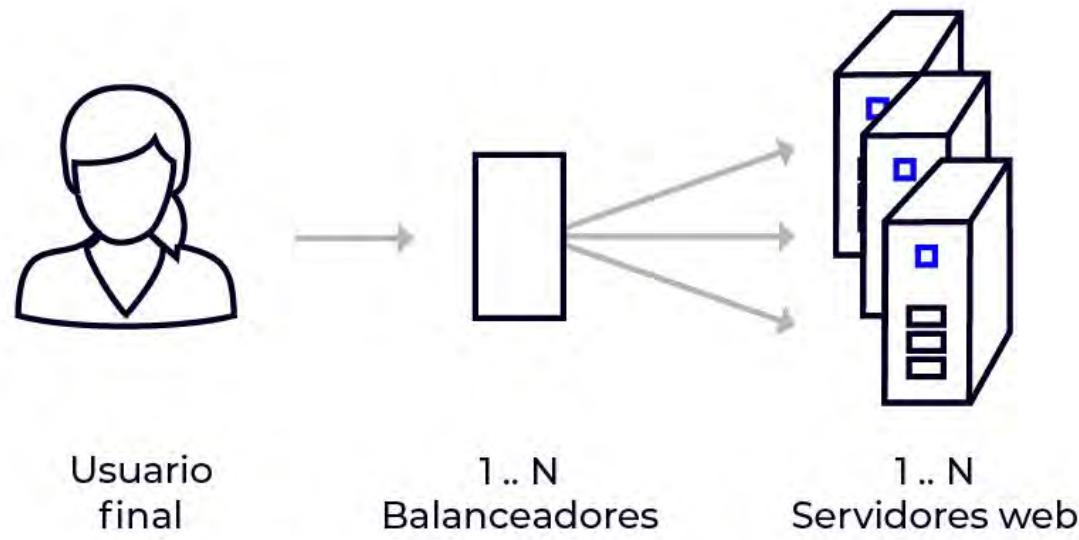
¿Qué son?

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.



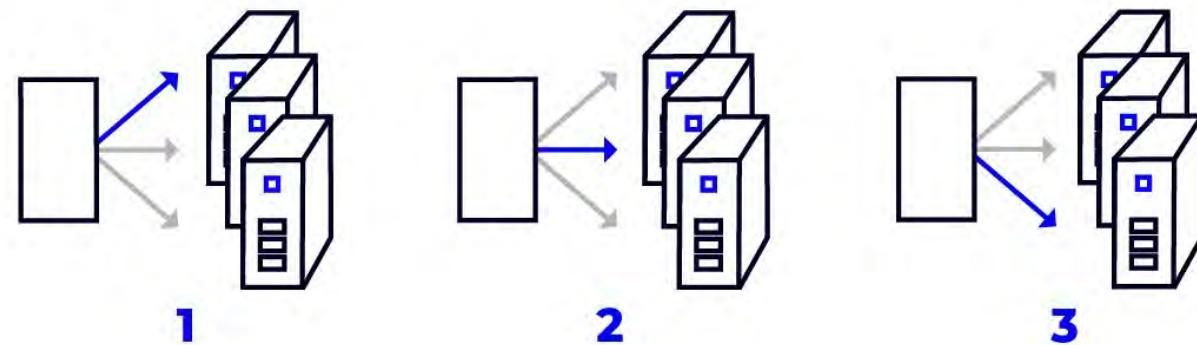
¿EN DÓNDE ENCAJAN?

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.

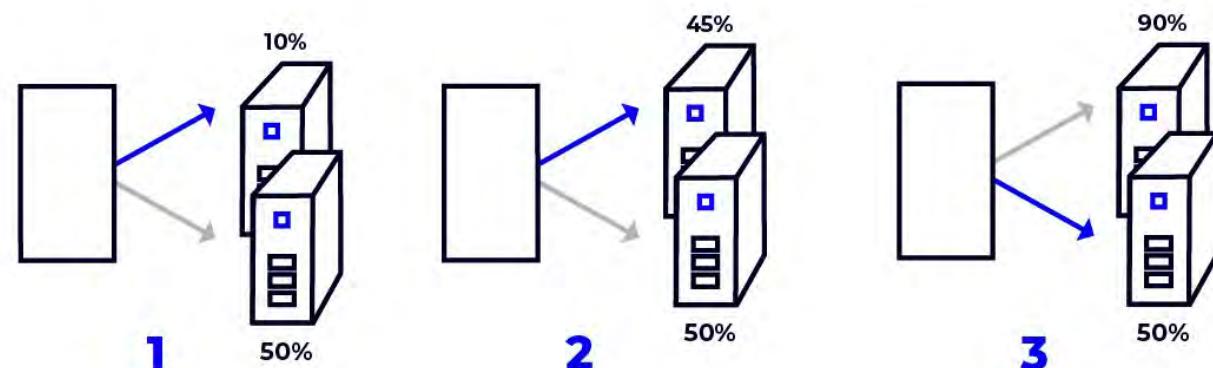


ALGORITMOS DE BALANCEO

- **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**.



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.



¿Qué es?

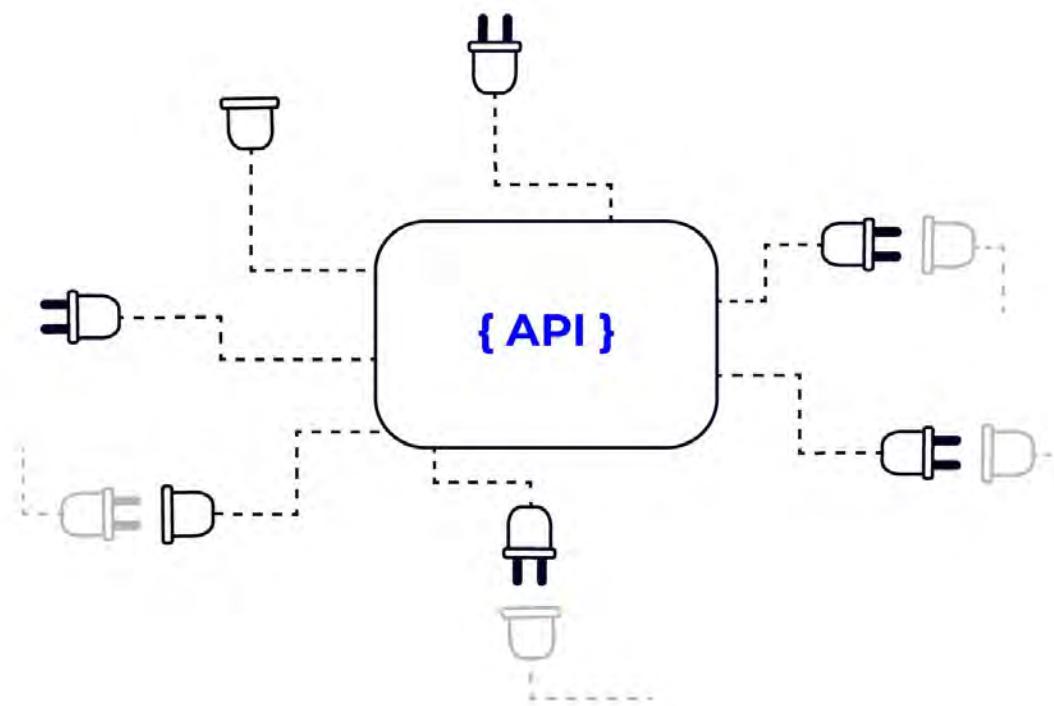
API es el acrónimo de Application Programming Interface, que **es un intermediario de software que permite que dos aplicaciones se comuniquen entre sí**. Es un conjunto de definiciones y protocolos para construir e integrar software. Cada vez que utilizas una aplicación como Facebook, envías un mensaje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

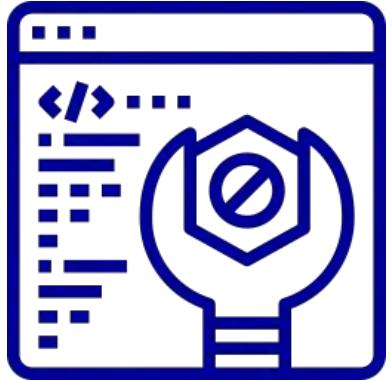
Una API web consta de uno o más endpoints expuestos públicamente de un sistema definido de mensajes de petición-respuesta, generalmente en formato JSON o XML que se expone a través de la web, más comúnmente por medio de un servidor HTTP.



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**.

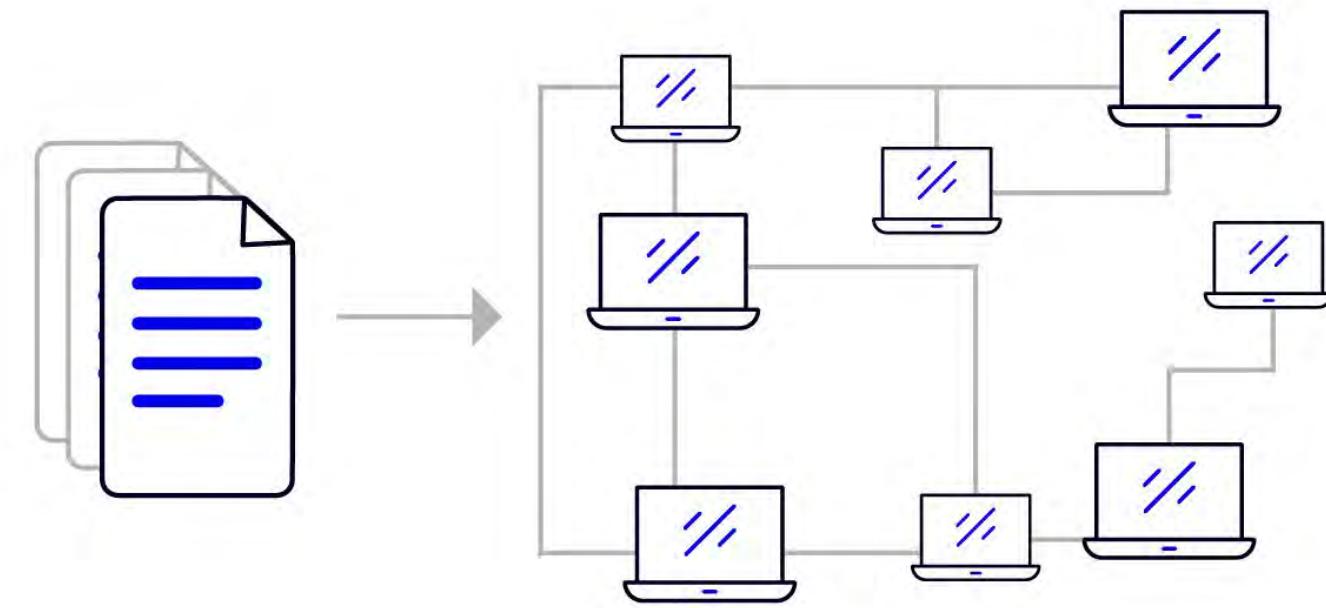


CONCEPTO

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.

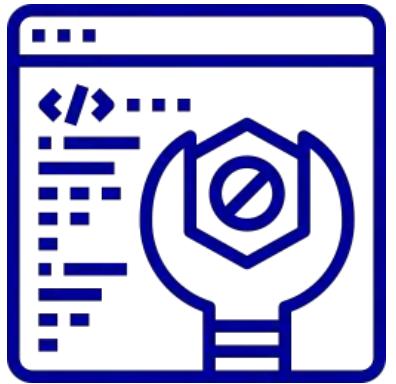
Con IaC los equipos hacen cambios en los entornos en un archivo de configuración, que normalmente tiene formato JSON, YAML o similar. Cuando se registra este cambio en el control de versiones hay un sistema de integración que genera los entornos tal y como se describen en el archivo del modelo. En definitiva,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

Configuración como código



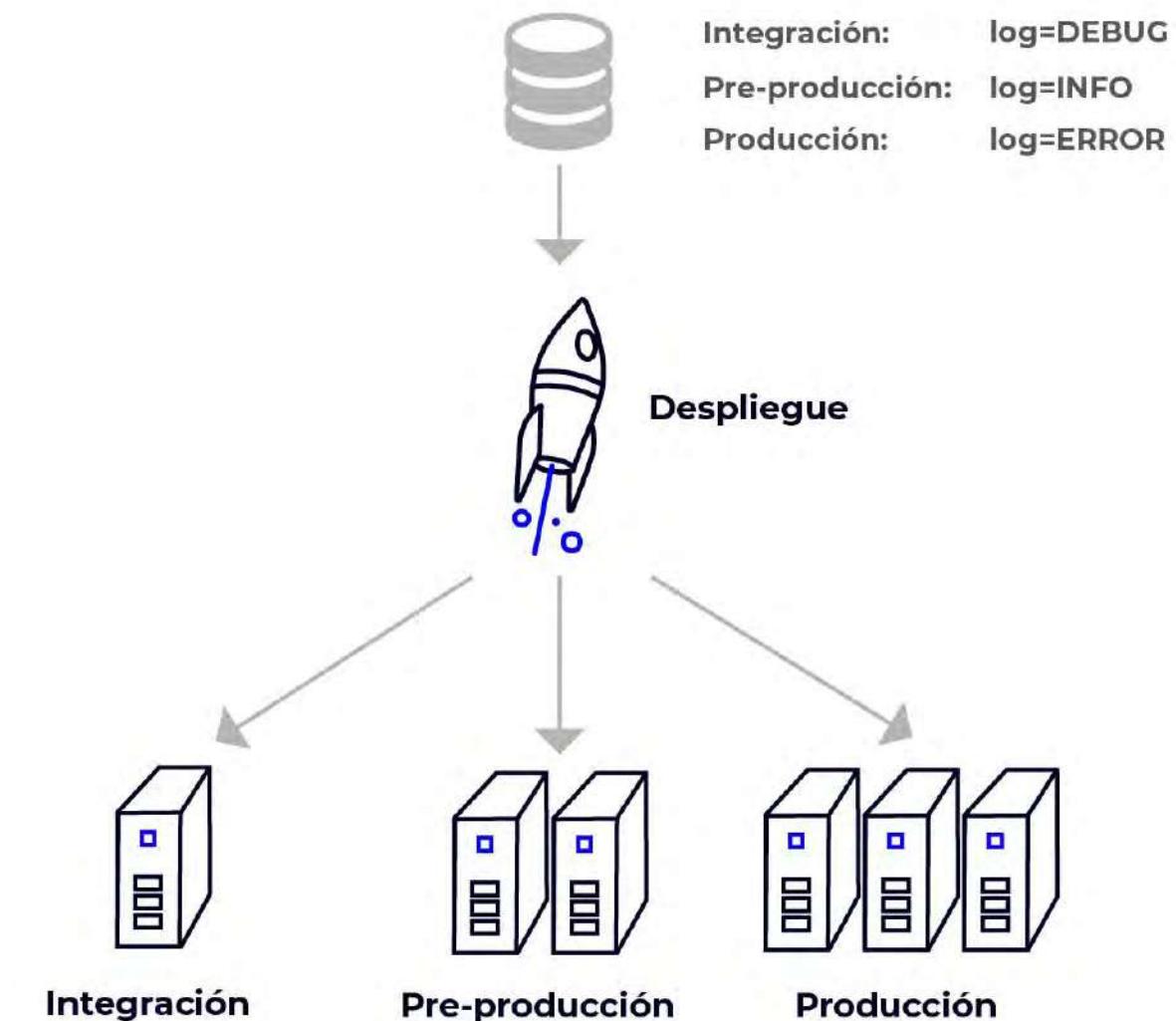
¿De qué se trata?

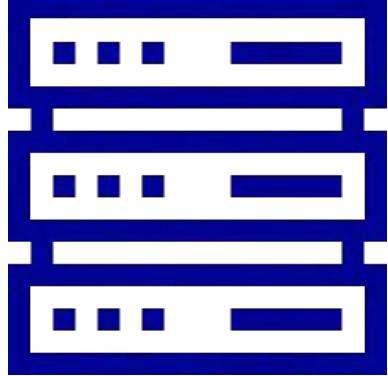
La configuración como código (o CaC, Configuration as Code, en sus siglas en inglés) es una práctica que involucra el versionado de los parámetros que configuran nuestras aplicaciones en distintos entornos.



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**.

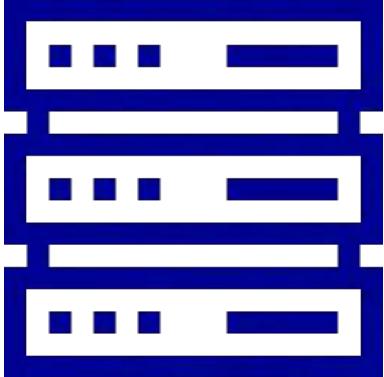


DESVENTAJAS

- 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 | • CouchDB | • Hbase | • BigTable |
| • Neo4j | • Postgres | • MongoDB | • Cassandra |



¿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

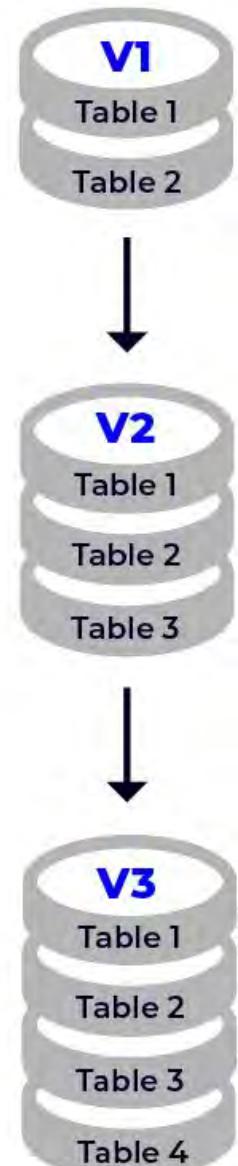
Flyway usa prefijos para conocer el orden en el que se ejecutarán los scripts, seguido de dos guiones bajos y una descripción de lo que se está haciendo.

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

¿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?

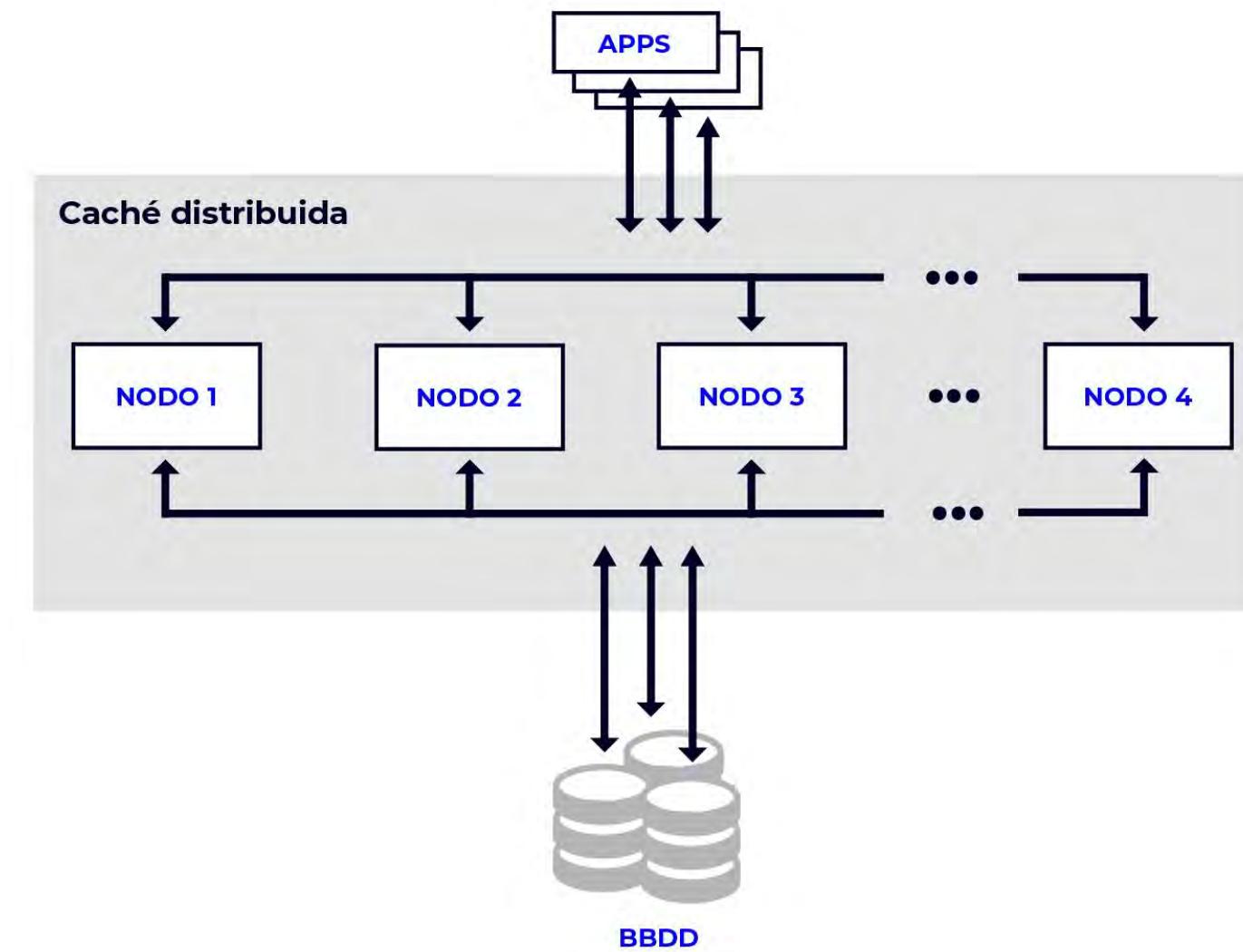
Normalmente una caché tradicional se encuentra en un único servidor físico o componente hardware y esto limita su uso en un sistema distribuido.

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.





¿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).



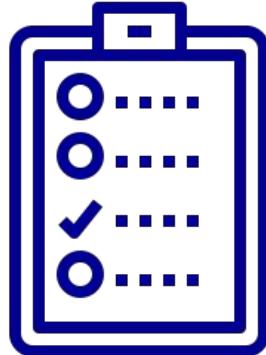
VENTAJAS

- **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.



¿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 **throughput** (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.

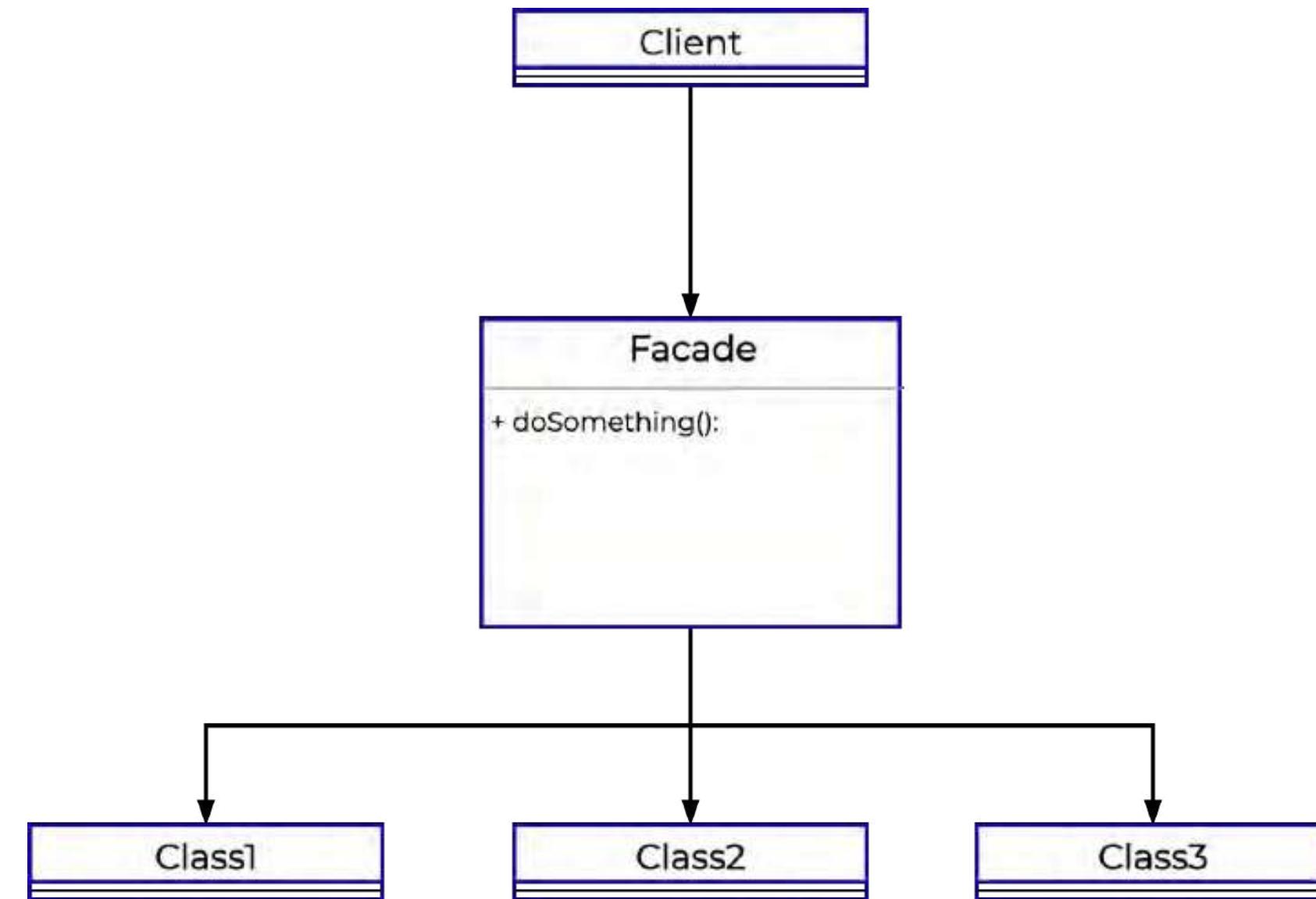


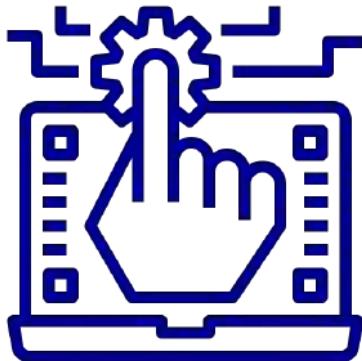
RAZONAMIENTO

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.





Site Reliability Engineering (SRE)

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 reanudarlos.



**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** <div class="relevant"> y el selector .relevant {}.
- Selectores de **id** <div id="cabecera"> 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 (:).



¿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. Pertece 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.

Seguridad y Privacidad	Usabilidad y Accesibilidad	Ciclo de Vida de la Aplicación	Servicios Comunes
Identidad, cifrado del API, múltiples factores de autenticación	Contenido y Software Accessible, Internacionalización	Modo "Sin conexión", despliegues, geoposicionamiento, sincronización	Social, pagos, anotaciones, red de datos
Rendimiento y Optimización	Medios y Comunicación en tiempo real	Interacción con el dispositivo	Diseño y Desarrollo Web Esencial
Perfilado, mejoras, diseño flexible	WebRTC, transmisión de medios, multipantalla	Sensores, orientación, vibración, pantallas táctiles, bluetooth, etc.	Composición, estilo, HTML, animaciones, tipografía



HTML

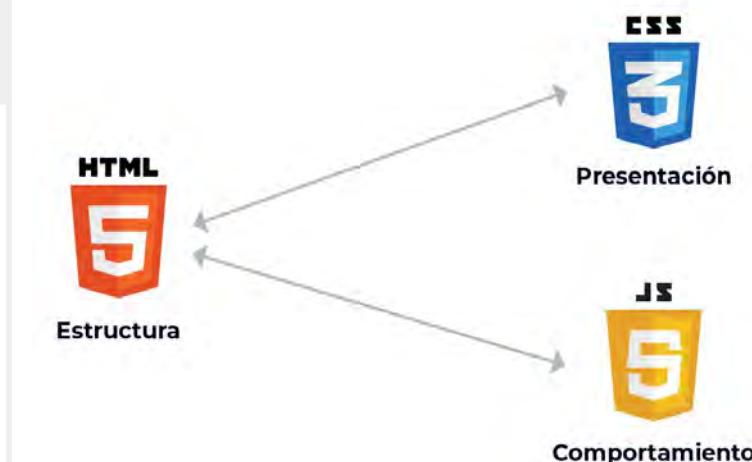
¿Qué es?

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

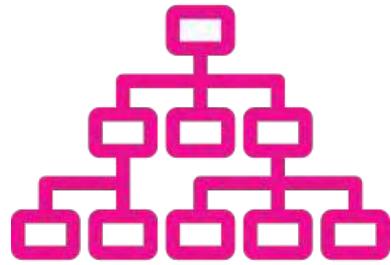
Ventajas:

- 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 una persona, CSS como la piel, pelo, etc. y JavaScript los músculos.



¿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.

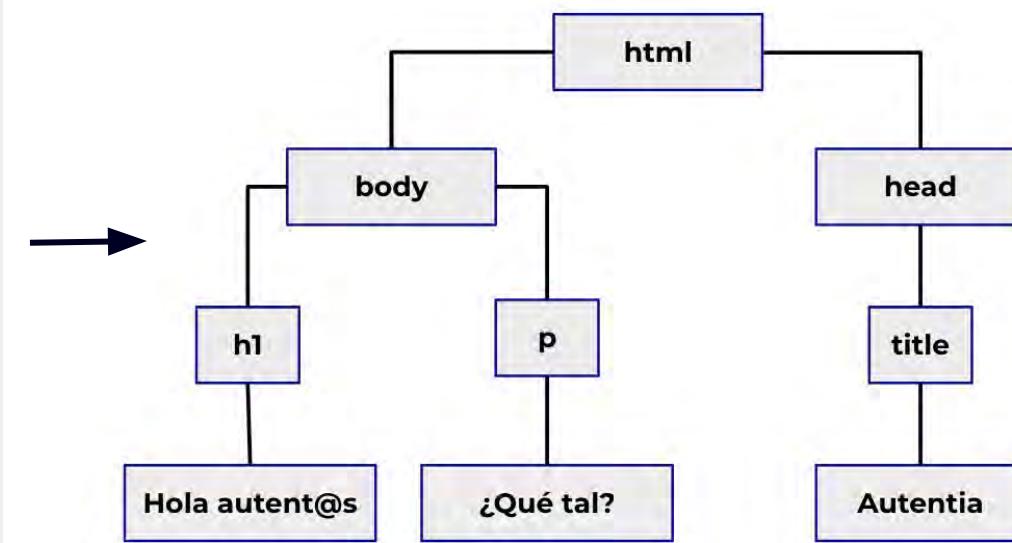
Tenemos funciones como *document.createElement*, *getElementbyid*, *window.alert*, entre otras muchas.



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.

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





¿Qué es?

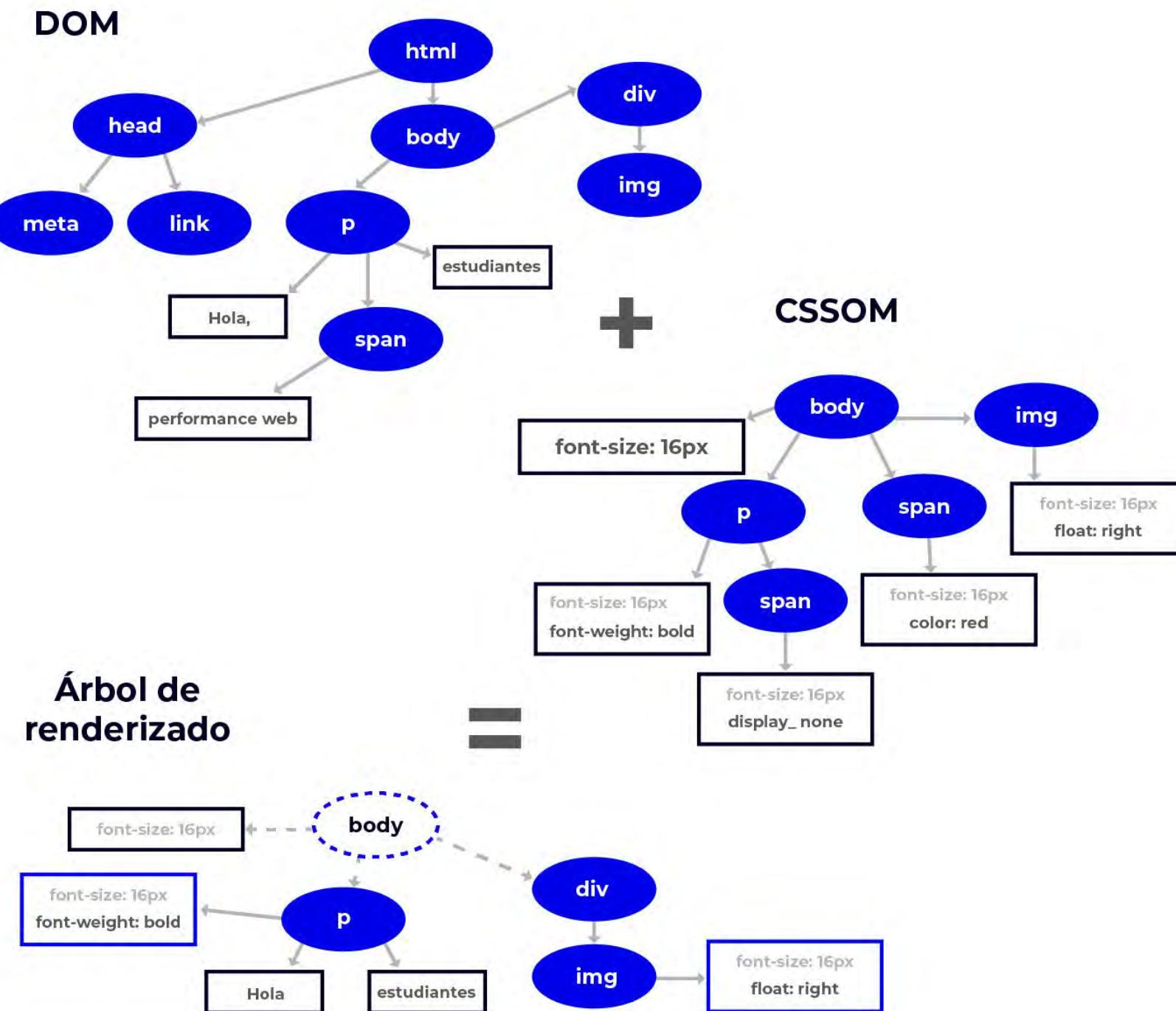
También referido como **Critical Rendering Path** (ruta de renderizado crítica) es la **secuencia de pasos** que sigue el navegador **para convertir HTML, CSS y JavaScript en píxeles en la pantalla**. Esta secuencia de pasos es realizada por el motor de renderizado del navegador.



¿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 básicamente 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
```

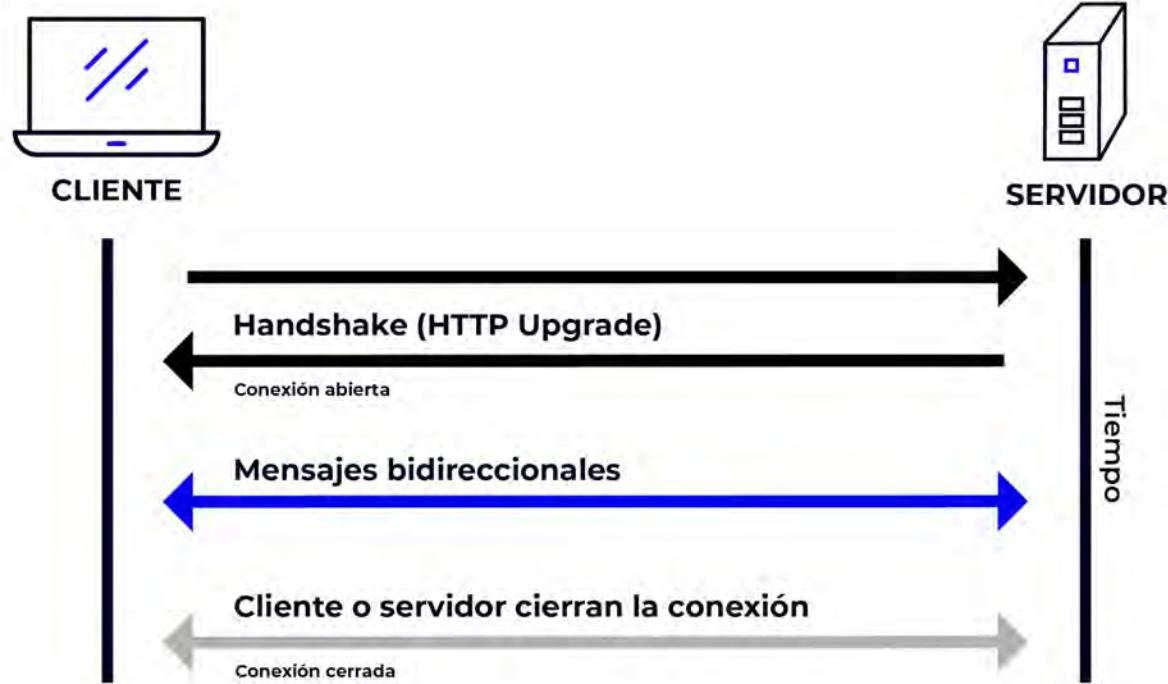
Si el servidor soporta el protocolo websocket, responderá con la cabecera **Upgrade** y de inmediato se podrá enviar o recibir datos de forma bidireccional.

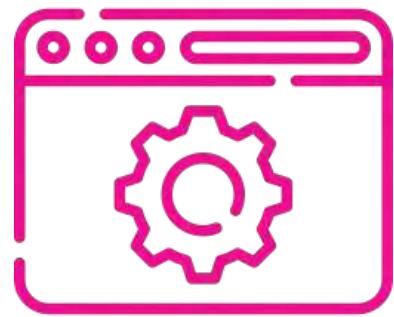
```
HTTP/1.1 101 WebSocket Protocol Handshake
Date: Mon, 15 Jun 2020 10:07:34 GMT
Connection: Upgrade
Upgrade: WebSocket
```



¿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

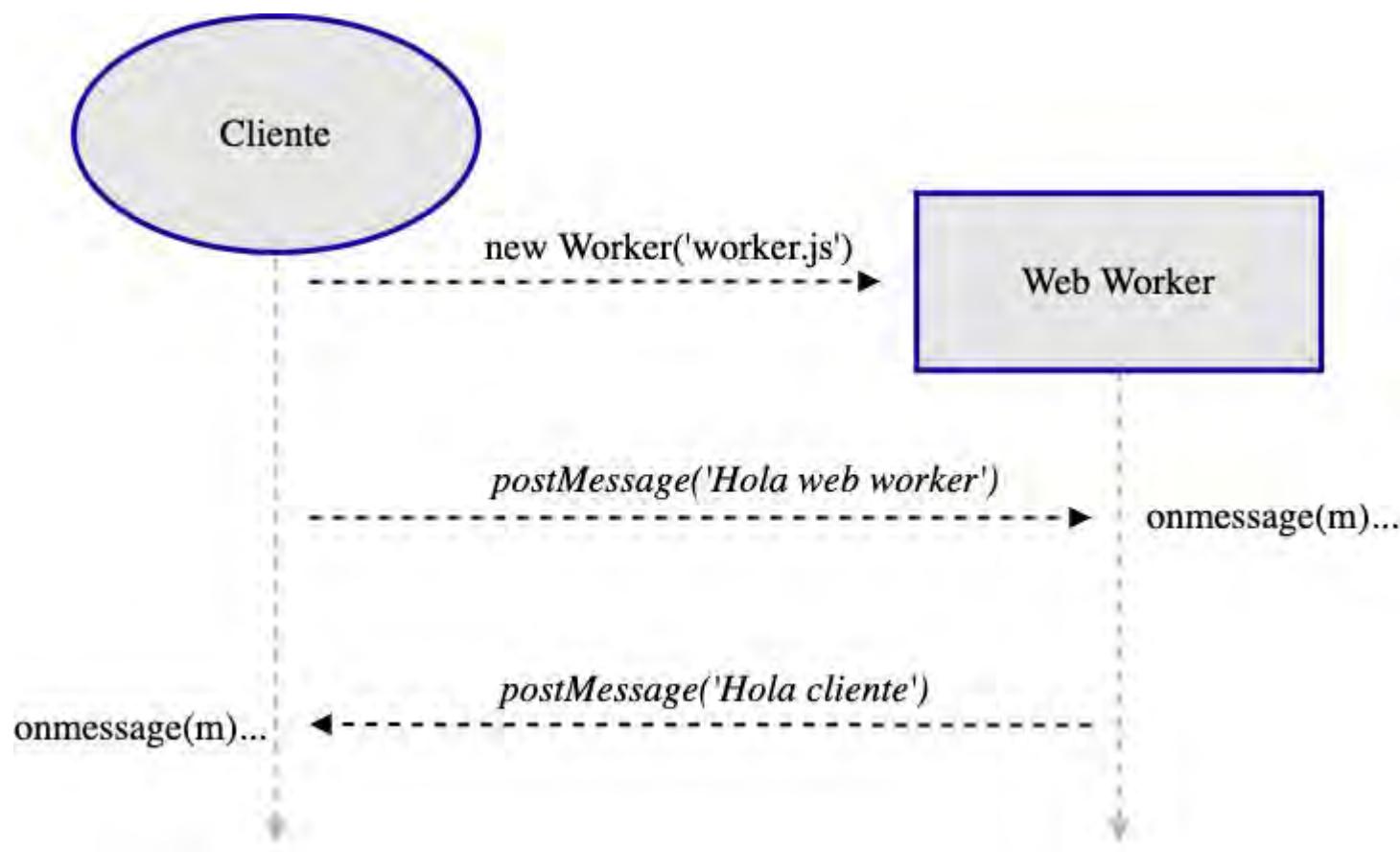
Una vez inicializado, tanto el *worker* como su cliente se comunicarán a través del método *postMessage* y el manejador de eventos *onmessage*.

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).

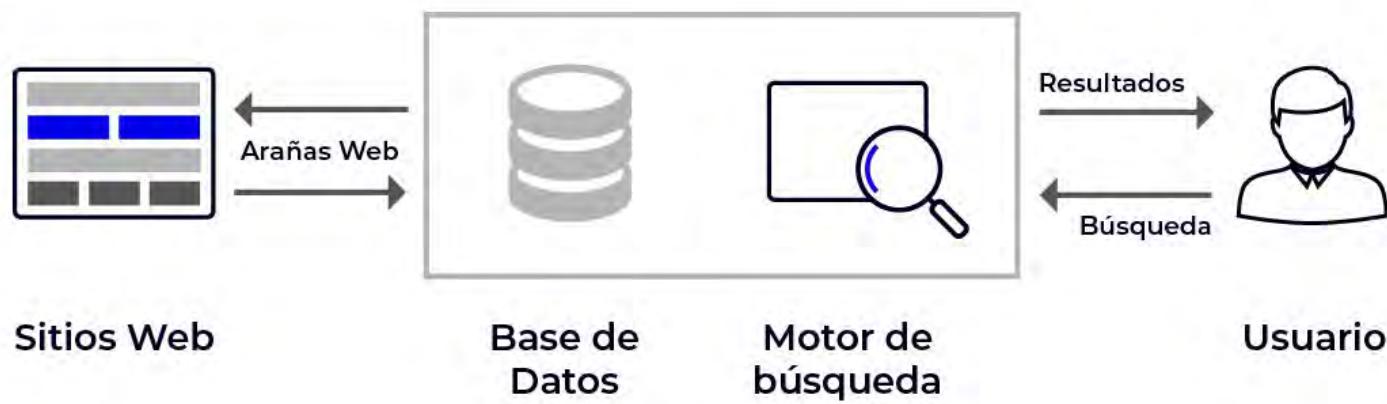


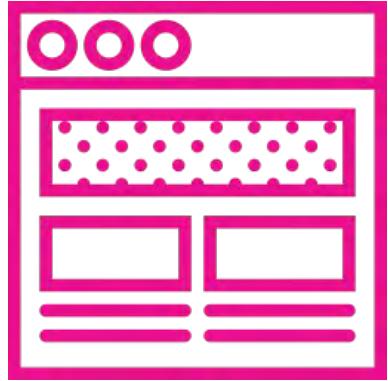
¿CÓMO FUNCIONAN LOS BUSCADORES?

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 sobreescibir.

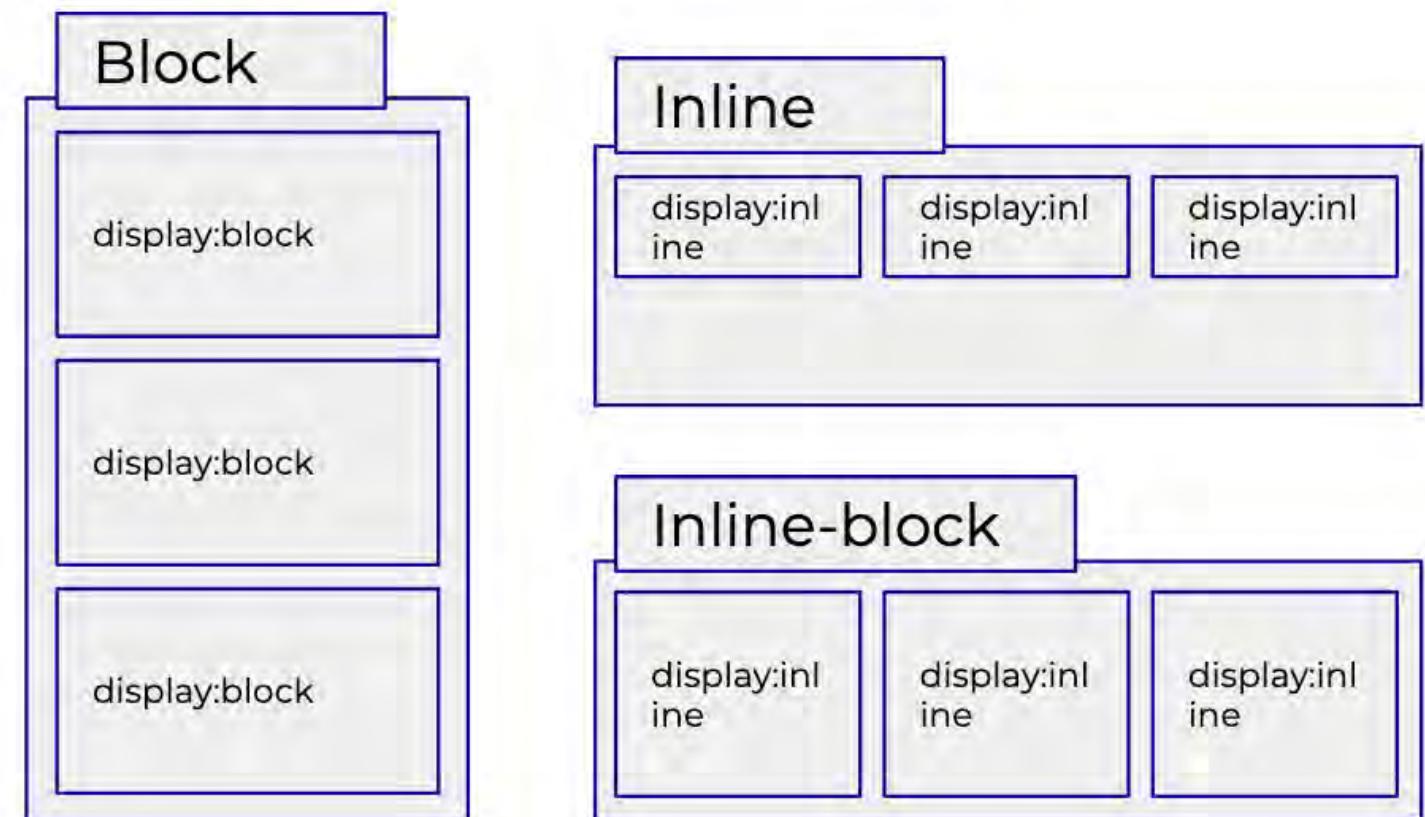


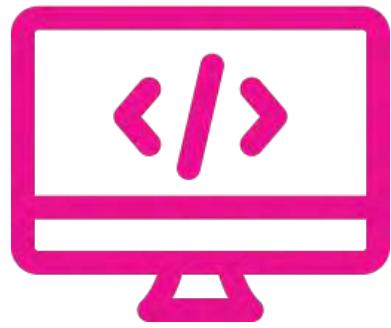
DIFERENCIAS

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

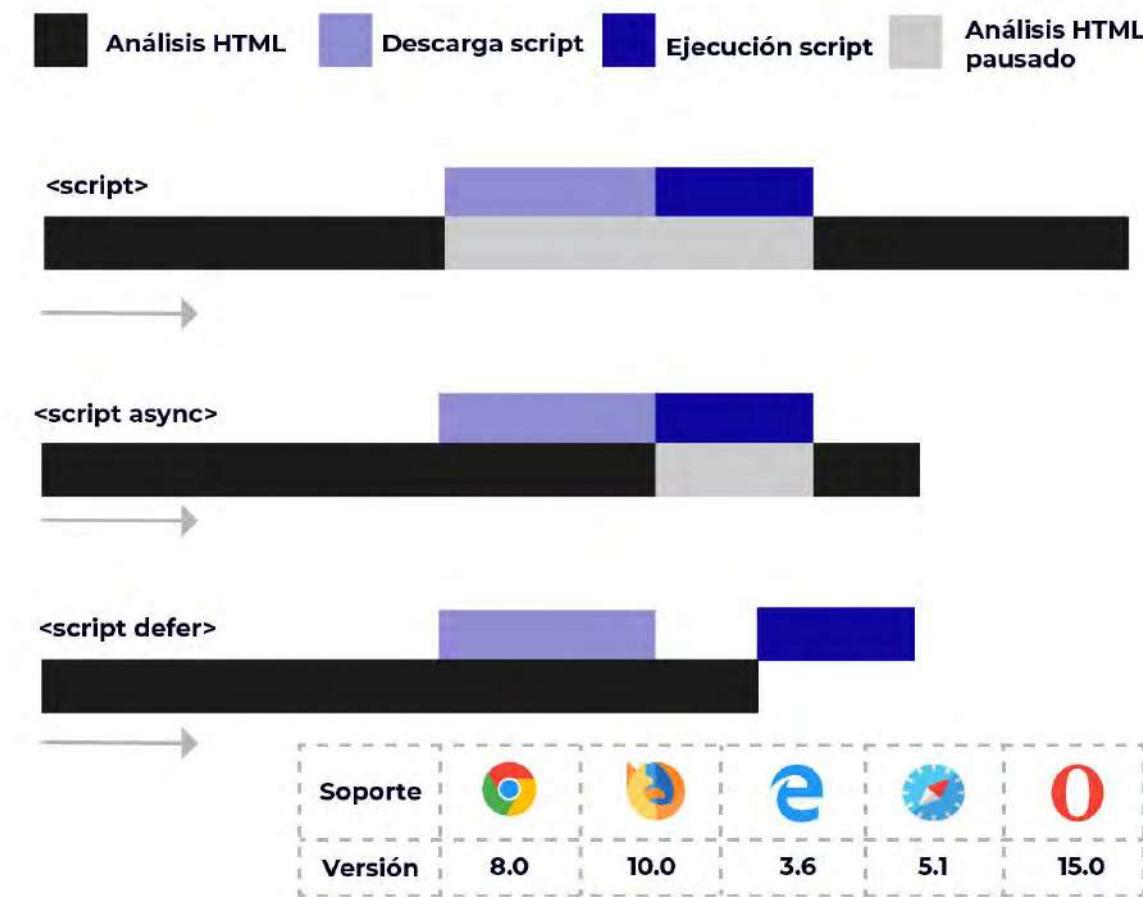
Antes de la aparición de estos atributos, se recomendaba colocar los elementos <script> al final del HTML, para que cuando el analizador se los encontrase, ya hubiese analizado y renderizado todo el documento. Los atributos **async** y **defer** nos ofrecen **una solución a este problema, sin forzarnos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



Compatibilidad entre navegadores



¿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

¿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.

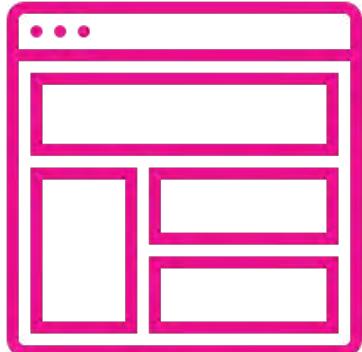
No se recomienda utilizar unidades absolutas si queremos un diseño **responsive**, ya que al ser unidades fijas, no se adaptan de igual manera a todas las pantallas.



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.



¿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**).

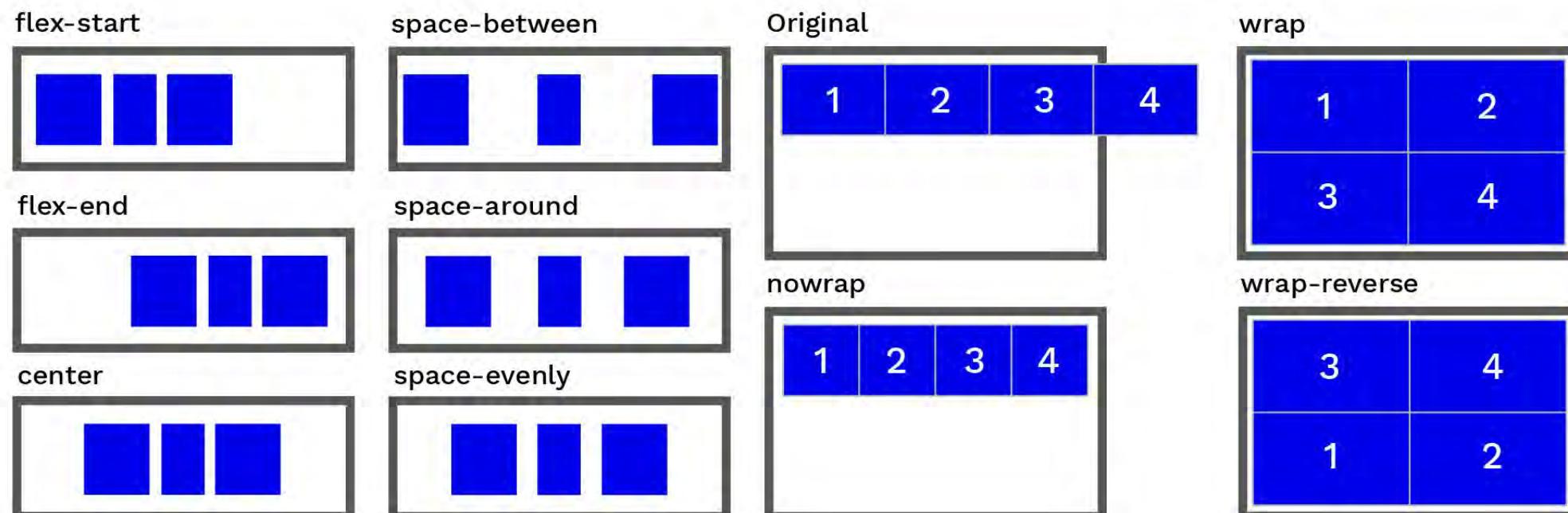


PROPIEDADES MÁS USADAS

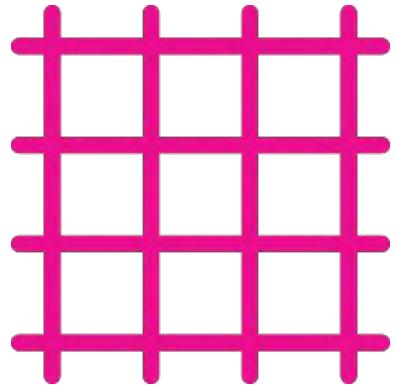
Para que un contenedor sea flexible, se usa la propiedad **display: flex**.

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 abajo, 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



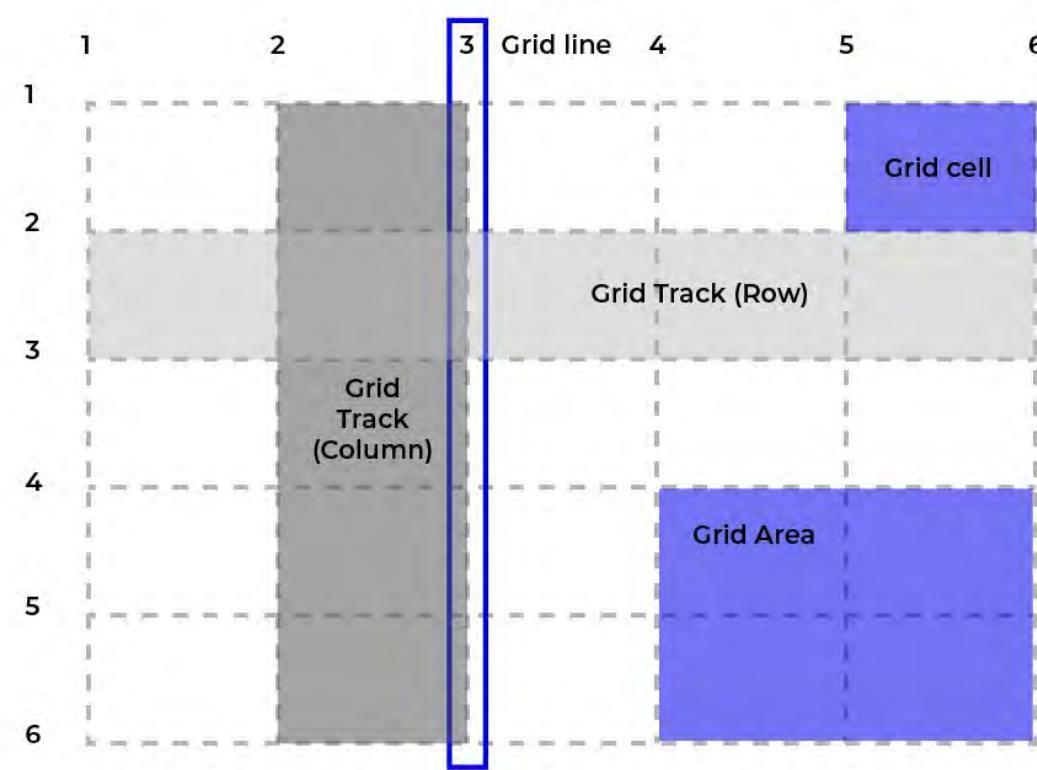
¿Qué es?

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.



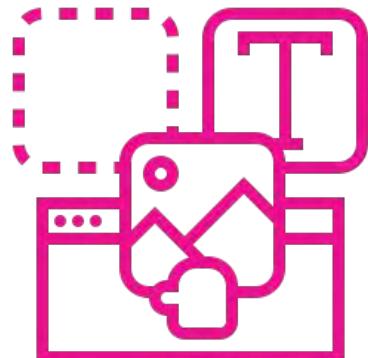
CONCEPTOS BÁSICOS

- **Grid Layout se compone de líneas** 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**.



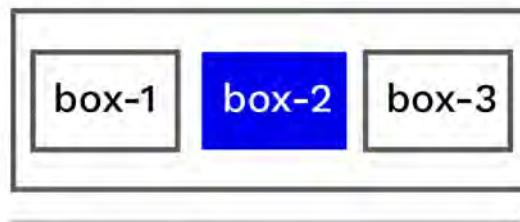
¿En qué consiste?

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*.

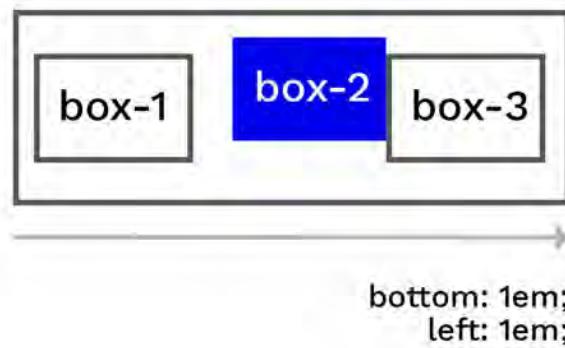


TIPOS

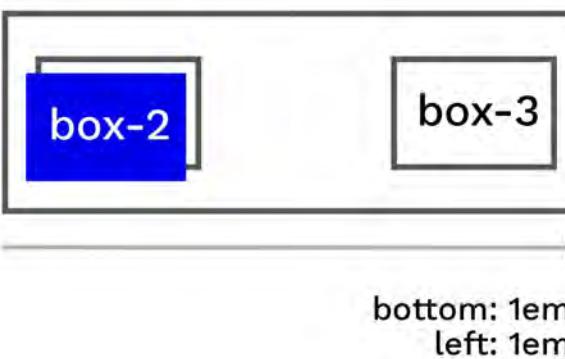
- **STATIC:** valor por defecto. Posiciona los elementos de acuerdo al flujo normal del documento y los elementos a su alrededor. Las propiedades de desplazamiento **no tienen efecto**.



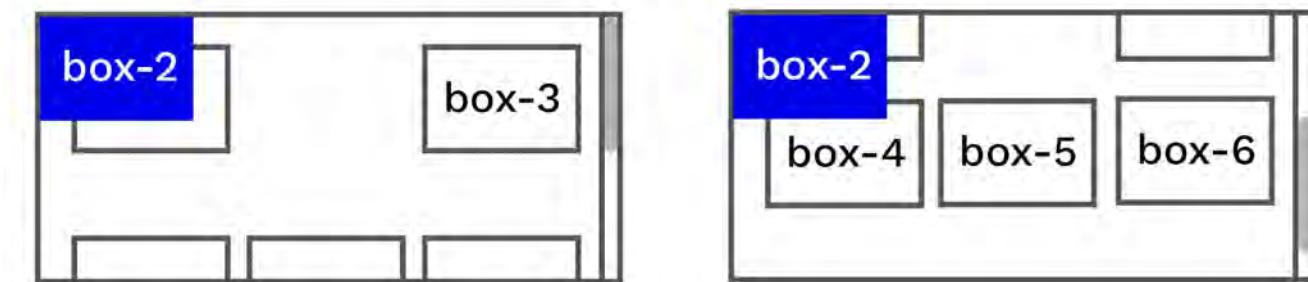
- **RELATIVE:** el elemento se posiciona como si fuese *static*, pero las distancias definidas en *top*, *right*, *bottom* y *left* desplazarán el elemento desde ese lado.



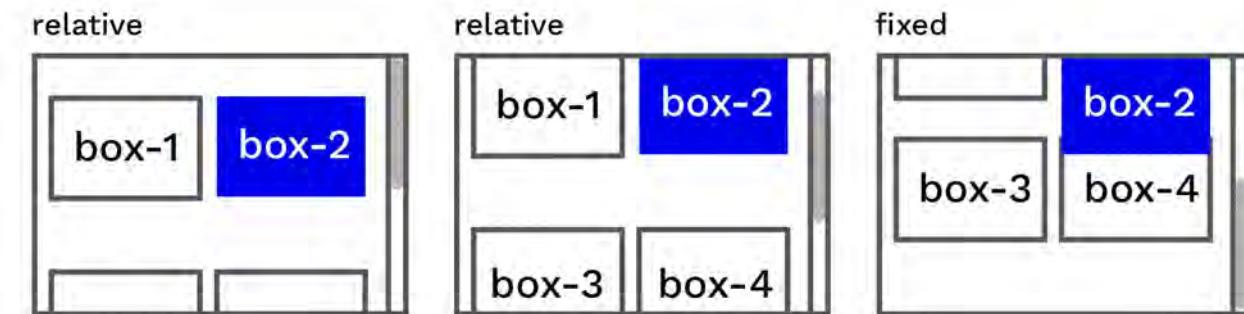
- **ABSOLUTE:** la posición del elemento se calcula con respecto al ancestro **posicionado** más cercano, o en su defecto al cuerpo del documento.



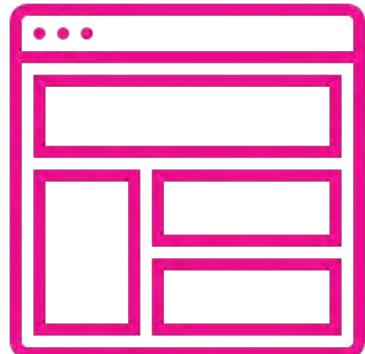
- **FIXED:** posiciona el elemento dentro del *viewport* inicial, fijándose en un sitio independientemente de la posición de los demás elementos.



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



Diseño fluido



¿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.

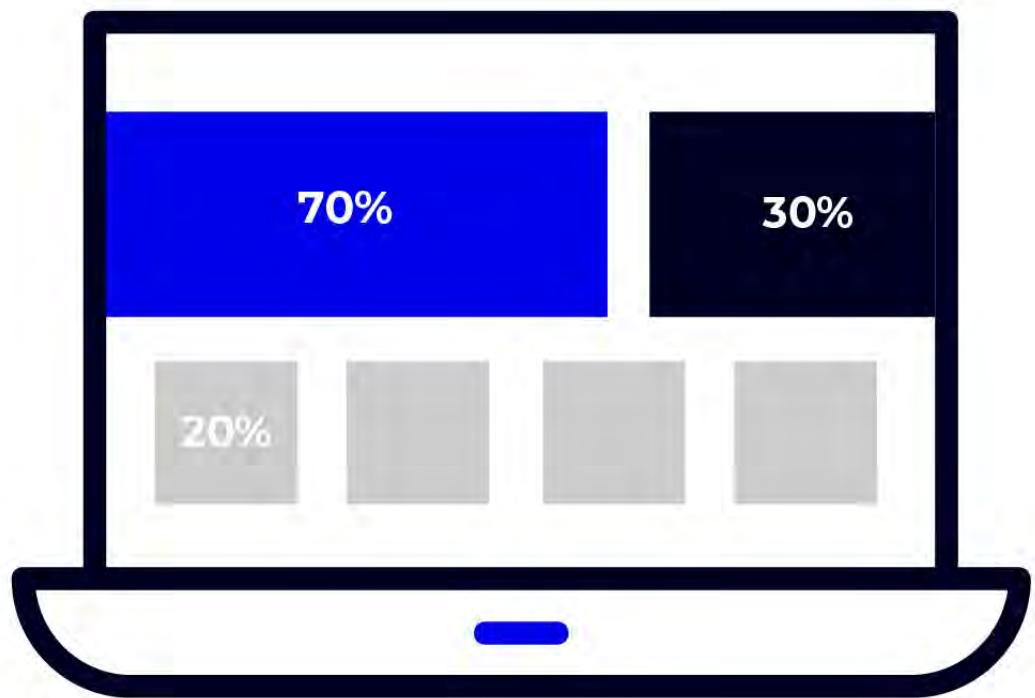
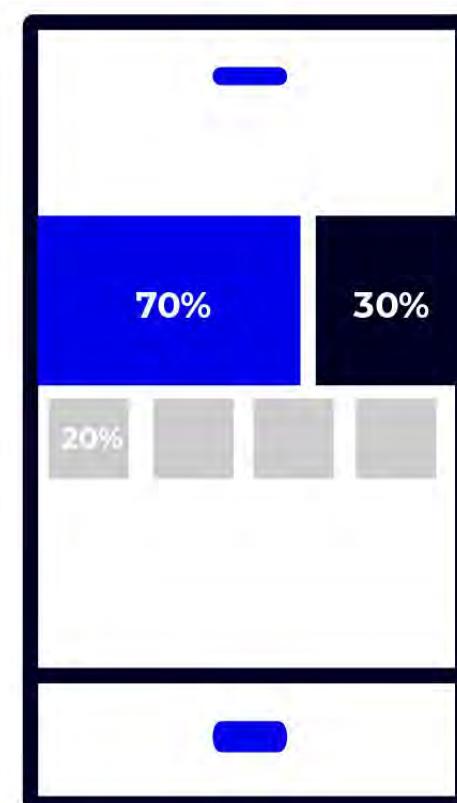


¿EN QUÉ CONSISTE?

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.





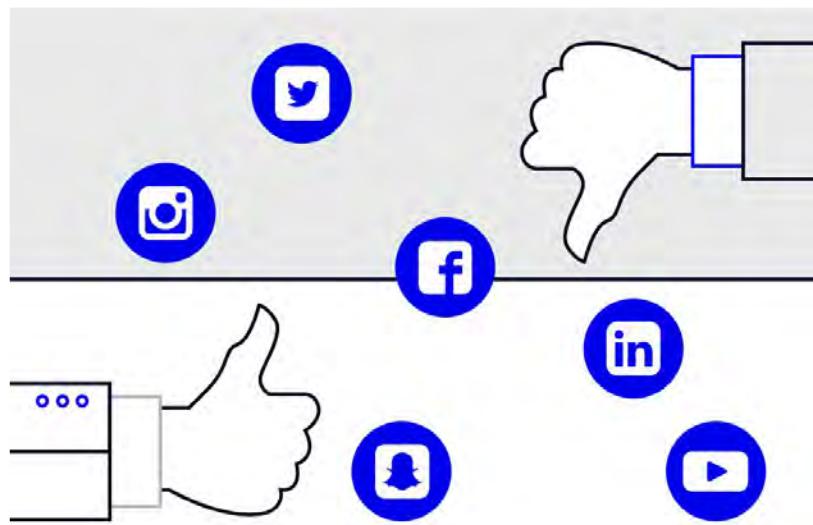
¿Qué es?

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.

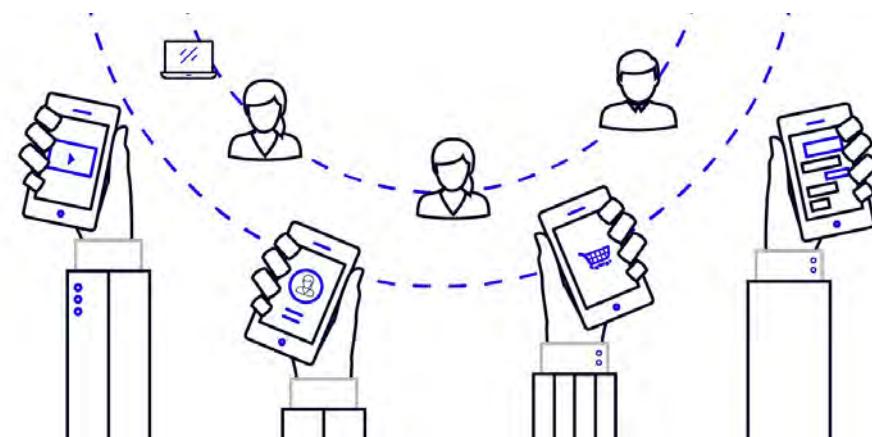


¿POR QUÉ LO NECESITAS?

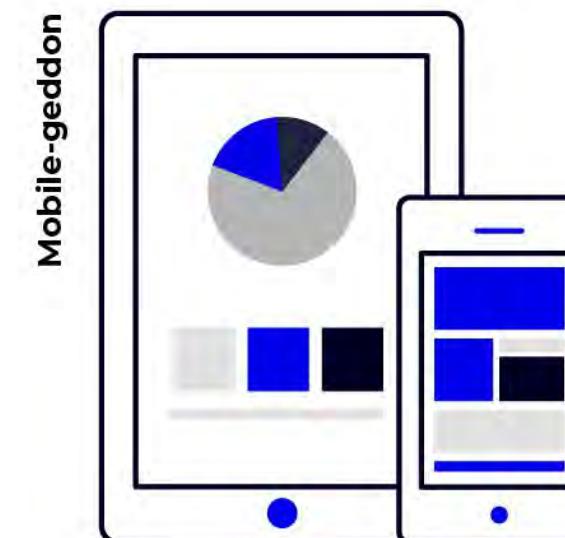
- **Mejorar la experiencia del usuario.** Según Google, los usuarios de móvil tienen una tasa de rebote del 61% frente al 67% de conversión, si tienen una buena experiencia.



- **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.

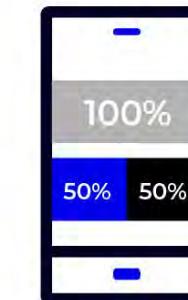


TÉCNICAS BÁSICAS

- **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**.



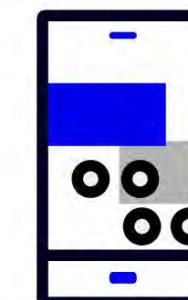
Unidades relativas



Unidades Estáticas

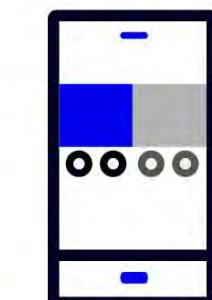


Con Breakpoints

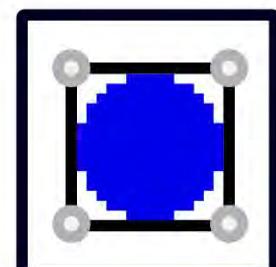
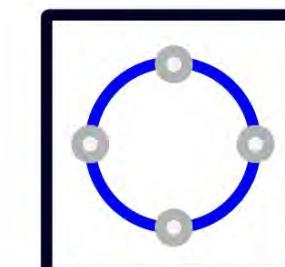


Vectores

Sin Breakpoints



Imágenes





Definición

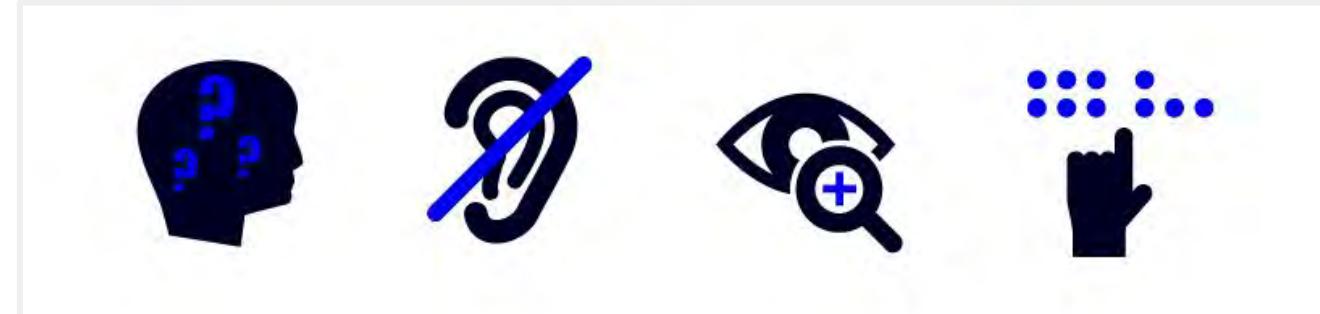
La accesibilidad web significa que **personas con algún tipo de discapacidad podrán hacer uso de la Web** gracias a un diseño que va a permitir que estas personas puedan percibir, entender, navegar e interactuar con ella.



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.

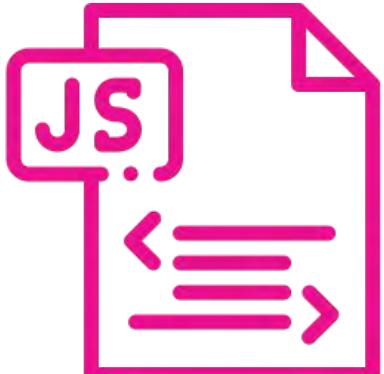


TÉCNICAS

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 independiente 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
```



¿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.)



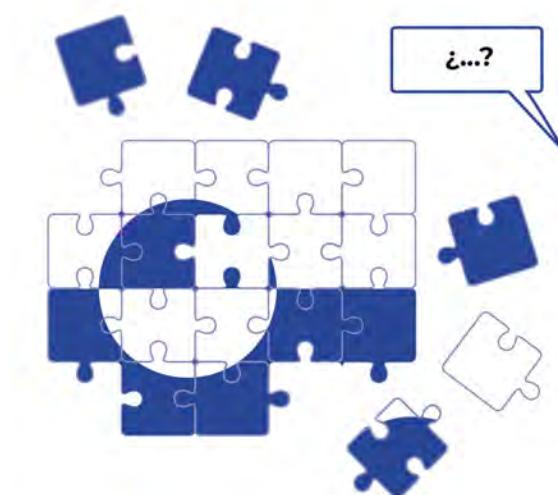
TIPADO FUERTE O DÉBIL

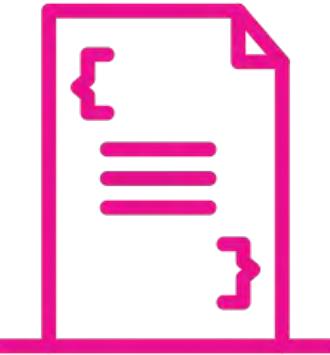
Suelen definirse como aquellos lenguajes que tienen más restricciones (o menos) en su tipado. Por ejemplo, Javascript y Elixir son tipados dinámicos pero Javascript permite sumar 1 (como entero) + '1' (como string) y da como resultado '11' (hace una concatenación). En Elixir recibiremos un error en tiempo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

Tipado dinámico



Tipado estático





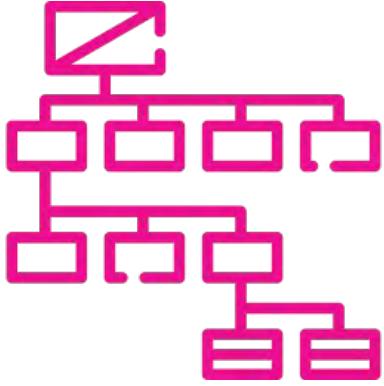
Un Resumen

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()



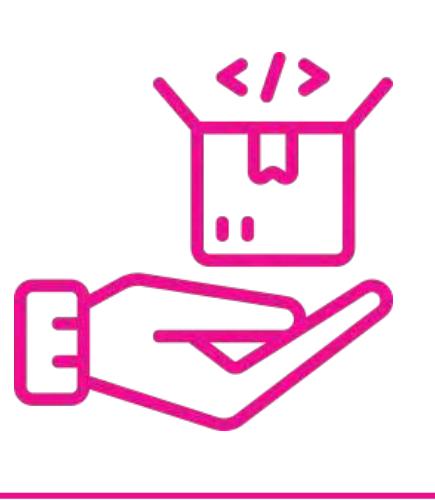
WEAKMAP

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()`.



Promesas

¿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.

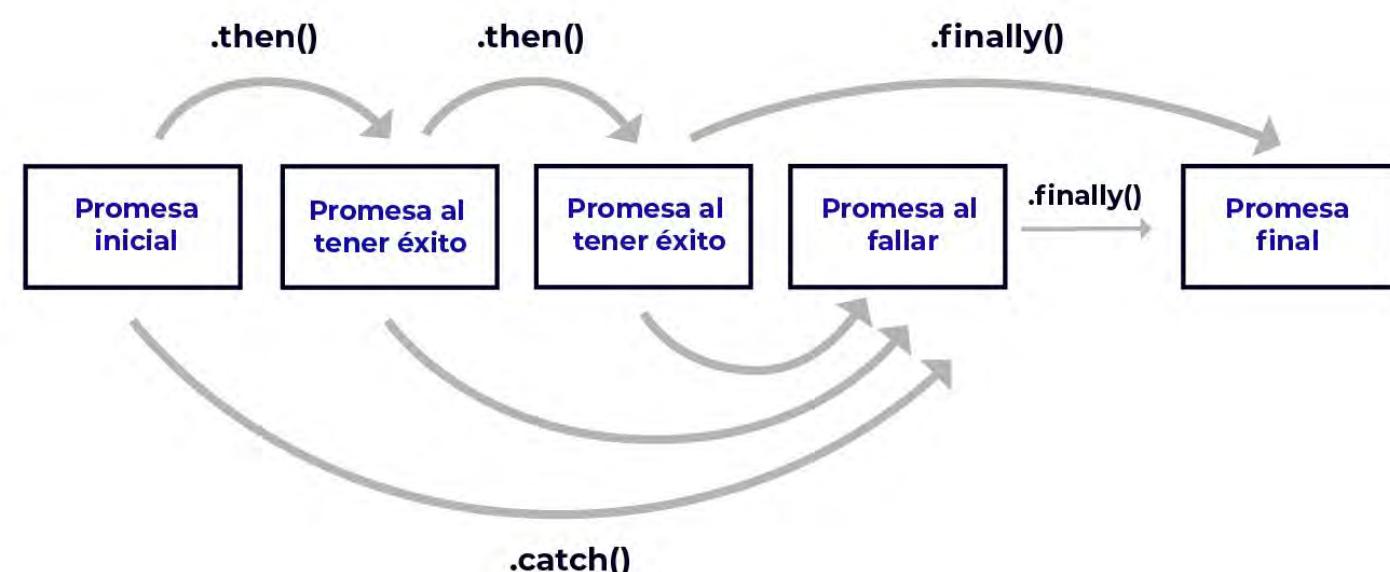


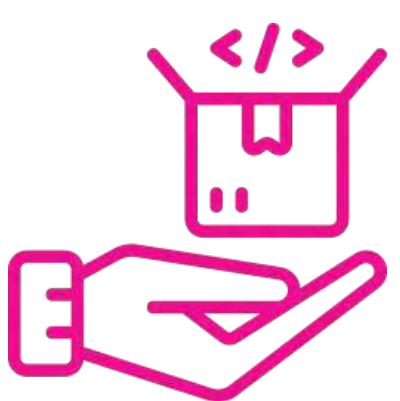
MÉTODOS

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.





¿Qué son?

Then/catch y async/await son dos formas distintas de 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) => getFileSize(files))
  .then((fileSize) => // 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);
    });
  });
});
```



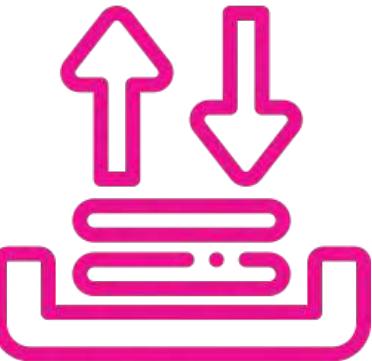
ASYNC / AWAIT

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);
}
```



¿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.

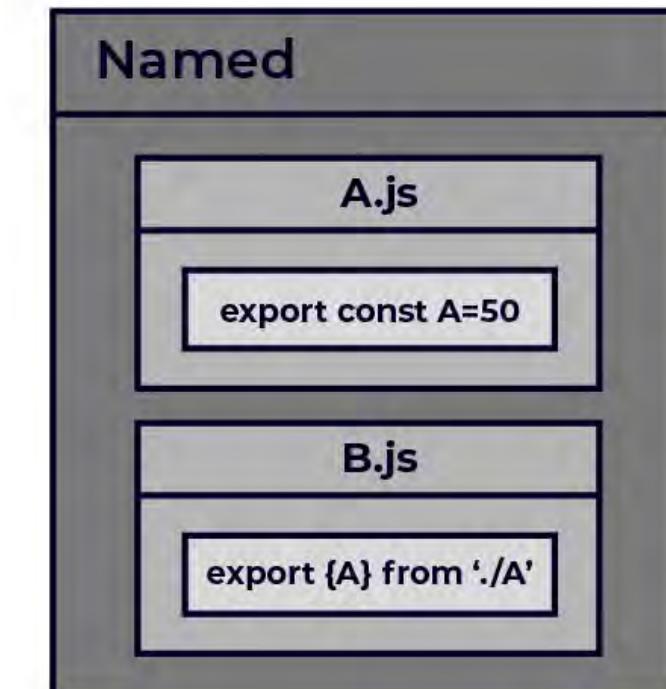
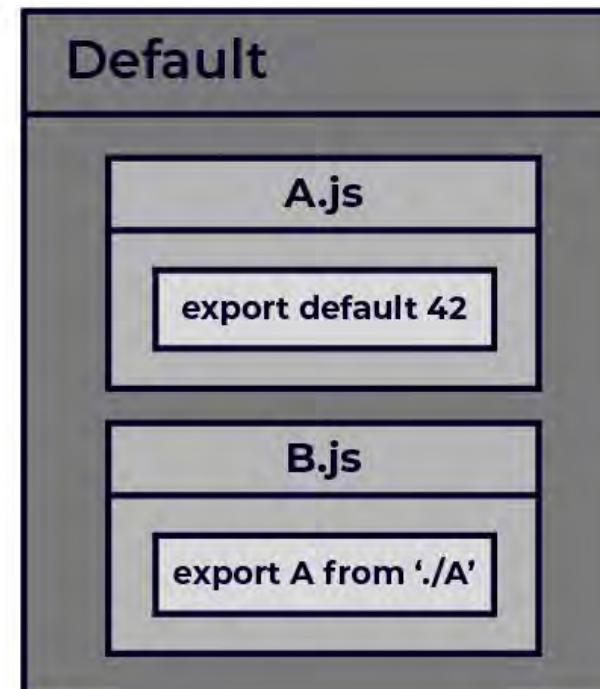
Podemos combinar el uso de *named* y *exports* aunque no se recomienda usar *default* ya que damos la libertad al desarrollador a importar el módulo con distintos nombres en distintos ficheros y a la hora de hacer un refactor en el futuro, puede dar muchos dolores de cabeza.

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.



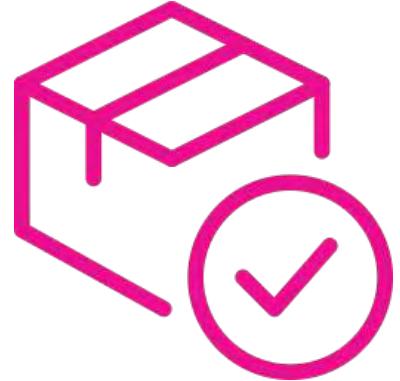
VENTAJAS

- 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",
  }
}
```

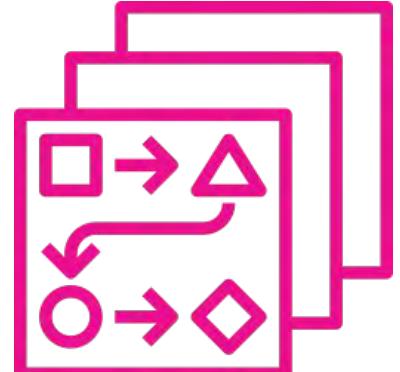


ESTRUCTURA DE PACKAGE.JSON

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_COMMANDO`.
- **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

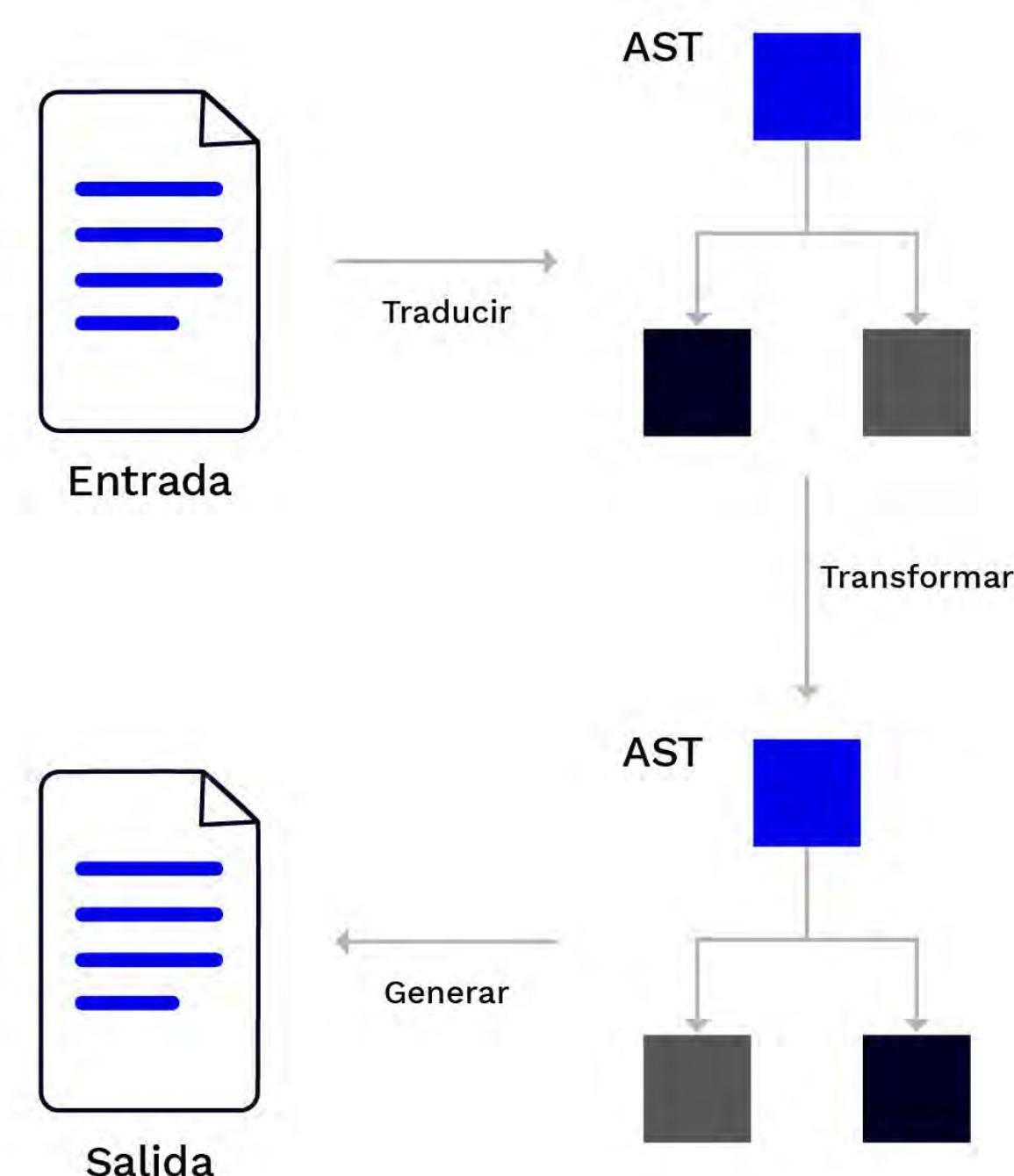
Babel coge un código de entrada y le hace una serie de transformaciones para entregar otro código como salida. Por defecto no hace ninguna transformación, **son los plugins los que hacen las transformaciones**. Entre las cosas que se pueden hacer con Babel están:

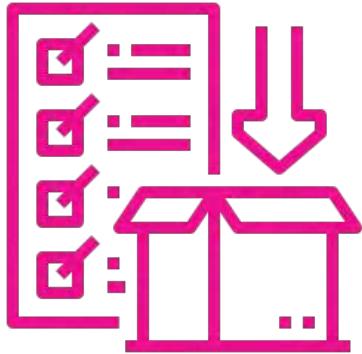
- Convertir la sintaxis entre versiones.
- Introducir Polyfills para funcionalidades más modernas.
- Automatizar refactorizaciones de código.

¿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**.

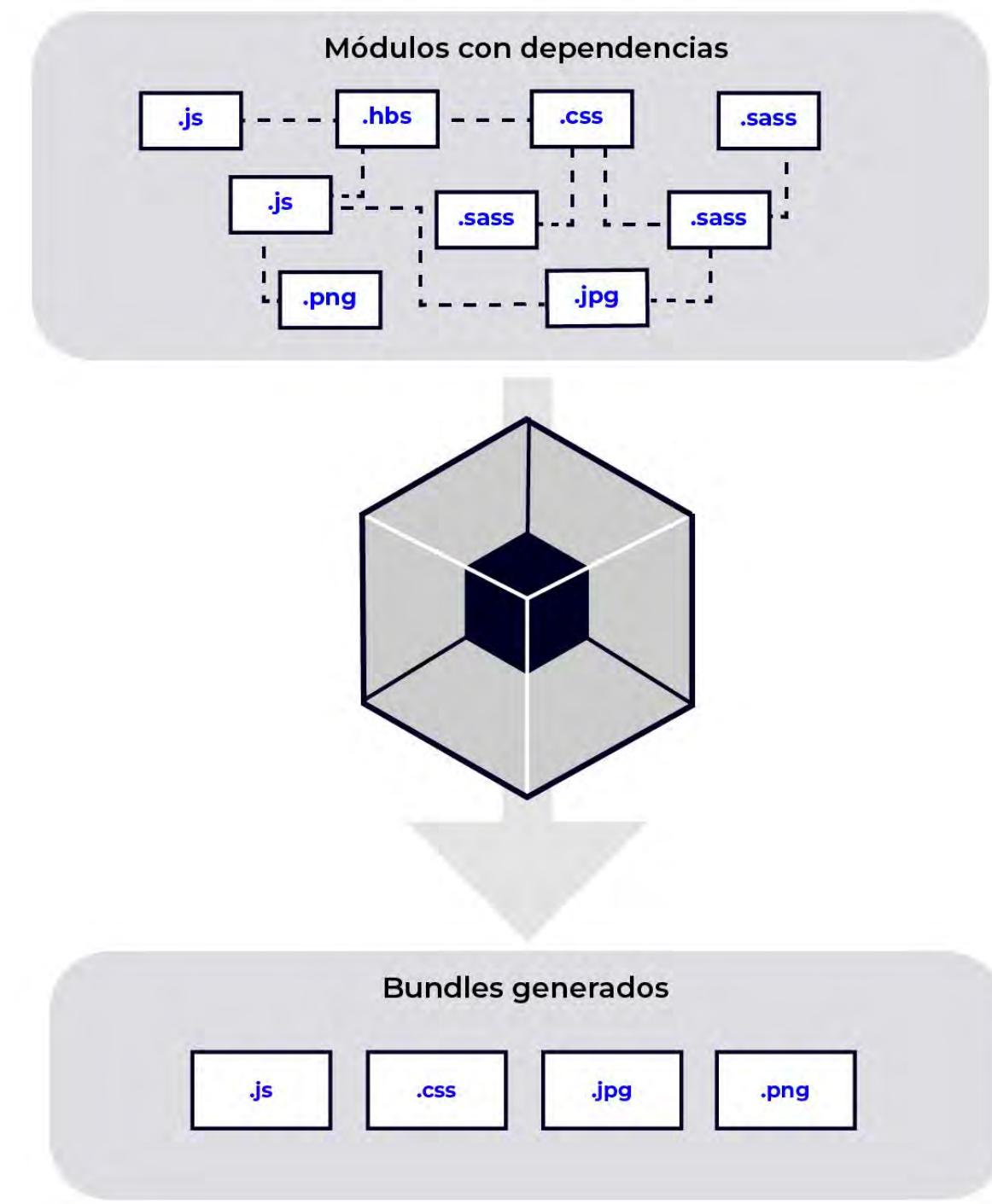


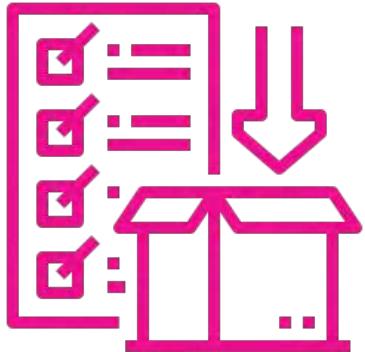
CARACTERÍSTICAS

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**.

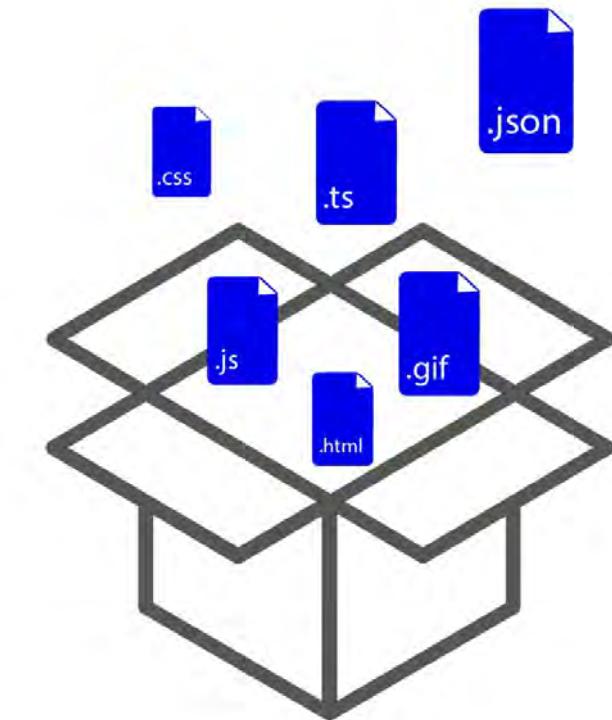
Para instalar Parcel, tan solo debemos ejecutar `npm install --save-dev parcel-bundler`. Seguidamente indicamos qué fichero será el punto de entrada del proyecto y con el flag `-p` indicamos un puerto específico para el servidor que nos ofrece Parcel en caso de que queramos visualizar lo que se ha generado: `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.

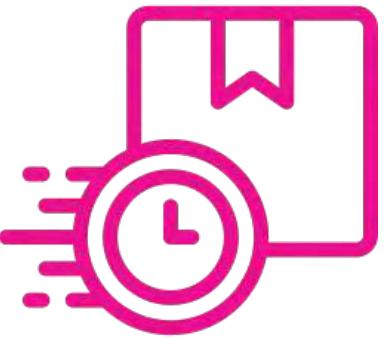


VENTAJAS

- **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





¿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.

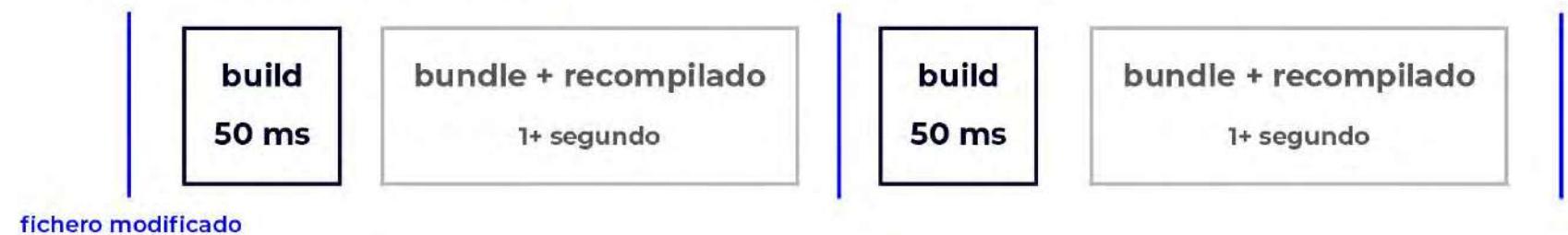
Al hacer el despliegue de la página o aplicación web, Snowpack es compatible con bundlers como Webpack o Parcel, lo que permite optimizar el código de producción. Además, durante el desarrollo, Snowpack servirá la aplicación sin empaquetar y sin depender del bundler usado en producción, por lo que los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



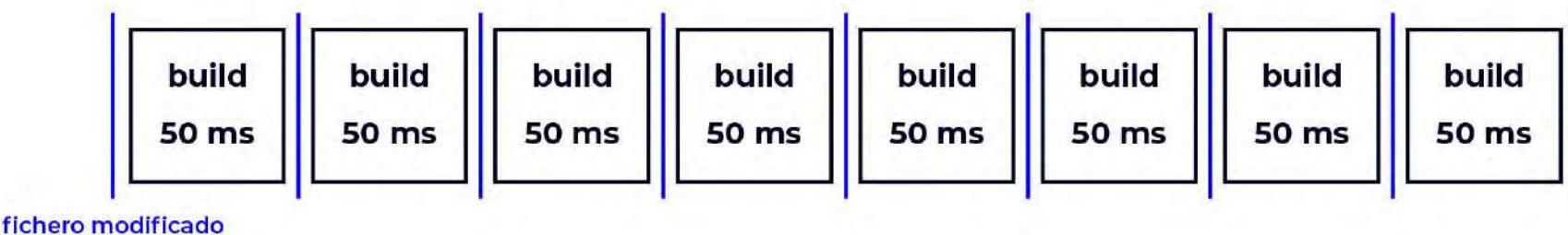
CARACTERÍSTICAS

- **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.

Bundled (ex: Webpack)



Unbundled (Snowpack)





¿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: var(--s) var(--m);
    border-radius: 7px;
}

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



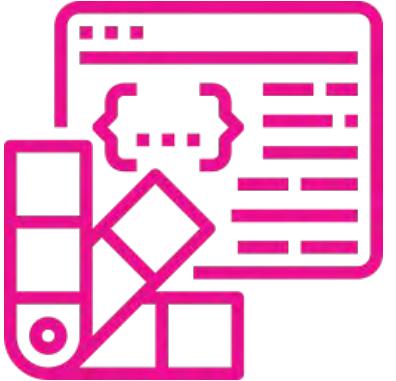
VENTAJAS

- 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.

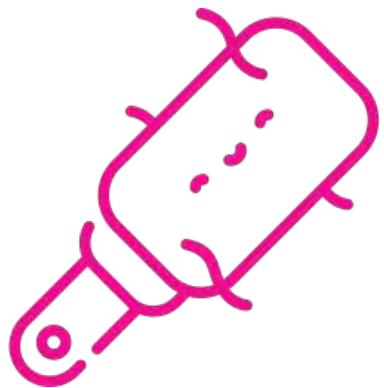
Con PostCSS puedes escoger qué funcionalidades utilizar, mientras que los demás preprocesadores incluyen un montón de funcionalidades que puedes necesitar o no.

AUTOPREFIXER

```
:fullscreen {
```



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



Lint de código

¿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

```

1  'use strict';
2  var foo = "bar";
3
4  fn(function (err) {});
```

Error foo is defined but never used (no-unused-vars) at line 2 col 5

Error foo is defined but never used (no-unused-vars) at line 2 col 5

Error Strings must use singlequote. (quotes) at line 2 col 11

Error "fn" is not defined. (no-undef) at line 4 col 1

Warning Expected error to be handled. (handle-callback-err) at line 4 col 4

Error err is defined but never used (no-unused-vars) at line 4 col 14

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

Compatibilidad entre navegadores



¿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.

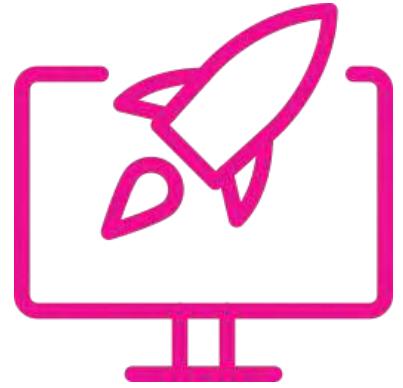


DIFERENCIAS ENTRE NAVEGADORES

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.

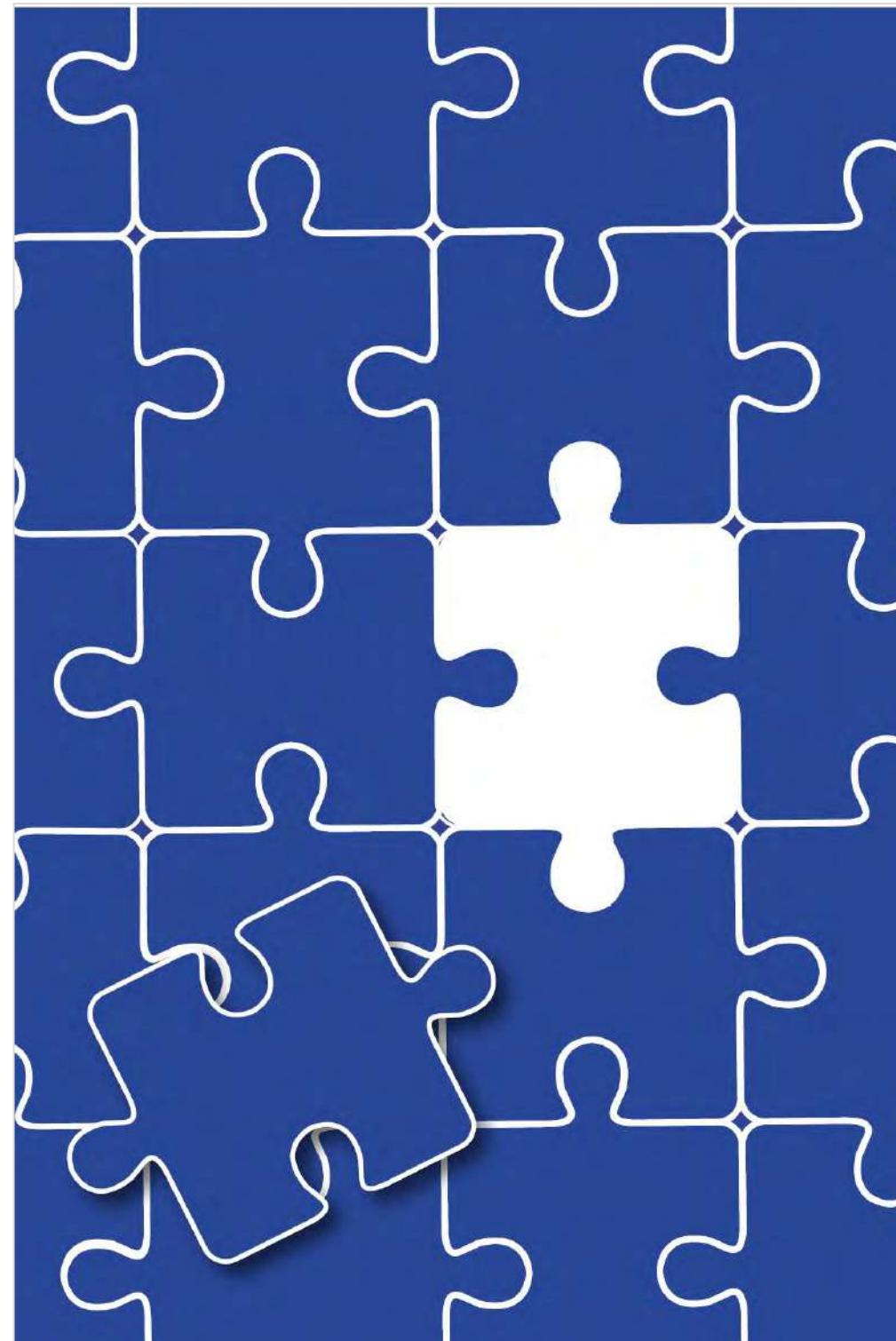


CARACTERÍSTICAS

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

¿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 una 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.



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



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.



WHATWG



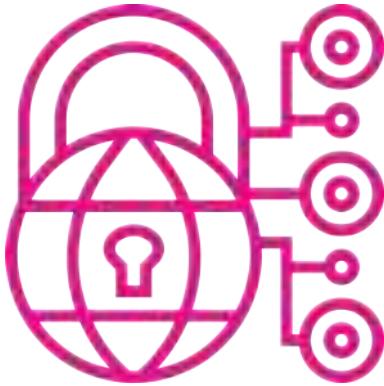
WHATWG

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.

10

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.



¿Qué es?

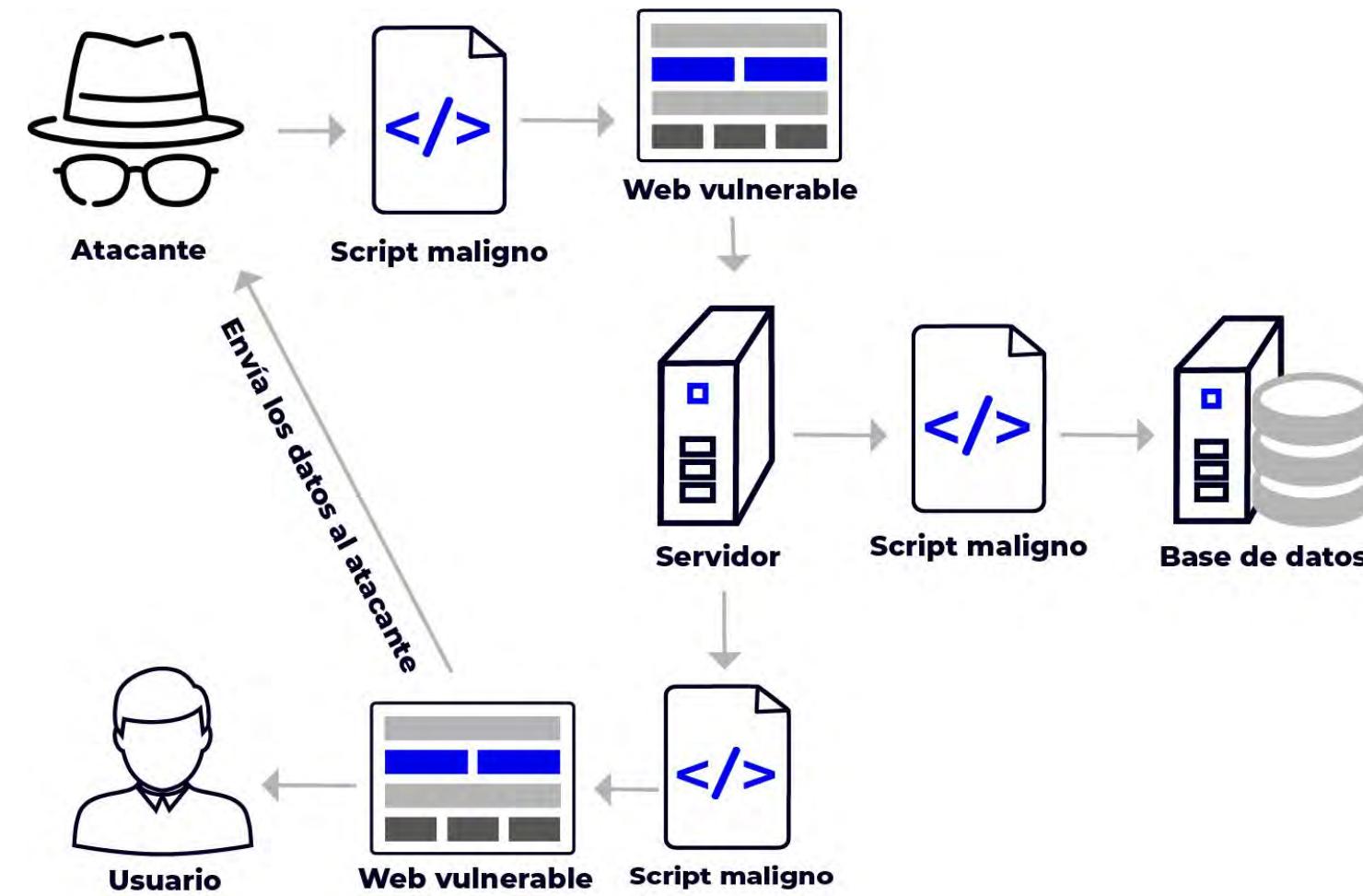
Consiste en conseguir **ejecutar código JavaScript en una página web** para tratar de acceder a datos confidenciales como las credenciales o las cookies, acceder a la cámara, etc.



VULNERABILIDADES

Existen **dos tipos** de vulnerabilidades:

- **Persistente:** el código JavaScript injectado 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 considerarse 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.



¿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 envíe, el servidor espera recibir el token enviado. Si el token falta o el valor es incorrecto, la petición se rechaza.





¿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.

Una captura de pantalla de la consola de desarrollo de Chrome. La parte izquierda muestra una lista jerárquica de recursos (location, m-outer, inner.html, conversion_async.js, etc.). La parte derecha muestra las cabeceras HTTP detalladas:

Name	Value
content-security-policy	script-src 'none'; object-src 'none'
content-type	image/gif
date	Mon, 15 Jun 2020 06:19:23 GMT
expires	Fri, 01 Jan 1990 00:00:00 GMT
p3p	policyref="https://www.googleadservices.com/pagead/p3p.xml", CP="NOI DEV PSA PSD IVA IVD OTP OUR OTR IND OTC"
pragma	no-cache
server	cafe
status	200
timing-allow-origin	*
x-content-type-options	nosniff
x-xss-protection	0



¿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.

10

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

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

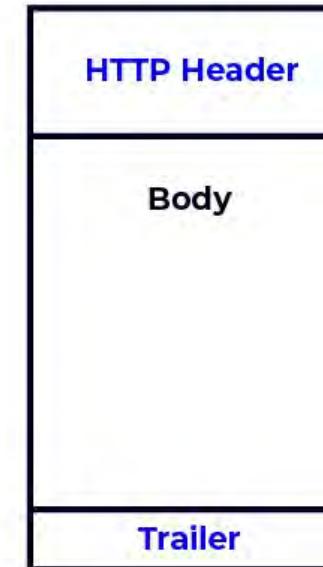
- **Cookie:** la cookie HTTP es un dato que permite al servidor **identificar las peticiones que viene de un mismo cliente**. HTTP es un protocolo sin estado. El servidor asigna una cookie a cada cliente y este las referencia usando la cabecera Cookie.
- **User-Agent:** identifica el tipo de aplicación, 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**.



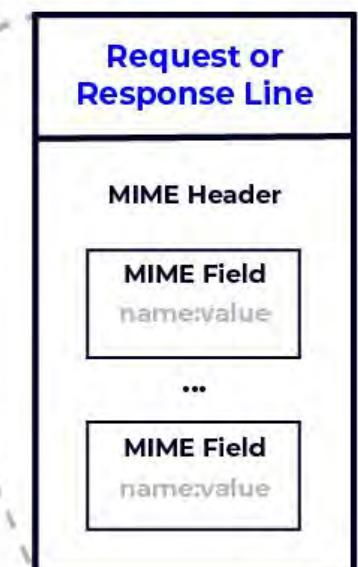
CABECERAS PRINCIPALES EN LA RESPONSE

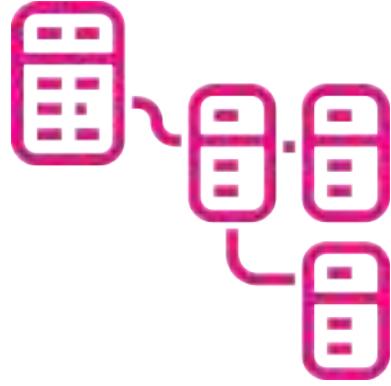
- **Content-Type:** determina el tipo de cuerpo (tipo MIME) y la codificación de la respuesta.
- **Content-Length:** indica el tamaño del cuerpo de la respuesta en bytes.
- **Set-Cookie:** es la cabecera que utiliza el servidor para enviar la cookie asignada al cliente. Con esta cookie, el servidor puede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

HTTP Request or Response



HTTP Header





¿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**.

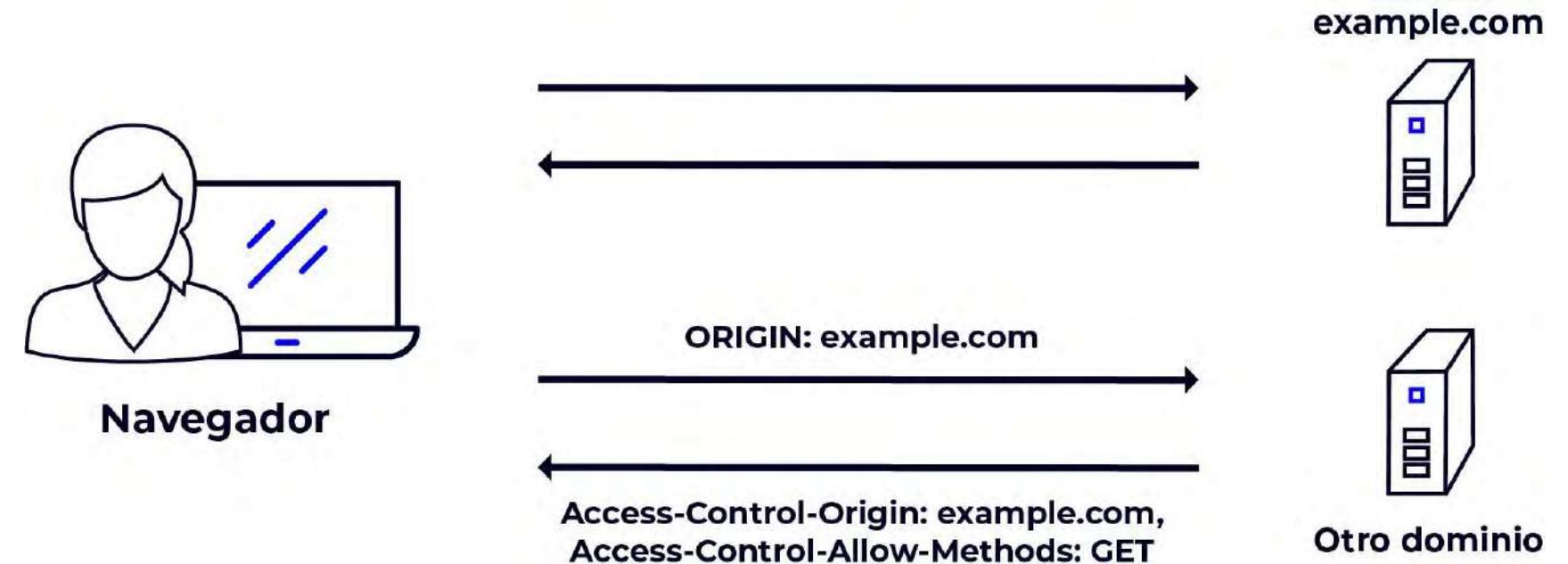


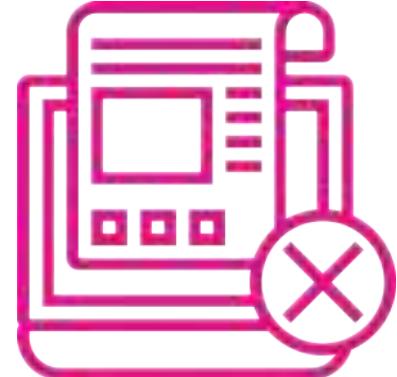
CONSEJOS DE SEGURIDAD

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.

Software Design: Principios y patrones del desarrollo de software



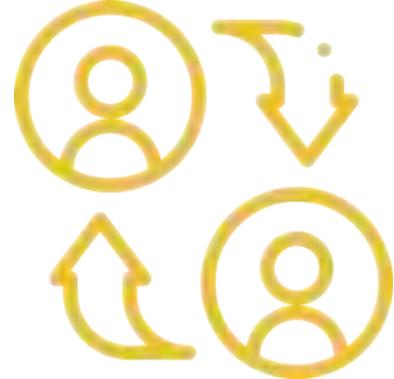
¿Qué son?

Los “olores de diseño” dentro del desarrollo del software son una manera de definir los síntomas que surgen con el avance del proyecto y crecimiento del software que influyen en la degradación del diseño.



CARACTERÍSTICAS

Síntoma	Definición	Consecuencias
Rigidez	Dificultad de realizar cambios en el software, incluso en tareas sencillas.	<ul style="list-style-type: none">• Las estimaciones son más abultadas.• Añadir funcionalidades nuevas cuesta mucho cuando antes era más rápido.
Fragilidad	Tendencia a que el software se rompa en múltiples sitios cada vez que se realiza un cambio en él, incluso sin tener relación el cambio con lo que se rompe.	<ul style="list-style-type: none">• Subir una nueva versión y que se rompan partes que aparentemente no tienen que ver con lo modificado.• Aparición de mayor cantidad de bugs, aumentando el coste de esfuerzo y tiempo en corregirlos.
Inmovilidad	Incapacidad de reutilizar piezas de código por el gran acoplamiento que tiene, haciendo que este sea inamovible cuando la funcionalidad es prácticamente la misma que la deseada.	<ul style="list-style-type: none">• Aumenta la complejidad del diseño y del código al introducir código duplicado.• Complica el entendimiento del código y genera equivocaciones.
Viscosidad	La tendencia de que, en el ámbito del software, realizar la solución buena requiere mayor esfuerzo comparado con la solución mala y rápida y, en el ámbito del entorno de desarrollo, éste es lento e ineficiente.	<ul style="list-style-type: none">• Provoca el efecto bola de nieve: cuanto más tarde se corrija esa deuda técnica más difícil y costoso será resolverlo.• Pérdida de tiempo, estrés, confusión, pasotismo, acciones inseguras, etc.



Una clase debe tener solo una razón para cambiar

El **Single Responsibility Principle (SRP)** o principio de responsabilidad única es un principio que defiende que una clase o módulo debería tener responsabilidad sobre una única funcionalidad del software.



CONCEPTO

Robert C. Martin define el principio con la siguiente frase: **“Una clase debe tener solo una razón para cambiar.”**

¿Pero qué entendemos como “razón”?

Robert aclara que el principio trata realmente sobre las **personas**. No queremos que una misma pieza de código tenga que cambiar debido a personas distintas ya que las personas son las que dirigen los cambios en el software.

Por ejemplo, un cambio en *Employee* sobre su forma de trabajar vendrá requerido por una persona distinta del que vendrá un cambio sobre la tecnología de persistencia de datos.

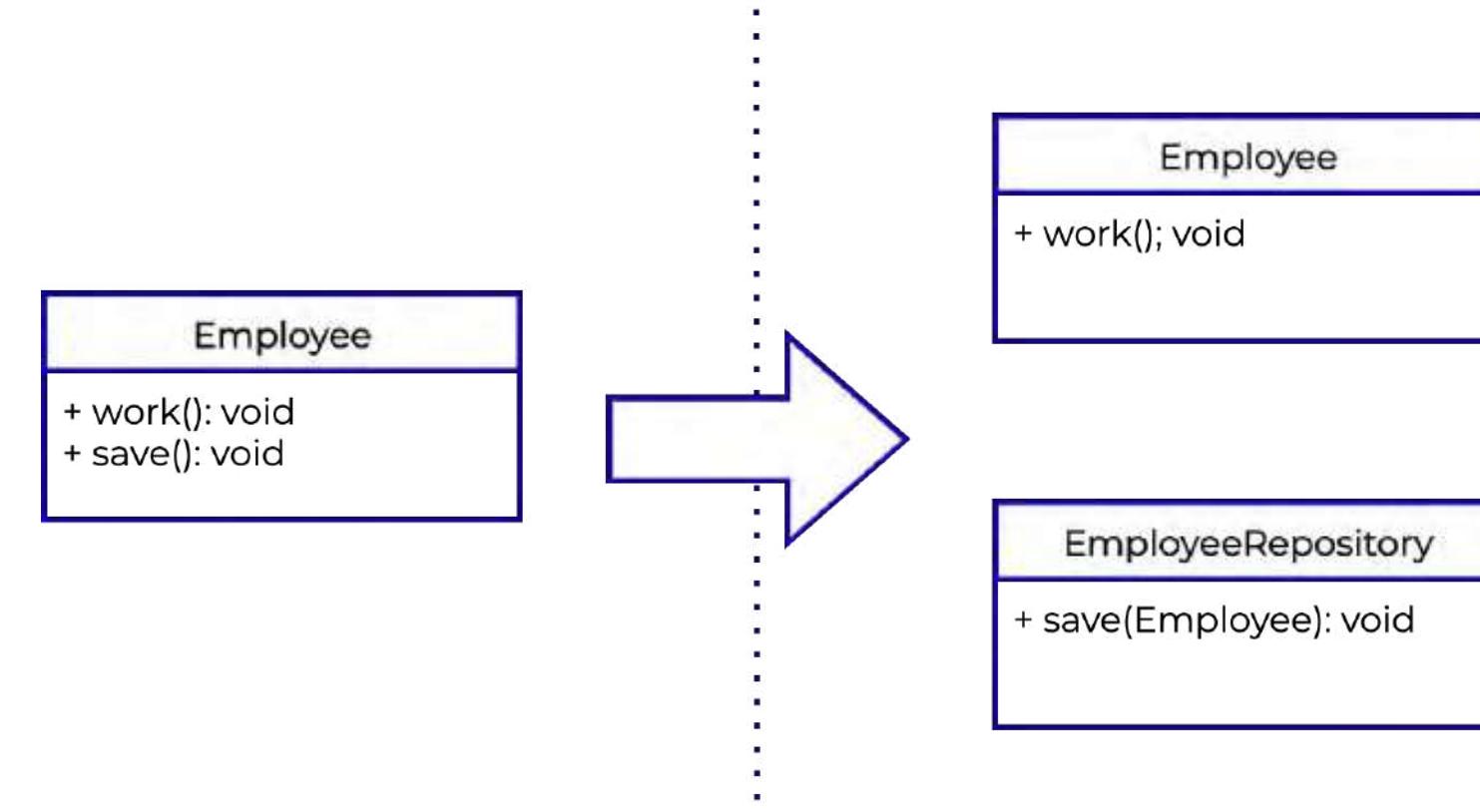
Esto podría entenderse como **razones** distintas para cambiar.



EJEMPLO

Podemos identificar que la clase *Employee* tiene dos responsabilidades, la propia de un empleado que es trabajar y la de persistir su estado.

Podemos separar ambas responsabilidades en dos clases:





Abierto a la extensión, cerrado a la modificación

Representa la O de SOLID. Con este principio se pretende minimizar el efecto cascada que puede suceder cuando cambiamos el comportamiento de una clase. Si existen clientes que dependan de ella, es posible que tengan que cambiar su comportamiento también.



PLANTEAMIENTO

El principio consta de dos partes:

- Las clases deben estar **abiertas** a la extensión. La extensión se refiere a las modificaciones que pueden ocurrir cuando se plantean nuevos requisitos de software.
- Las clases deben estar **cerradas** a la modificación. No se puede cambiar el código fuente de una clase.

El polimorfismo es la herramienta principal para unir estos dos conceptos que a primera vista parecen contradecirse.

Este principio nos proporciona una mayor estabilidad de los clientes que dependan de la antigua clase. La antigua clase ya funciona, está probado y demostrado. Si incorporamos nuevas funcionalidades a ella, aumentamos la posibilidad de introducir bugs en el software. Por ello, cada vez que se quiera añadir o modificar un comportamiento, en vez de cambiar el código existente, se extiende la clase abstracta o la interfaz y se realizan los cambios en una nueva clase.

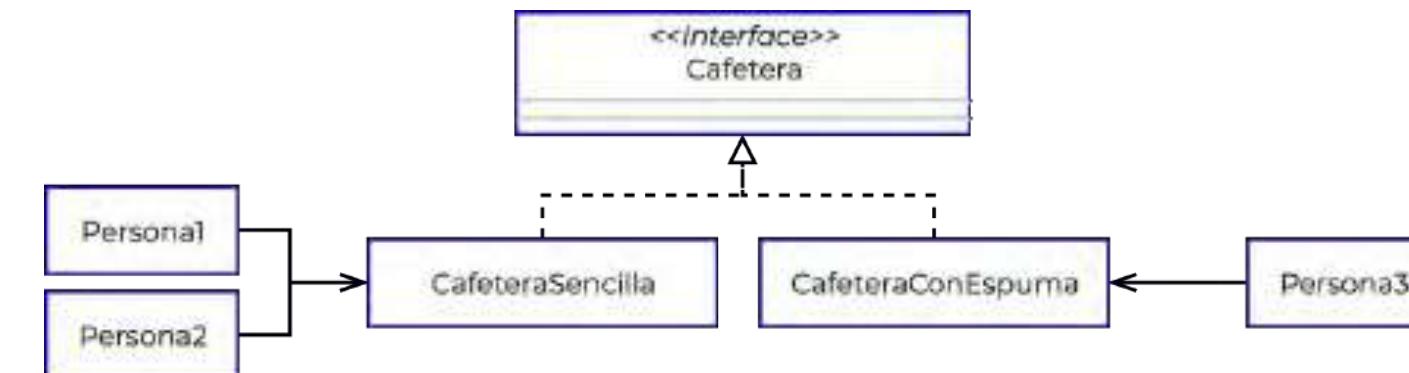


EJEMPLO

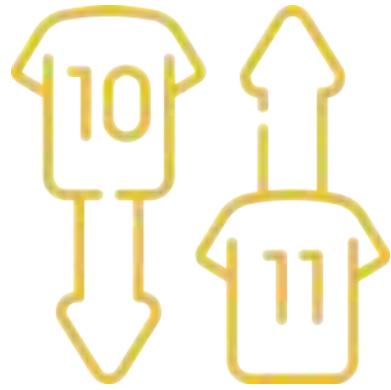
En una oficina se encuentra una cafetera utilizada por varias personas. Un día, alguien pidió que el café se pudiera preparar distinto. Esta solicitud implicaba un cambio en la configuración y comportamiento de la cafetera.



En vez de cambiar el comportamiento de la cafetera actual para intentar satisfacer a dos clientes distintos, aplicaremos el principio Open/Closed. Cerramos nuestra clase `CafeteraSencilla` ante cualquier modificación, pero creamos una interfaz para abrirla a extensiones:



Logramos cumplir con el nuevo requisito sin tener que arriesgar los anteriores. Es más, ahora ofrecemos la posibilidad de utilizar una cafetera o la otra, en caso de que cambien de opinión.



Definición

Concepto introducido por Barbara Liskov que representa la L de los principios S.O.L.I.D. El principio dice: **una clase que hereda de otra debe poder usarse como su padre sin necesidad de conocer las diferencias entre ellas.**



¿EN QUÉ CONSISTE?

Este principio nos ayuda a utilizar correctamente la herencia ya que **los objetos de nuestras subclases se deben comportar de igual forma que los de la superclase.**

Imaginemos que tenemos una clase que hereda de otra, pero hay un método que no se necesita o no se usa en esa clase. Para solucionar esto, podríamos devolver null o una excepción en ese método. Al hacer esto, estamos violando el Principio de Sustitución de Liskov. Si el método de la clase original no lanza ninguna excepción, los métodos sobrescritos de las subclases tampoco deberían hacerlo. O, si estamos heredando de clases abstractas que nos obligan a devolver null o lanzan una excepción, también estaríamos ante una violación del principio.

Hay una frase muy conocida que dice 'Si se ve como un pato, hace cuac como un pato, pero necesita baterías, probablemente tengas la abstracción incorrecta'.



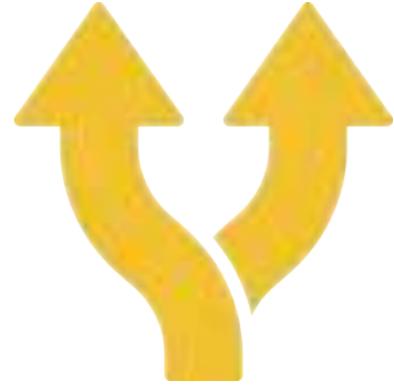
DISEÑO POR CONTRATO

Para cumplir con el principio, Liskov propuso un concepto parecido al *diseño por contrato* (*design by contract*).

Cuando se llame a un método de la subclase, estos deben cumplir una serie de precondiciones y postcondiciones. Las precondiciones deben ser verdaderas para que el método se pueda ejecutar. Una vez se ha ejecutado, las postcondiciones deben ser verdaderas también.

Se establecieron ciertas restricciones sobre cómo el contrato de una superclase podía seguir ciertos patrones que permitiese aplicar la herencia correctamente.

- Las **precondiciones** no pueden ser **más restrictivas** en un subtipo.
- Las **postcondiciones** no pueden ser **menos restrictivas** en un subtipo.
- Las **invariantes** establecidas por el supertipo deben ser mantenidas por los subtipos.



¿Cómo lo aplico?

El **interface Segregation Principle (ISP)** o principio de segregación de interfaces defiende que ninguna clase debería depender de métodos que no usa.



CONCEPTO

Cuando creamos una interfaz debemos estar seguros de que la clase que va a implementar la interfaz va a poder implementar todos los métodos.

En caso contrario **debemos separar la interfaz en interfaces más pequeñas** con menos métodos.

A veces, anticipar qué métodos va realmente a necesitar las implementaciones no es sencillo. Si hemos aplicado correctamente el principio de responsabilidad única y el principio de sustitución de Liskov podremos tener una buena aproximación.

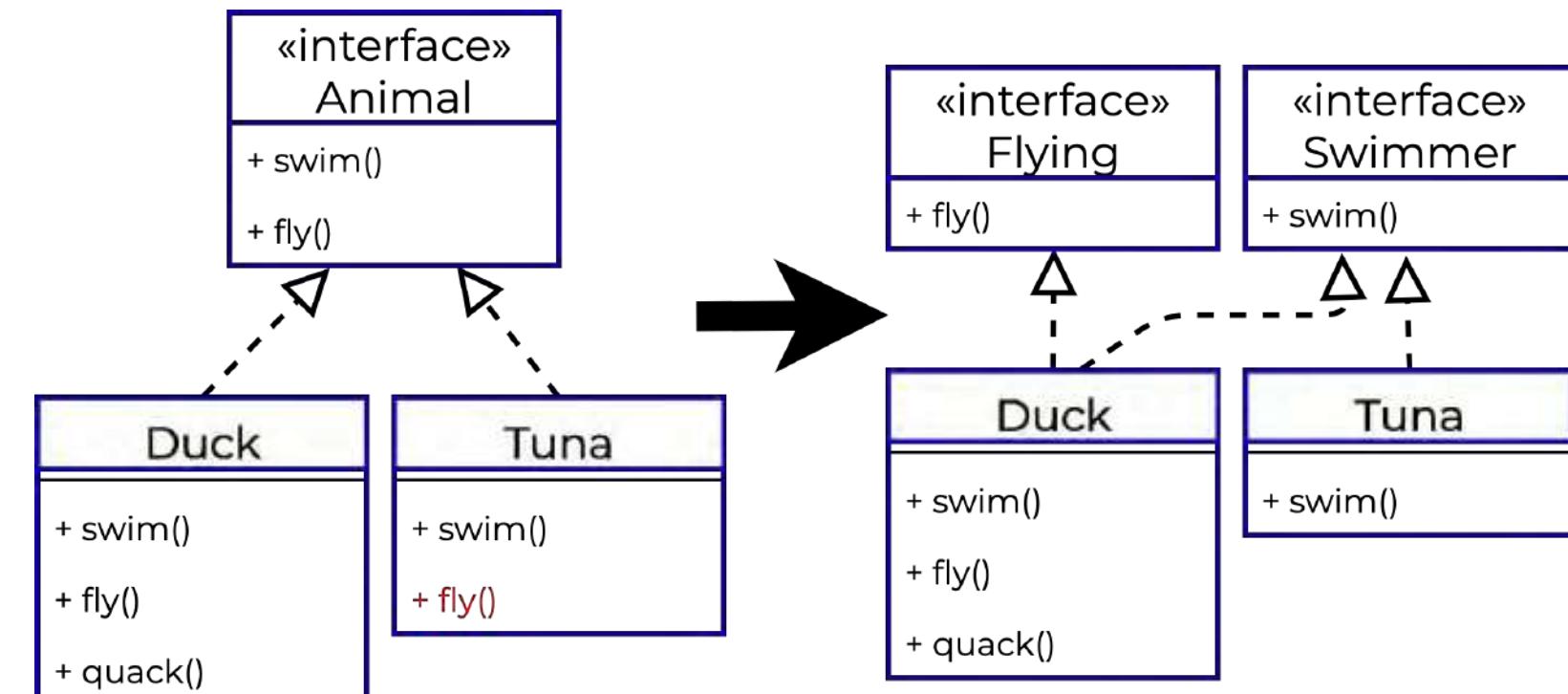
Por lo general siempre será preferible **muchas interfaces pequeñas** a una gran interfaz con muchos métodos.

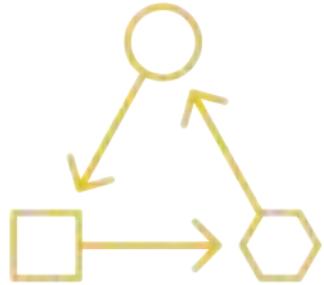


EJEMPLO

En el ejemplo podemos observar que un Atún se ve forzado a tener un método volar, el cual no puede implementar, ya que un atún no puedo volar. Podríamos separar ambas habilidades en distintas interfaces. Una para voladores y otra para nadadores.

De este modo los animales implementarán solo las interfaces que necesiten.





¿Qué es?

Representa la D de los principios S.O.L.I.D. El principio dice, los **módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones**. El principio tiene como fin evitar depender de concreciones para minimizar el grado de acoplamiento entre los componentes.

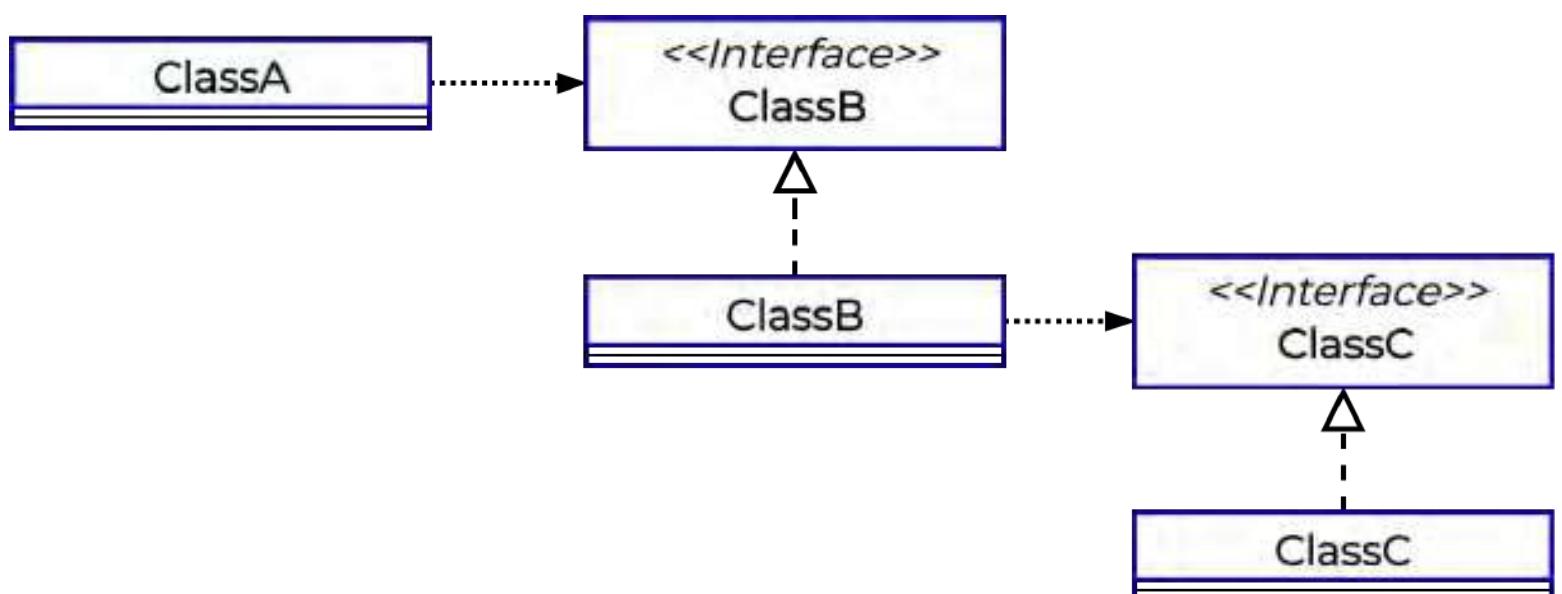


PROBLEMA - SOLUCIÓN

Normalmente, las capas de alto nivel (ClassA) consumen las de bajo nivel (ClassC), generando un fuerte acoplamiento.



Para evitar este problema, se introduce una capa de abstracción que permite reutilizar las capas de mayor nivel.



VENTAJAS

Si aplicamos correctamente los anteriores principios SOLID como el Open/Closed y el de Liskov, estamos implícitamente cumpliendo con la inversión de dependencias. Pero, ¿qué ventajas nos aporta?

- **Mayor flexibilidad haciendo testing:** gracias a que se depende de abstracciones, podemos reemplazar objetos reales por mocks que simulen el comportamiento deseado.
- **Reduce el acoplamiento** entre clases: al no depender de una implementación en concreto, no acoplamos nuestro código a ciertas clases específicas.
- Código más **limpio, legible y mantenible** en el tiempo: al no depender de implementaciones, estamos contribuyendo a un desarrollo más robusto y que no sea frágil a cualquier cambio.
- Al depender de abstracciones, tenemos la **posibilidad de elegir en tiempo de ejecución la implementación** adecuada.



Principio DRY

¿Qué es?

El principio DRY es un acrónimo del inglés “Don’t Repeat Yourself” que significa “No te repitas” y éste consiste en evitar las duplicaciones lógicas en el software.



DESTACAR

Evitar duplicaciones lógicas

≠

Evitar duplicaciones de código



OBJETIVOS

Cada pieza de funcionalidad lógica debe tener dentro del sistema:

- **Una identidad única.**
- **No ambigua.**
- **Representativa.**



EJEMPLO

1. Imagina tener tres métodos para abrir una conexión a una base de datos. Cada uno tiene su propio código, pero la función final es conectarse a la base de datos.
2. Si más adelante, cambia la manera de conectarse a la base de datos, el tipo de base de datos o enviar datos a otros sistemas de almacenamiento. Se deberá modificar los tres métodos.
3. Triplicando el coste de trabajo, introduciendo más posibilidad de cometer errores, aumentando la complejidad del código, etc.
4. Si no eres el autor original, la labor de mantenerlo lo complica aún más.



VENTAJAS

Las ventajas de la aplicación del principio son:

- **Hace el código más mantenible.** Cualquier cambio en la funcionalidad lógica, no tienes que cambiarlo en todos los sitios repetidos. Por lo tanto, evita fallos si las funcionalidades repetidas no se encuentran sincronizadas.
- **Reduce el tamaño del código.** Tener menos código ayuda a que el código sea más legible y entendible.
- **Ahorra tiempo.** La disponibilidad de funcionalidades lógicas para reutilizar en el futuro, permite estar más preparados para lograr lo mismo en menos tiempo.



Inversión de control

¿Qué es?

En la programación tradicional, la interacción entre clases y funciones se hace de forma imperativa. La inversión de control es un principio que delega en un tercero (un framework o contenedor) el control del flujo de un programa para la creación de un objeto, la inyección de objetos dependientes, etc.



¿EN QUÉ CONSISTE?

La inversión de control **está basada en el principio de Hollywood**, ya que era muy habitual la frase que decían los directores a los aspirantes: "No nos llames; nosotros te llamaremos".

Podemos encontrar distintas formas de implementar la inversión de control. Las formas más conocidas de este principio son:

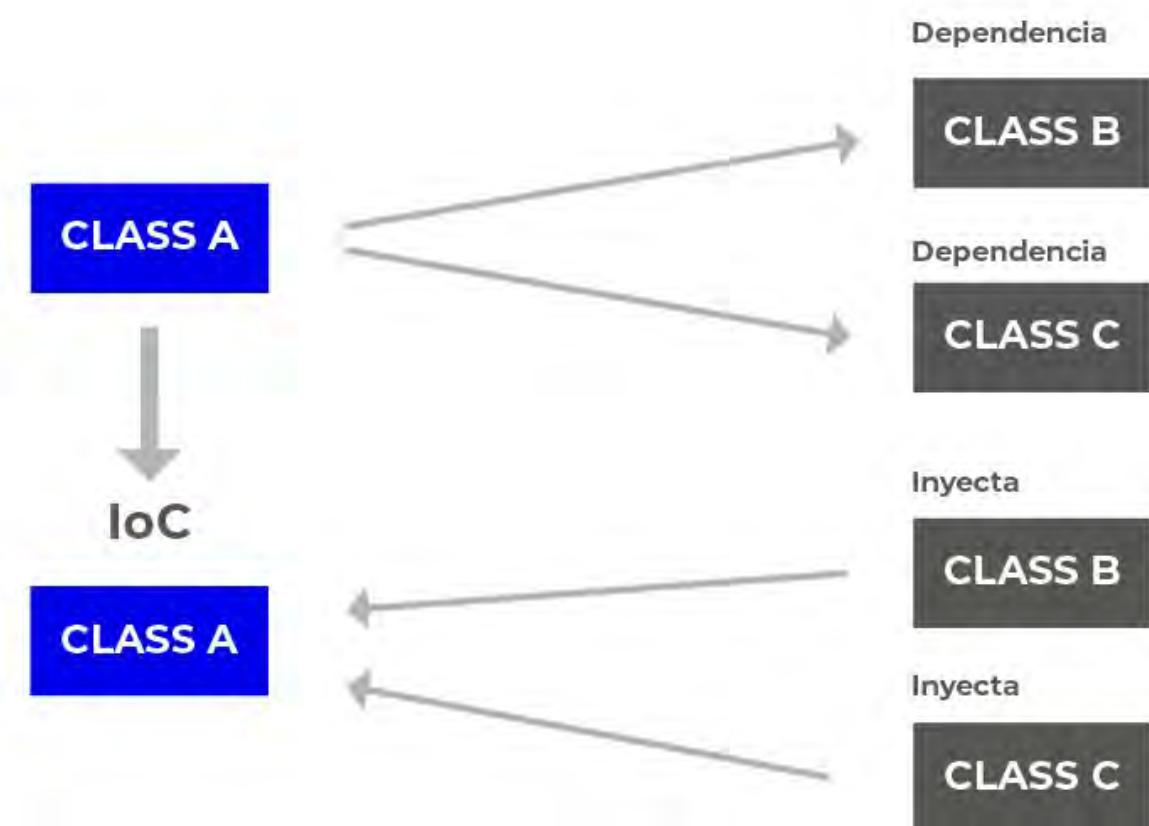
- Localizador de servicios (Service Locator).
- Inyección de dependencias.

La diferencia fundamental entre ambas es que con el Service Locator las dependencias aún se solicitan **explícitamente** desde la clase dependiente, mientras que con la inyección de dependencias, un agente externo se encarga de proveer las mismas, sin mediar acoplamiento entre la dependencia y su proveedor.



VENTAJAS

- **Reduce el acoplamiento** entre clases y a su vez **aumenta la modularidad y extensibilidad**.
- A raíz del punto anterior, **mejora la testeabilidad** del código ya que al reducir el acoplamiento podemos crear dobles de prueba de las dependencias de una clase de una forma muy sencilla.





Origen

Kent Beck describió cuatro reglas fundamentales para diseñar software en la década de los 90. Su objetivo era medir la calidad del software de manera objetiva buscando la perspectiva de minimizar los costes y maximizar el beneficio, huyendo de valoraciones subjetivas.



REGLAS

Las 4 reglas en orden de relevancia son:

1. **Los test pasan:** en el desarrollo del software, el testing nunca debería faltar. Los tests son los garantes de que la tarea funciona como se espera y que los criterios establecidos se cumplen.
2. **Expresan intención:** el código tiene que ser autoexplicativo, sencillo de entender y facilitar la comunicación del propósito del mismo .
3. **Sin duplicaciones (DRY):** debe reducirse al máximo la duplicidad de la lógica en el código. Hacer esto evita construir software frágil.
4. **Mínimo número de elementos:** debe reducirse a lo imprescindible el número de componentes, clases, métodos, etc. eliminando todas aquellas cosas que incrementen la complejidad del sistema de manera innecesaria.



CONTROVERSIAS

Tras tanto tiempo desde la definición de estas reglas se han generado multitud de discusiones en foros, libros... dando lugar a cantidad de **ideas interesantes** como:

- No hay unanimidad en el orden de los **puntos 2 y 3**, haciendo a la idea de que ambos están al mismo nivel.
- La **primera regla** podría no ser considerado como una regla del diseño simple, sino como algo esencial o inherente al desarrollo del software. Es decir, no debería plantearse siquiera el desarrollo de código sin tests.
- La **cuarta regla** es considerada para muchos como una consecuencia de la aplicación continua de las reglas 2 y 3.



Patrones de diseño GoF

¿Qué son?

Los patrones de diseño **ofrecen soluciones a problemas recurrentes** en el desarrollo del software. Normalmente constan de una serie de pautas a seguir (una receta) que resuelven un problema concreto que ha sido ya probado y documentado por gran parte de la comunidad.



GoF (Gang of Four)

GoF surgió a raíz del libro 'Design Patterns - Elements of Reusable Software' escrito por cuatro desarrolladores que descubren una forma esencial de enfrentarse a la programación.

Aplicando con criterio el uso de patrones, podemos desarrollar **software más robusto, escalable y mantenible**, pero tampoco se debe abusar de ellos y seguirlos al pie de la letra. Debemos ser flexibles ya que dependiendo de nuestras necesidades, se pueden implementar de una forma u otra.

Características de los patrones:

- Proponen **soluciones sólidas a problemas concretos** probadas por la comunidad.
- Buscan **maximizar la cohesión y minimizar el acoplamiento**.
- Se basan en principios (SOLID, separation of concerns, ley de Demeter, etc.) que favorecen el **código limpio**.



TIPOS

- **Creacionales:** se utilizan para la instanciación de objetos, encapsulan la lógica de creación y aíslan al cliente de esta responsabilidad. Se intenta evitar el uso del operador **new** entre clases para reducir el acoplamiento.
- **Comportamiento:** se utilizan para definir la interacción entre distintos objetos. La interacción debe ser de tal manera que puedan comunicarse fácilmente entre sí, minimizando el grado de acoplamiento.
- **Estructurales:** se utilizan para resolver problemas de composición (agregación) de clases y objetos. Intentan que los cambios en los requisitos de la aplicación no ocasionen cambios en las relaciones entre los objetos. Normalmente estas relaciones están determinadas por las interfaces que soportan los objetos.



¿En qué consiste?

Patrón creacional que permite la **creación de diferentes representaciones de un objeto**. Se utiliza en situaciones en las que el objeto tiene una gran cantidad de atributos en el constructor por lo que la construcción se realiza en un conjunto de pasos.

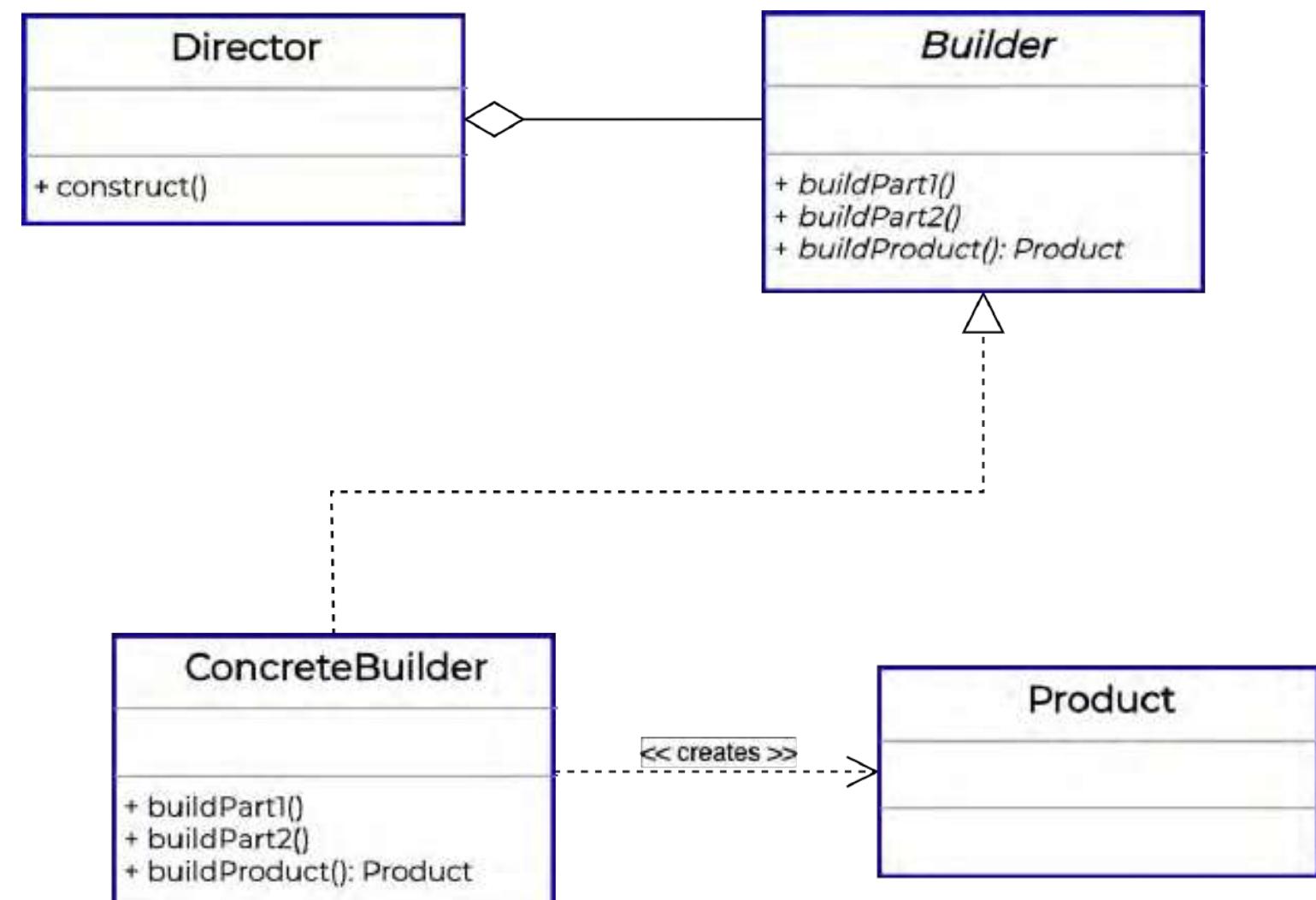


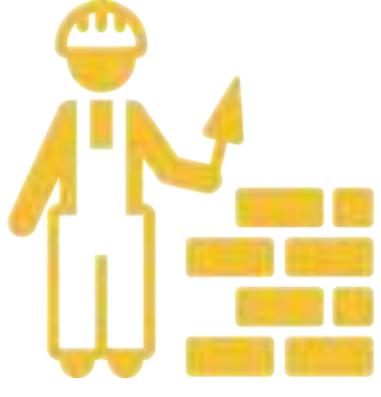
RAZONAMIENTO

Construir un objeto cuyo constructor tiene una larga lista de parámetros, puede llegar a ser tedioso. Sobretodo, porque debemos estar muy pendientes de cuál es el parámetro que va en cada posición. Otro problema es que a veces solo necesitamos construir el objeto con ciertas propiedades ya que las otras no son necesarias. La siguiente línea de código nos muestra un claro ejemplo del problema: **new Product(null, null, null, 'Madrid')**. Un objeto Builder soluciona este problema simplificando la construcción y creando un objeto consistente.

También se suele usar este patrón para construir objetos inmutables por lo que la clase del objeto original no debe tener setters. Se tendrá un '**director**' que se encargue de crear siempre los objetos, de este modo, el cliente se abstraerá de saber con qué propiedades se está construyendo dicho objeto.

Para el siguiente ejemplo no vamos a tener en cuenta esto, pero debemos conocer otro tipo de variaciones del patrón.





Implementación



SOLUCIÓN

Builder organiza la construcción del objeto en una serie de pasos, siendo estos pasos básicamente los métodos que hay por cada atributo del objeto.

¿Qué ventajas tiene?

- **Mayor control** a la hora de construir un objeto.
- Se pueden construir objetos **inmutables**.

El cliente (Director) dirige la construcción del objeto User usando el UserBuilder:

```
User user = new UserBuilder().city("Madrid").build();
```

```
public class User {
    private String name;
    private String username;
    private Long age;
    private String city;

    public User(String name, String
username, Long age, String city) {
        this.name = name;
        this.username = username;
        this.age = age;
        this.city = city;
    }
    // getters...
}
```

```
public class UserBuilder {
    private String name;
    private String username;
    private Long age;
    private String city;

    public UserBuilder name(String name) {
        this.name = name;
        return this;
    }

    public UserBuilder username(String username) {
        this.username = username;
        return this;
    }

    public UserBuilder age(Long age) {
        this.age = age;
        return this;
    }

    public UserBuilder city(String city) {
        this.city = city;
        return this;
    }

    public User build() {
        return new User(name, username, age, city);
    }
}
```



Un único ser sin igual

El patrón *Singleton* resuelve el problema de mantener una única instancia de una clase en memoria durante la ejecución del programa.



DISEÑO

El diseño de este patrón impide que otras clases creen nuevas instancias del Singleton. La primera vez que una clase la necesite, se creará la primera y única instancia de ella.

A partir de ahora, cada vez que se solicita una instancia del Singleton, se hará referencia a la misma instancia, asegurándonos de que no se cree otra.



PRECAUCIÓN

Este patrón se comporta como un objeto global, donde cualquier parte de la aplicación la puede utilizar.

Cambios hechos al estado de una instancia Singleton pueden repercutir en otras clases que la utilicen. Si ocurriese algún problema, es probable que nos resulte difícil de detectar y corregir.

También dificulta el desarrollo de pruebas, ya que el uso de un Singleton implica una dependencia oculta que al momento de hacer pruebas, puede causar sorpresas.



APLICACIÓN

Singleton

- `instance: Singleton`
- `Singleton()`
- + `getInstance(): Singleton`

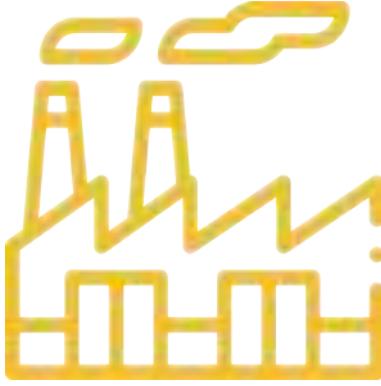
A partir del diagrama se infiere que no se podrán crear nuevas instancias de Singleton, dado que el constructor es privado. El método `getInstance` debe ser estático y será el punto de acceso para utilizar la referencia encontrada en `instance`.



UTILIDAD

Un uso común de este patrón se encuentra en componentes que quieren restringir su uso desde un solo punto:

- Parámetros de entorno o de configuración: permite tener una fuente única y rápida de información. Sólo lectura.
- Acceso a interfaces de hardware: se restringe el acceso a recursos que deben ser utilizados uno a la vez y no se pueden paralelizar.



¿En qué consiste?

Patrón creacional que **permite crear familias de objetos sin tener que especificar la clase concreta usando interfaces**. Es similar a Factory Method pero esta vez, **se crean familias o grupos de factorías** (factoría de factorías) por lo que se tienen varios métodos de creación en vez de uno solo.



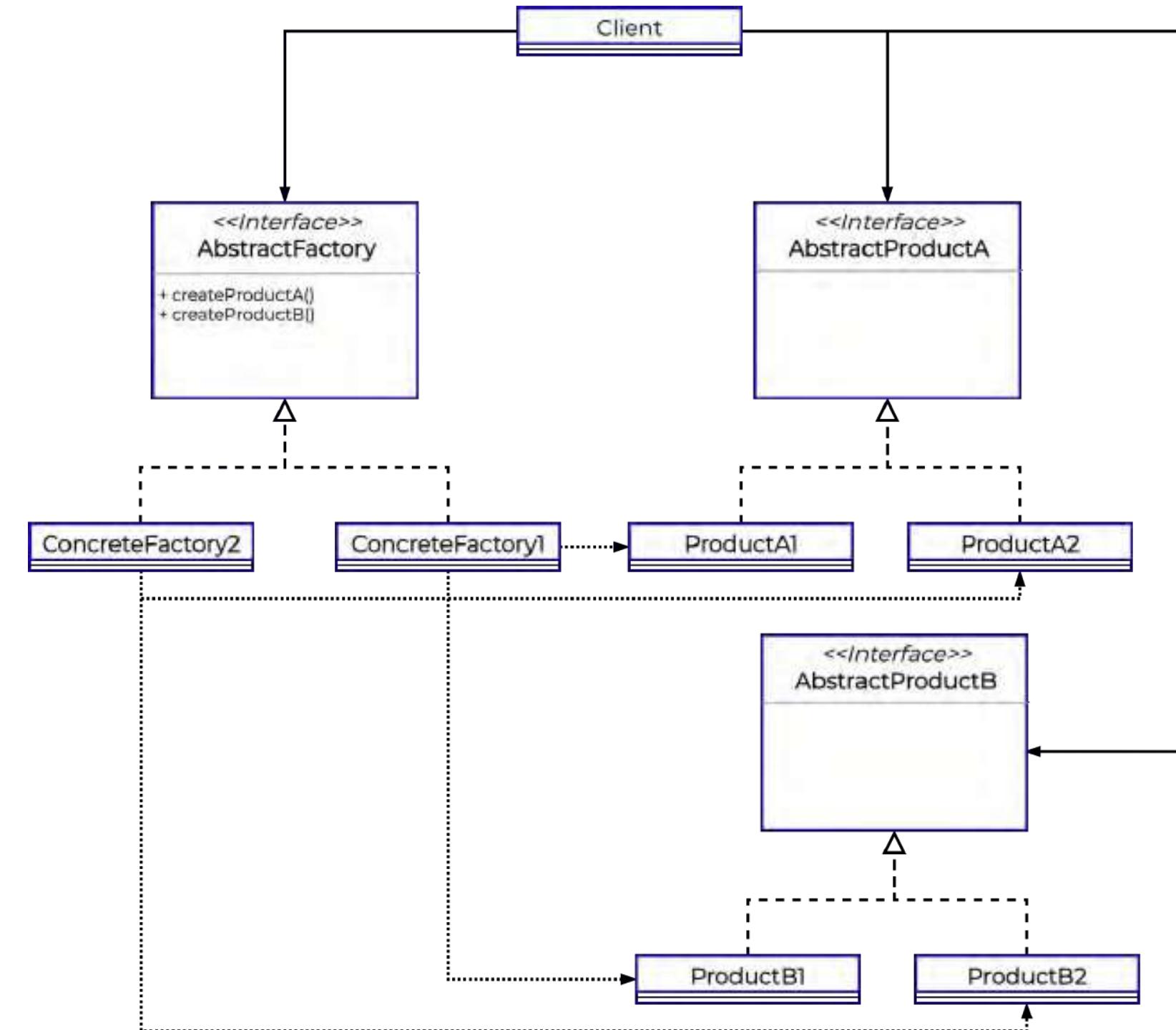
RAZONAMIENTO

Se puede aplicar Abstract Factory cuando tenemos un conjunto de **Factory methods y necesitamos trabajar con una familia de productos**.

El cliente tendrá que tratar con las factorías a través de sus respectivas interfaces. Esto permite cambiar el tipo de factoría, sin romper el contrato.

En el diagrama UML se observa cómo AbstractFactory tiene dos métodos **create()**. Aunque se puede tener tantos como productos a crear por familia se necesiten.

Este patrón evita el uso de sentencias condicionales como if o switch.





Implementación (1/2)

```
public interface CloudStorage {  
    String show();  
}  
  
public class GoogleCloudStorage implements CloudStorage {  
    @Override  
    public String show() {  
        return "show Google cloud info";  
    }  
}  
  
public class MicrosoftCloudStorage implements CloudStorage{  
    @Override  
    public String show() {  
        return "show Microsoft cloud info";  
    }  
}  
  
public interface Mail {  
    String show();  
}  
  
public class GoogleMail implements Mail {  
    @Override  
    public String show() {  
        return "show Google mail info";  
    }  
}  
  
public class MicrosoftMail implements Mail {  
    @Override  
    public String show() {  
        return "show Microsoft mail info";  
    }  
}
```



SOLUCIÓN

Tenemos **dos tipos de ‘familia’**. La familia Google y la familia Microsoft. Cada familia tiene el producto Mail y el producto CloudStorage, por lo que aplicando **polimorfismo** podemos crear implementaciones específicas para cada producto.



Implementación (2/2)



SOLUCIÓN

La interfaz **AbstractFactory** declara un conjunto de métodos para **crear los productos** (Mail y CloudStorage), pero no sabemos a qué familia pertenecen, solo queremos que mediante esta abstracción, se puedan crear productos (en general), independientemente de si son de una familia u otra.

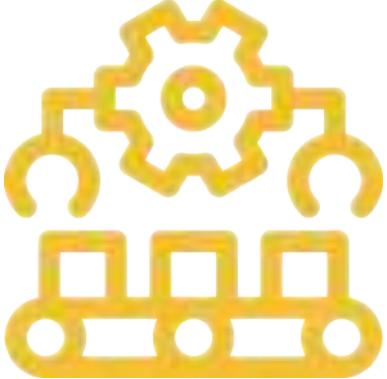
Dicho esto, la firma de los métodos debe devolver la interfaz correspondiente, de esta manera, el código no se acopla a una única implementación y el cliente se aísla de los detalles.

Se debe crear una implementación de AbstractFactory **por cada familia**.

¿Qué ventajas tiene?

- **Reduce el acoplamiento.**
- Aplica el **Principio de Responsabilidad Unica**.
- Aplica el **Principio de Abierto a la extensión y cerrado a la modificación.**

```
public interface AbstractFactory {  
    Mail createMail();  
    CloudStorage createCloudStorage();  
}  
  
public class GoogleFactory implements AbstractFactory {  
    @Override  
    public Mail createMail() {  
        return new GoogleMail();  
    }  
  
    @Override  
    public CloudStorage createCloudStorage() {  
        return new GoogleCloudStorage();  
    }  
}  
  
public class MicrosoftFactory implements AbstractFactory {  
    @Override  
    public Mail createMail() {  
        return new MicrosoftMail();  
    }  
  
    @Override  
    public CloudStorage createCloudStorage() {  
        return new MicrosoftCloudStorage();  
    }  
}
```



¿En qué consiste?

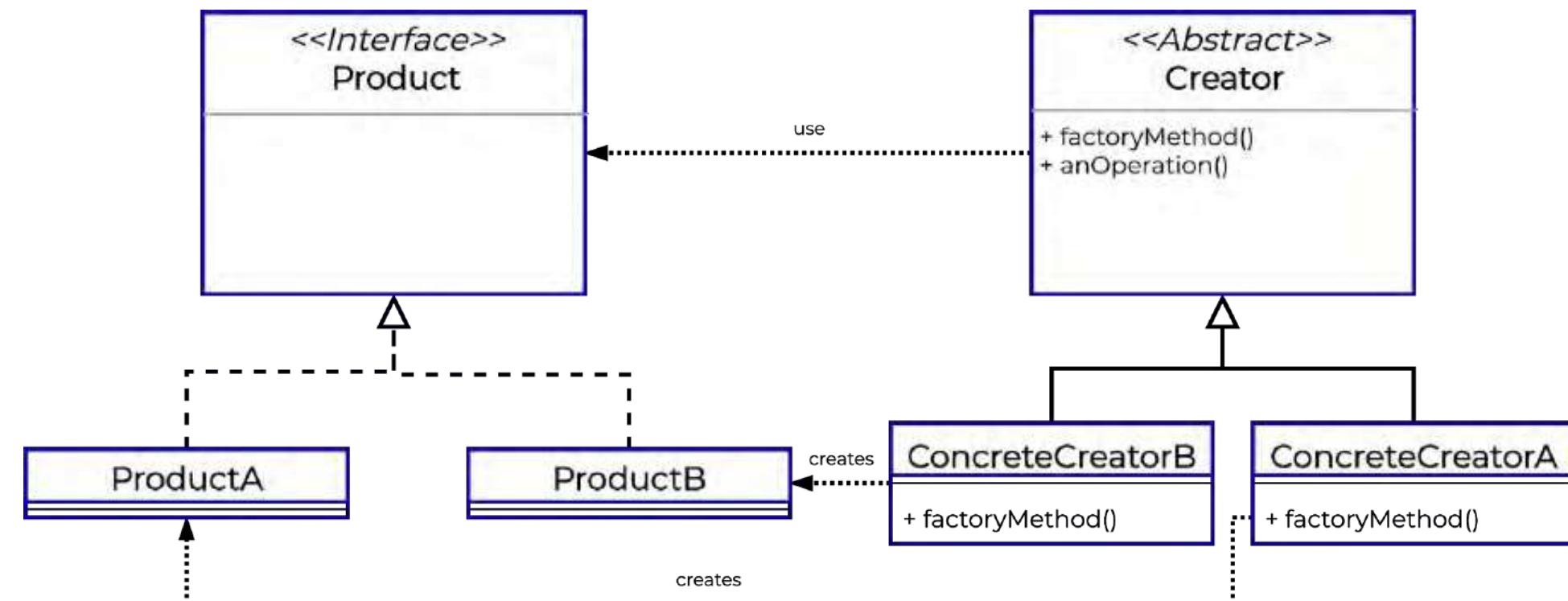
Patrón creacional que provee **una interfaz o clase abstracta**(creator) que permite encapsular la lógica de creación de los objetos en subclases. Las subclases deciden qué clase instanciar. **Los objetos se crean a partir de un método** y no a través de un constructor como se hace normalmente.

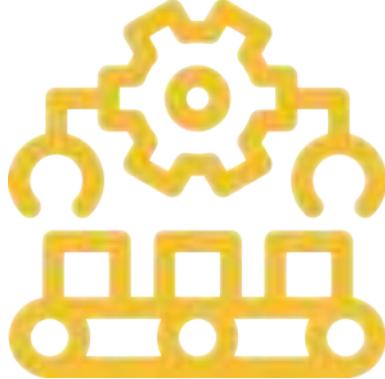


RAZONAMIENTO

Al contrario que Simple Factory, que instancia todos los objetos en la misma clase y no hay subclases, **Factory Method crea una implementación o subclase por cada producto**.

En el ejemplo, tenemos como productos distintos tipos de animales. Todos ellos implementan la interfaz **Animal** para que siempre se dependa de una abstracción y nunca de una concreción.





Implementación



SOLUCIÓN

AnimalFactory tiene un método **createAnimal (factory method)** y cada implementación se encarga de la creación de sus objetos. Podríamos incluso crear una factoría **DomesticAnimalFactory** que se encargue de crear solo aquellos animales considerados como domésticos.

Todas las factorías implementan la misma interfaz, por lo que gracias al **polimorfismo** podemos crear tantas implementaciones como necesitemos o cambiar la implementación de una factoría sin que la clase que la use se vea afectada.

¿Qué ventajas tiene?

- **Reduce el acoplamiento y encapsula** el código encargado de crear objetos.
- Aplica el **Principio de Responsabilidad Unica**.
- Aplica el **Principio de Abierto a la extensión y cerrado a la modificación**.

Se puede hacer una analogía de este patrón a la programación declarativa. Quiero una instancia del objeto X, pero, cómo se construya por debajo o cómo esté implementado no es mi responsabilidad.

```
public interface Animal {}  
  
public class Cat implements Animal {}  
  
public class Cocodrile implements Animal {}  
  
public class Dog implements Animal {}  
  
public class Lion implements Animal {}  
  
public interface AnimalFactory {  
    Animal createAnimal();  
}  
  
public class WildAnimalFactory implements AnimalFactory {  
    @Override  
    public Animal createAnimal() {  
        // creates only wild animals  
    }  
}  
  
public class RandomAnimalFactory implements AnimalFactory {  
    @Override  
    public Animal createAnimal() {  
        // creates random animals  
    }  
}
```



¿Qué es?

El patrón adaptador actúa como un **conector entre dos interfaces que son incompatibles y que no pueden estar conectadas directamente.**



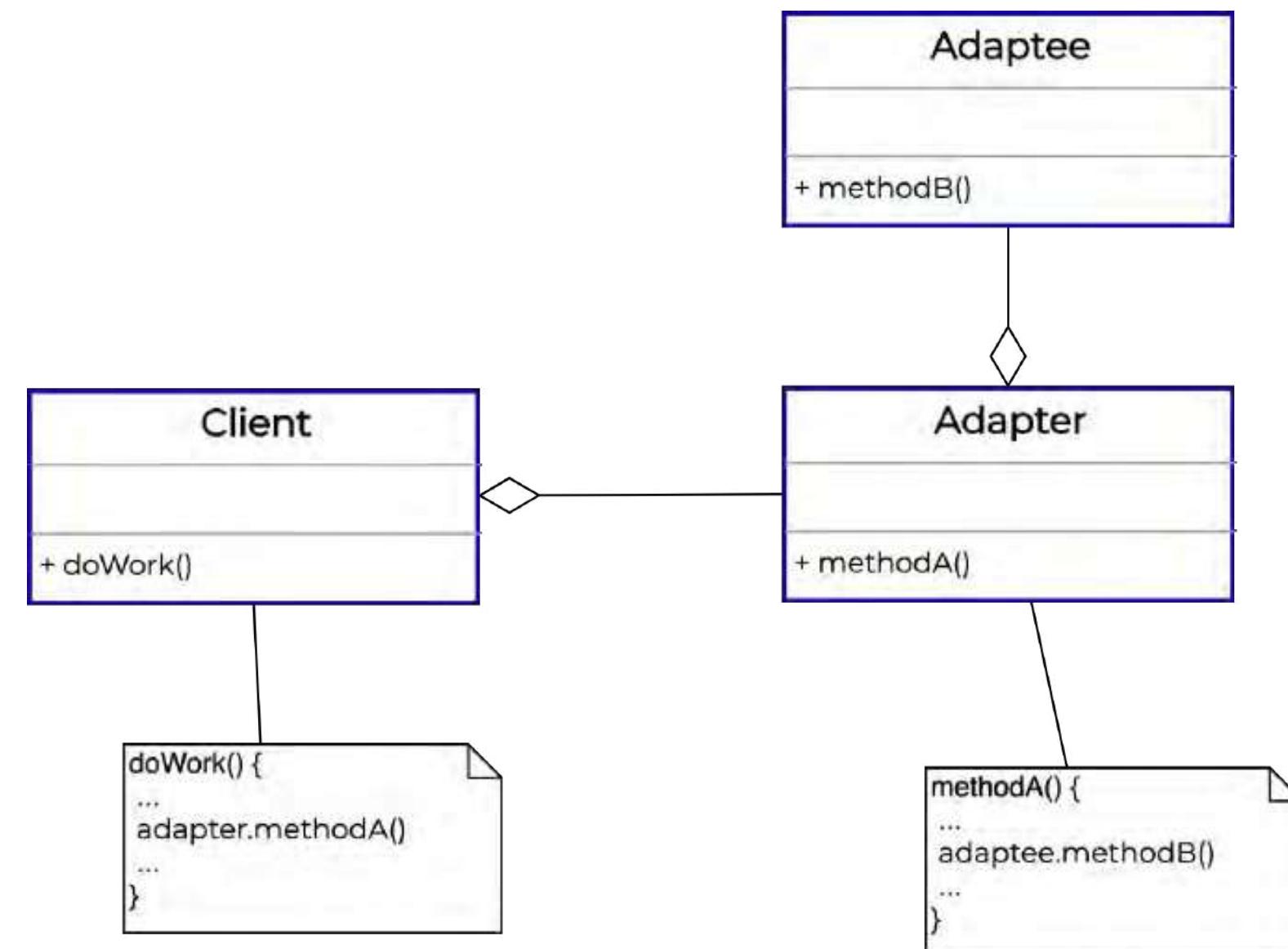
CONCEPTO

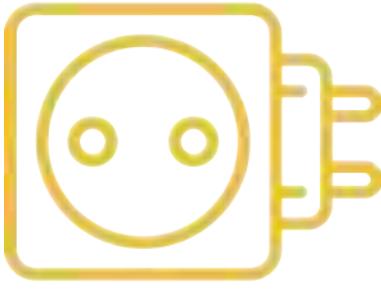
Sean dos interfaces A y B incompatibles entre sí existiendo la necesidad de que A llame a B. **Este patrón permite a través de una interfaz adaptador, envolver el contenido de la interfaz B de modo que pueda ser llamada por A.** En el diagrama de la derecha, la clase Client interactúa con la clase Adapter para poder utilizar Adaptee.

El patrón adaptador es conveniente utilizarlo cuando:

- Un componente tercero ya proporciona una funcionalidad que se puede integrar en nuestro sistema pero es incompatible.
- La aplicación no es compatible con la interfaz que espera consumir el cliente.
- Existe código *legacy* que se quiere utilizar en la aplicación sin tener que hacer ningún cambio sobre él.

Dentro del desarrollo de software este adaptador es también conocido como *wrapper*.





Implementación



EJEMPLO

Consideremos el escenario en el que tenemos una aplicación diseñada en Estados Unidos que devuelve el precio de un coche en dólares. Queremos utilizar esta funcionalidad dentro de nuestra aplicación pero los precios los tenemos que devolver en euros.

Para ello, vamos a construir una interfaz, *AdapterPricer* que envuelva a la interfaz *Pricer*, que es la que devuelve el precio en dólares.

Las implementaciones de este adaptador tendrán como dependencia la instancia de *Pricer* que corresponda. Se implementará el método *getPrice()* de *AdapterPricer* de modo que tras recuperar el precio en dólares de *Pricer* haga la conversión para devolver el precio en euros.

```
public interface AdapterPricer {
    // Return price in
    double getPrice();
}

public interface Pricer {
    // Return price in
    double getPrice();
}

public class FerrariPricer implements Pricer {
    // Return price in $
    double getPrice() {
        return 2000000;
    }
}

public interface AdapterPricer {
    // Return price in
    double getPrice();
}

public class AdapterPricerImpl implements AdapterPricer {
    private final Pricer pricer;

    //Constructor
    public AdapterPricerImpl(Pricer pricer) {
        this.pricer = pricer;
    }

    public double getPrice() {
        ...
        double priceInDolars = pricer.getPrice();
        double priceInEuro =
convertToEuros(priceInDolars);
    }

    private double convertToEuros(double price){
        //Returns price in euros
    }
}

public class Main {
    public static void main(String[] args) {
        final FerrariPricer ferrariPricer = new FerrariPricer();
        final AdapterPricer ferrariPricerAdapter = new AdapterPricerImpl(ferrariPricer);
        ...
        double ferrariEuroPrice = ferrariPricerAdapter.getPrice();
        // ferrari price in Euros!!
    }
}
```



¿En qué consiste?

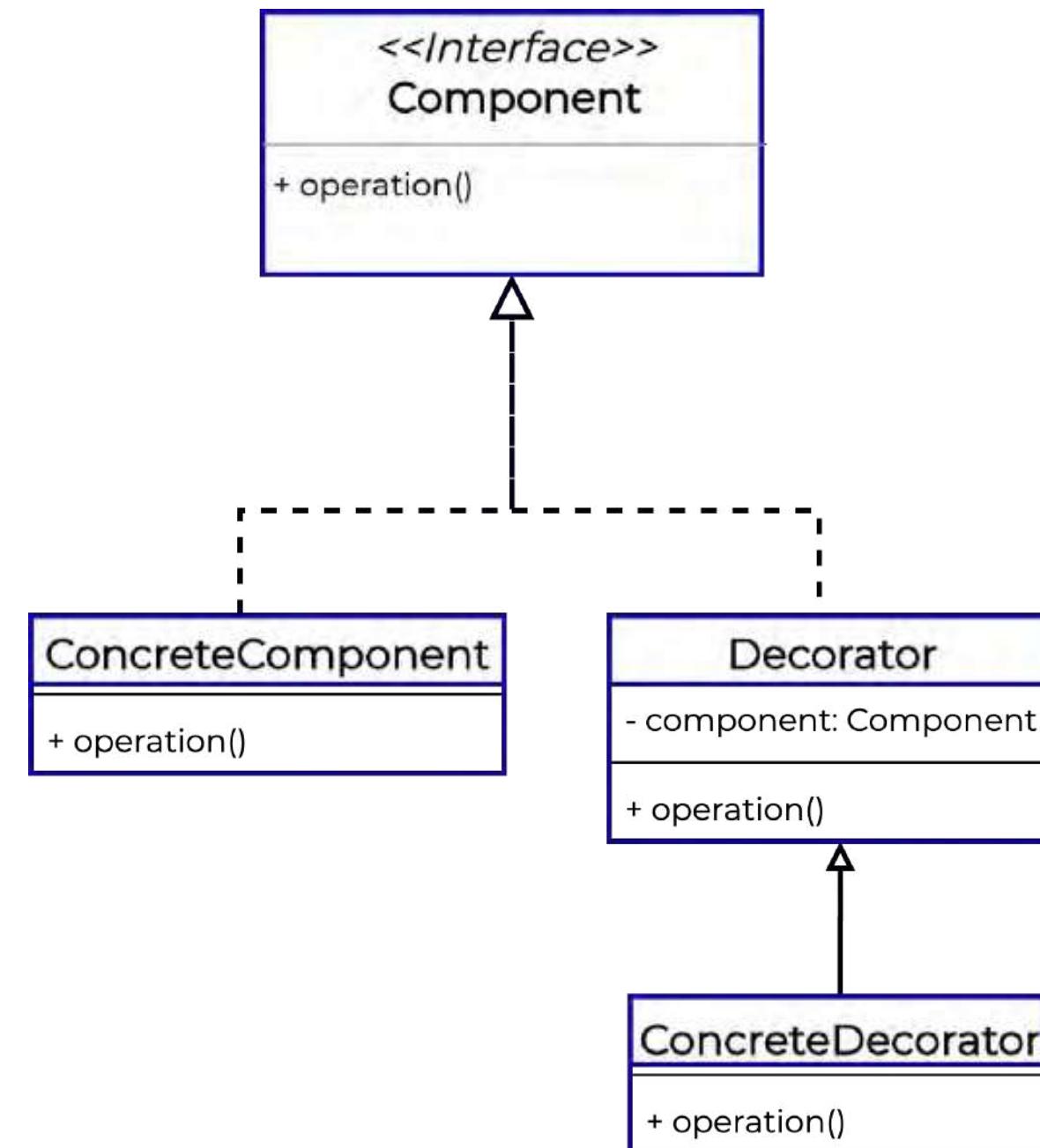
Patrón que **permite añadir nuevas funcionalidades a un objeto en tiempo de ejecución sin modificar su estructura** y a través de una envoltura (wrapper). El decorador envuelve la clase original sin cambiar la firma de los métodos existentes.



CONCEPTO

Decorator ofrece una alternativa cuando no es posible extender el comportamiento de un objeto a través de la herencia.

Normalmente tenemos una interfaz con varias implementaciones. Para aplicar este patrón, debemos crear una nueva ‘implementación’ que será nuestro *Decorator*. A partir de esta clase, creamos clases concretas de *Decorator* con las nuevas funcionalidades que se desean añadir.





Implementación

```

public interface Text {
    void write();
}

public class BaseText implements Text {
    @Override
    public void write() {
        System.out.println("some basic text");
    }
}

public class TextDecorator implements Text {
    private Text decoratedText;

    public TextDecorator(Text decoratedText) {
        this.decoratedText = decoratedText;
    }

    @Override
    public void write() {
        decoratedText.write();
    }
}

public class Main {
    public static void main(String[] args) {
        Text baseText = new BaseText();
        baseText.write();

        Text boldText = new BoldTextDecorator(baseText);
        boldText.write();

        Text italicText = new ItalicTextDecorator(baseText);
        italicText.write();

        //...
    }
}

public class BoldTextDecorator extends TextDecorator {
    public BoldTextDecorator(Text decoratedText) {
        super(decoratedText);
    }

    @Override
    public void write() {
        System.out.println("adding bold style to text");
        super.write();
    }
}

/*
Another class for ItalicTextDecorator...
Another class for UnderlineTextDecorator...
*/

```

SOLUCIÓN

Definimos la interfaz *Text* con su implementación *BaseText* que define un comportamiento básico que podrá ser modificado por un decorator. Importante que la clase *TextDecorator* tenga un campo de tipo *Text* ya que será el que envolvamos para su posterior modificación. *BoldTextDecorador*, *Italic* y *Underline* definen el comportamiento que se va a añadir dinámicamente al componente, en nuestro caso, a *BaseText*. Desde la perspectiva del cliente, los objetos son iguales.

Ventajas de usar este patrón:

- **Añade o elimina funcionalidades de forma flexible** a un objeto en tiempo de ejecución.
- Sigue el **principio Open/Closed**.
- Permite envolver un objeto en varios decorators.



¿En qué consiste?

Es un patrón que busca **desacoplar la abstracción de su implementación**. Permite que la abstracción y la implementación se desarrollen de forma independiente y el código del cliente solo puede acceder a la abstracción sin preocuparse por la parte de implementación. Se puede pensar en una **abstracción con dos capas**.



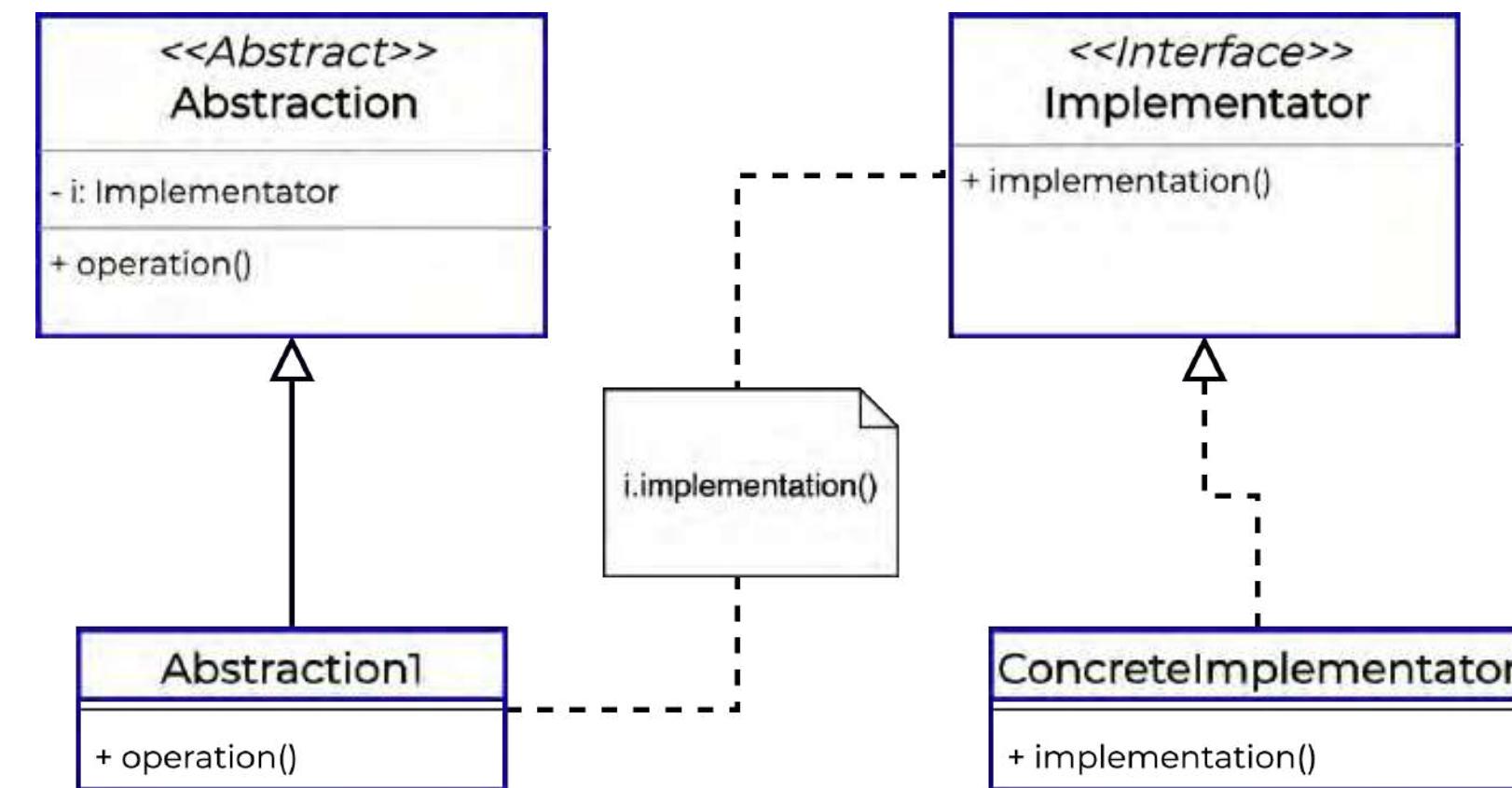
CONCEPTO

Este patrón normalmente define una abstracción y son las implementaciones las que cambian, pero hay casos en los que desde un principio, no estamos seguros del alcance de la abstracción y sabemos que va a ir evolucionando. Para poder **hacer cambios a la abstracción, sin estar rompiendo las implementaciones continuamente**, usamos este patrón.

Este patrón es útil en las siguientes situaciones:

- Queremos que una abstracción y su implementación se definan y extiendan de forma independiente.
- Queremos desacoplar la abstracción y su implementación para poder cambiarla en ejecución.

En la figura se ve la clase *Abstraction*, que tendrá una referencia a *Implementator*. El método *operation()* de *Abstraction1* puede trabajar con distintas implementaciones de la interfaz *Implementator*.





Implementación



EJEMPLO

Implementamos la clase abstracta **RemoteControl** a la cual podemos injectar una **TV**. La **TV**, en nuestro caso, será el implementador del patrón **Bridge**.

El modo en el que injectamos el implementador no es relevante para el patrón. Si necesitamos cambiar la implementación en **runtime** podríamos hacerlo mediante un setter en vez de utilizar el constructor.

Tenemos un **RemoteControl** específico llamado **ModernRemoteControl** que tiene la posibilidad adicional de cambiar de canal a la posición siguiente o anterior.

Como **TV** podemos tener una digital y otra analógica. De esta forma tenemos **dos jerarquías totalmente independientes** y podemos separar la evolución de **RemoteControl** de las implementaciones concretas de **TV** para las que se utilizan.

```
public abstract class RemoteControl {
    private final TV implementator;
    private final int currentChannel = 0;

    public RemoteControl(TV implementator) {
        this.implementator = implementator;
    }

    protected final void setChannel(int channel) {
        implementator.setChannel(channel);
        this.currentChannel = channel;
    }

    protected final int getCurrentChannel() {
        return this.currentChannel;
    }
}
```

```
public interface TV {
    setChannel();
}

public class DigitalTV implements TV {
    @Override
    protected final setChannel(int channel) {
        // Sintoniza de forma digital
    }
}

public class AnalogTV implements TV {
    @Override
    protected final setChannel(int channel) {
        // Sintoniza de forma analógica
    }
}
```

```
public class ModernRemoteControl extends RemoteControl {

    public ModernRemoteControl(TV implementator) {
        super(implementator);
    }

    public void previousChannel(int channel) {
        this.setChannel(this.getCurrentChannel() - 1);
    }

    public void nextChannel(int channel) {
        this.setChannel(this.getCurrentChannel() + 1);
    }
}
```

```
public static void main(String[] args) {
    TV implementator = new DigitalTV();
    RemoteControl remoteControl = new ModernRemoteControl(implementator);
    remoteControl.setChannel(5);
}
```



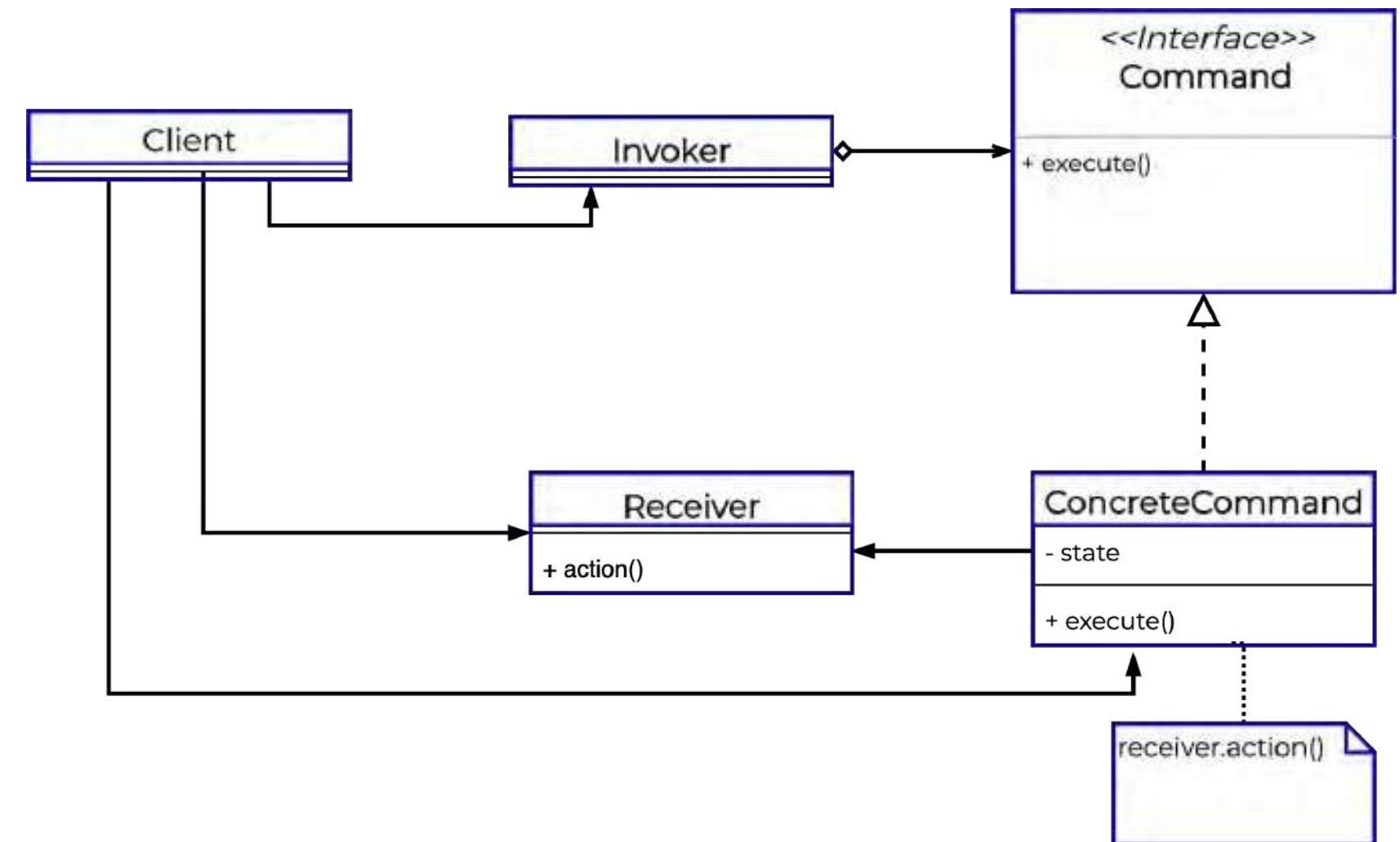
¿En qué consiste?

Patrón que **permite encapsular una petición en un objeto** que contiene toda la información necesaria para realizar una acción (un comando).



CARACTERÍSTICAS

- **Command:** objeto que contiene toda la información necesaria para realizar una acción específica.
- **Receiver:** objeto que realiza las operaciones cuando se ejecuta el comando.
- **Invoker:** objeto encargado de ejecutar la acción, pero desconoce la implementación del comando. Él únicamente recibe la interfaz y llama al método execute, aunque también se encarga de gestionar una cola de comandos (en caso de que se necesite) o de realizar la acción de revertir (si fuera necesario). Invoker actúa como mediador y permite desacoplar a los consumidores del objeto comando.
- **Client:** objeto que controla el proceso de ejecución de los comandos. Instancia los comandos deseados y se los pasa al Invoker.





Implementación (1/2)



SOLUCIÓN

Se han creado solo dos clases como *Receivers* para simplificar el ejemplo, pero se podrían haber creado más como *Television*, *Radio*, *Heater*, *Cooker* etc., cada una con sus respectivas acciones. Las clases *command* tienen como atributo su respectivo receiver y se encargan de ejecutar y revertir la acciones pertinentes.

Se podría también, tener una clase *TurnOffAllDevicesCommand* donde a través del constructor nos llegue una lista de electrodomésticos y se ejecute la acción de apagarlos todos.

Una pequeña desventaja que se puede observar es el incremento del número de clases por comando.

```
//Receiver
public class Speaker {
    public String turnUpVolume() {
        return "turning up volume...";
    }

    public String turnDownVolume() {
        return "turning down volume...";
    }
}

public class VolumeDownSpeakerCommand
implements Command {
    private Speaker speaker;

    //Constructor

    @Override
    public void execute() {
        speaker.turnDownVolume();
    }

    @Override
    public void undo() {
        speaker.turnUpVolume();
    }
}
//Similar para clase AirConditionOnCommand
```

```
public interface Command {
    void execute();
    void undo();
}
```

```
//Receiver
public class AirCondition {
    public String on() {
        return "A.C is on";
    }

    public String off() {
        return "A.C is off";
    }
}
```

```
public class VolumeUpSpeakerCommand
implements Command {
    private Speaker speaker;

    //Constructor

    @Override
    public void execute() {
        speaker.turnUpVolume();
    }

    @Override
    public void undo() {
        speaker.turnDownVolume();
    }
}
// similar para clase AirConditionOffCommand
```



Implementación (2/2)



SOLUCIÓN

La clase *RemoteControl* no sabe la implementación del comando que se quiere ejecutar, solo conoce su interfaz.

El cliente será el encargado de instanciar los comandos pertinentes, ya sean tanto para Speaker como para AirCondition, y se los pasará al *Invoker* (*RemoteControl*).

Algunas ventajas del patrón:

- Cumple con **SRP (Single Responsibility Principle)**. Desacoplamos las clases que invocan las acciones de las que las ejecutan.
- Cumple con el **principio Open Closed**. Se puede introducir tantos comandos como se necesiten.
- Se pueden gestionar operaciones de **reversión o encolado de comandos** (en caso de necesitarlo).

```
//Invoker
public class RemoteControl {
    private Command command;

    public RemoteControl(Command command) {
        this.command = command;
    }

    public void executeOperation(){
        command.execute();
    }

    public void undoOperation(){
        command.undo();
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Speaker speaker = new Speaker();
        Command volumeUpSpeaker = new VolumeUpSpeakerCommand(speaker);
        RemoteControl remoteControl = new RemoteControl(volumeUpSpeaker);

        remoteControl.executeOperation();
        remoteControl.undoOperation();

        //...
    }
}
```



Delegando como se debe

Este patrón permite delegar una operación a lo largo de una cadena de objetos hasta encontrar al eslabón que pueda realizar la acción. Su diseño permite la generación de cadenas dinámicas durante la ejecución del programa, cambiando el comportamiento del componente.



DISEÑO

El diseño de este patrón consiste en definir:

1. Uno o más **eslabones**, o *handlers*. Cada uno tendrá un sólo comportamiento que se ejecutará si las condiciones se cumplen. En caso de no cumplirse, el eslabón tendrá una referencia al siguiente para delegarlo.
2. Un **cliente** inicia la cadena de ejecución.

La construcción de la cadena se puede hacer dentro del cliente o fuera de él, en un componente externo.

La flexibilidad a la hora de generar la cadena permite incluir y/o excluir ciertas condiciones mientras la aplicación se ejecuta.

Podemos controlar el tamaño de la cadena en base a criterios externos (el tipo de mensaje o el tipo de operación que se quiere realizar), abriendo el camino para **reutilizar** los eslabones en otros componentes y/o cadenas.



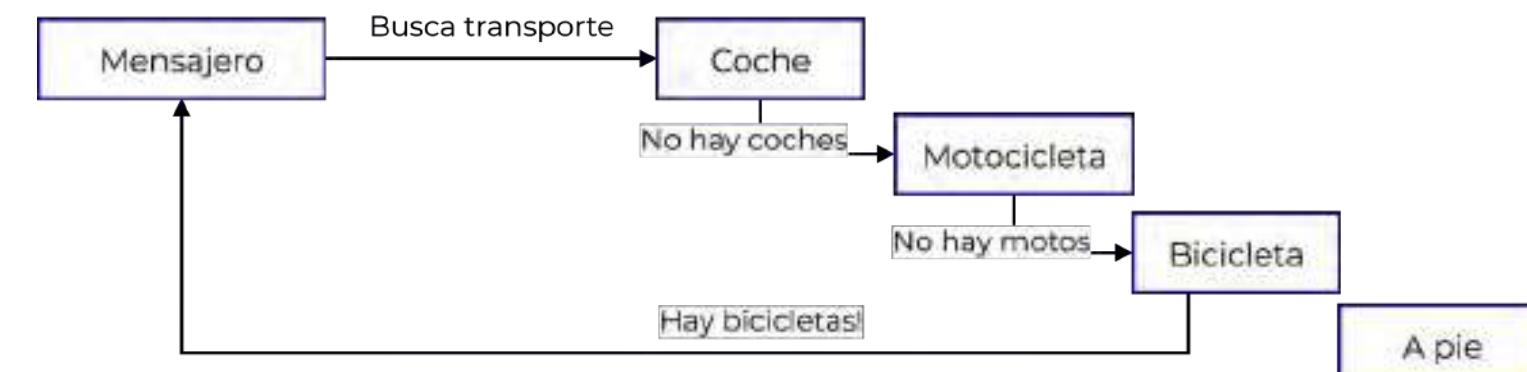
APLICACIÓN

Queremos determinar el medio de transporte para enviar un paquete dentro de una ciudad. Si un medio de transporte no está disponible, se consulta al siguiente. La cadena de prioridad sería la siguiente:

Coche → Motocicleta → Bicicleta → A pie

Obviando la viabilidad del medio de transporte en relación a la distancia, lo que se tiene es una cadena de responsabilidades. Cada eslabón de la cadena sabrá si su medio de transporte se encuentra disponible.

El mensajero utilizará el medio de transporte del primer eslabón que acepte la responsabilidad.





¿Qué es?

El patrón estrategia define una familia de algoritmos, quedando encapsulados y pudiendo intercambiarse entre ellos. Los algoritmos quedan independientes de los clientes que los usan.



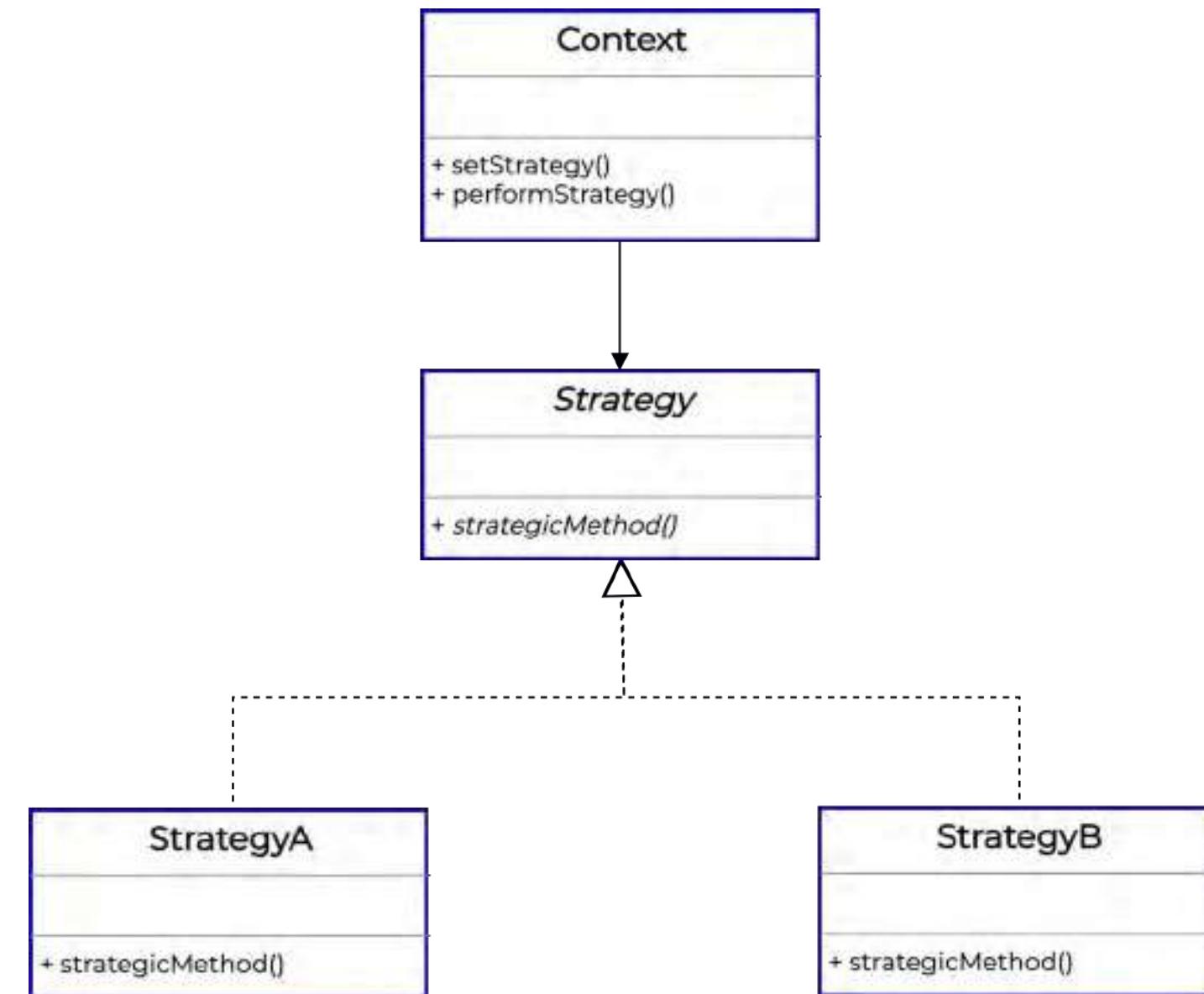
CONCEPTO

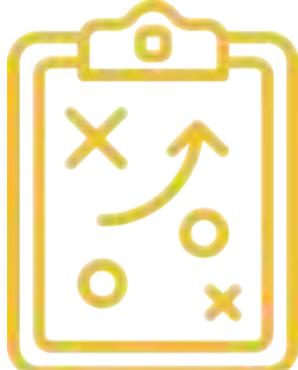
Este patrón resulta útil cuando tenemos un problema que se puede resolver de varias formas y queremos la libertad de elegir la forma de hacerlo en ejecución.

Se **define una familia de algoritmos (o estrategias) y el objeto cliente elige entre ellos**. Puede haber cualquier número de estrategias y todas implementan la misma interfaz, por lo que son intercambiables entre sí, incluso en tiempo de ejecución.

De este modo, si hay información del algoritmo que los clientes no deberían saber, ésta va a quedar encapsulada fuera del código del cliente.

Este patrón es compatible con el principio abierto/cerrado (**OCP**), que propone que las clases deben estar abiertas para extensión y cerradas para modificación. Se añade comportamiento creando nuevas estrategias pero no se modifica el código existente.





Implementación

```

interface PathfindingStrategy {
    public void find();
}

public class AStarAlgorithm implements PathfindingStrategy {
    public void find() {
        System.out.println("Has usado A*");
    }
}

public class DijkstraAlgorithm implements PathfindingStrategy {
    public void find() {
        System.out.println("Has usado Dijkstra");
    }
}

public class Main {
    public static void main(String args[]) {
        // Usamos el algoritmo A*
        PathfindingStrategy aStar = new AStarAlgorithm();
        Context context = new Context(aStar);
        context.performTask();
        // Usamos el algoritmo de Dijkstra
        PathfindingStrategy dijkstra = new DijkstraAlgorithm();
        context.setStrategy(dijkstra);
        context.performTask();
        // Volvemos al algoritmo A*
        context.setStrategy(aStar);
        context.performTask();
    }
}

```

```

public class Context {
    PathfindingStrategy c;

    public Context(Strategy c){
        this.c = c;
    }

    public void setStrategy(Strategy c)
    {
        this.c = c;
    }

    public void performTask(){
        c.find();
    }
}

```

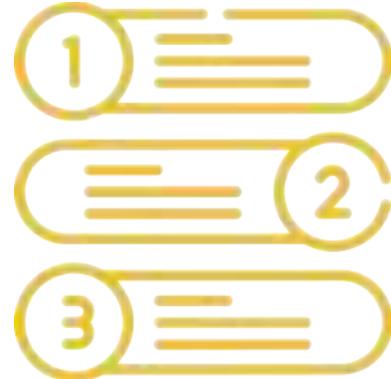


EJEMPLO

Queremos encontrar la ruta para llegar de un punto a otro utilizando los algoritmos de búsqueda de caminos más famosos. Mediante el patrón estrategia vamos a crear la interfaz PathfindingStrategy que en el método **find()** va a hacer los cálculos oportunos para encontrar una ruta.

Tenemos dos implementaciones que extienden la interfaz con los algoritmos más reconocidos para resolver este problema: A* y Dijkstra.

La clase de **Contexto es el cliente de la estrategia**. La estrategia se puede cambiar en cualquier punto de la ejecución, gracias a que tiene un setter disponible.



¿En qué consiste?

Este patrón permite **definir el esqueleto de un algoritmo** para luego implementar los detalles del mismo mediante herencia, sin cambiar la estructura del algoritmo.



CARACTERÍSTICAS

El patrón Template Method consiste en definir los pasos de un algoritmo y permitir que **las subclases proporcionen la implementación para uno o más pasos**. De este modo, se pueden definir algunos pasos del algoritmo pero mantener su estructura.

El algoritmo **va a ser un método que dentro invoca a otros métodos en un orden determinado** que son los pasos del algoritmo. Estos pasos serán métodos abstractos que las subclases van a implementar.

Con esto el código está en un solo sitio y además, está protegido dentro de la clase. Añadir nuevas implementaciones del algoritmo solo requiere implementar las operaciones del algoritmo.

También puede haber métodos que no hagan nada, pero que una subclase pueda darles una implementación. A estos métodos se los llama *hooks*.



TEMPLATE METHOD VS STRATEGY

Ambos patrones son usados para encapsular algoritmos, uno mediante herencia y otro mediante composición.

Con el patrón Template Method se define el esqueleto del algoritmo pero se deja a las subclases parte de la responsabilidad. Por otro lado, con el patrón Strategy se define una familia de algoritmos que pueden tener estructuras distintas y se hacen intercambiables en tiempo de ejecución.

```
abstract class AbstractClass {  
    final void templateMethod() {  
        step1();  
        step2();  
        concreteOperation();  
        hook();  
    }  
    abstract void step1();  
    abstract void step2();  
  
    final void concreteOperation() {  
        // implementation here...  
    }  
    void hook() {}  
}
```



Reaccionando a cambios de estado

El patrón observador (*observer* en inglés) describe una solución que se asemeja al manejo de eventos. Principalmente es utilizado para permitir que ciertos objetos puedan reaccionar a los cambios que suceden en un momento dado en otros objetos.



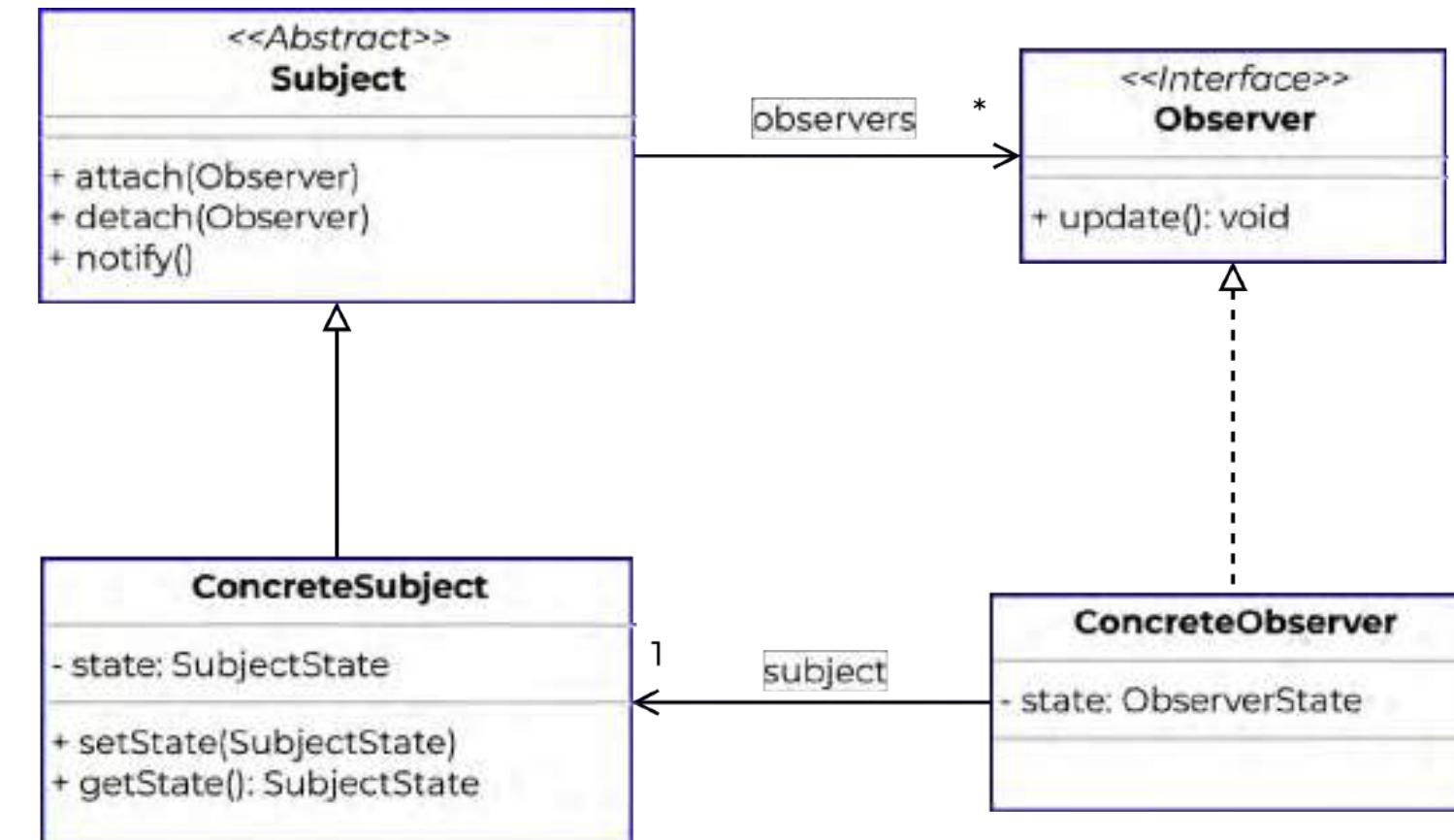
CONCEPTO

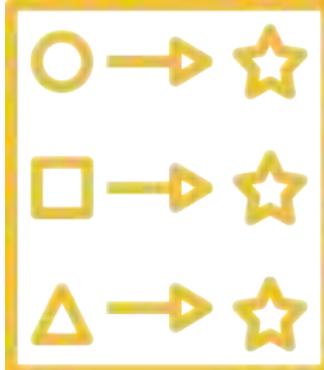
Dado el diagrama, tenemos una clase abstracta, **Subject**, que implementa una serie de métodos para enlazar observadores a la clase. Cuando el estado del subject cambie, notificará a sus observadores invocando el método *update* en cada una de ellas.

La interfaz **Observer** expone un solo método, cuya invocación dependerá del Subject. La lógica de este método es una reacción al cambio sucedido en Subject.

Las clases **ConcreteSubject** y **ConcreteObserver** denotan una manera de implementar el patrón. Se observa que el **ConcreteObserver** tiene una referencia al **ConcreteSubject**, lo cual le permite obtener su estado. De esta manera, cuando el **Subject** notifique a sus **Observers**, esta implementación en particular, tendrá una referencia directa al **Subject** para poder reaccionar en base a los cambios sucedidos.

Esta no es la única forma de diseñar el patrón Observer, pero si es la clásica descrita por el GoF (Gang of Four).





División del comportamiento con estados

El patrón Estado (o *State*, en inglés) utiliza clases para representar el comportamiento de un objeto en función de su estado.



DISEÑO

El diseño de este patrón consiste primero en definir dos grupos de clases: el contexto y los estados.

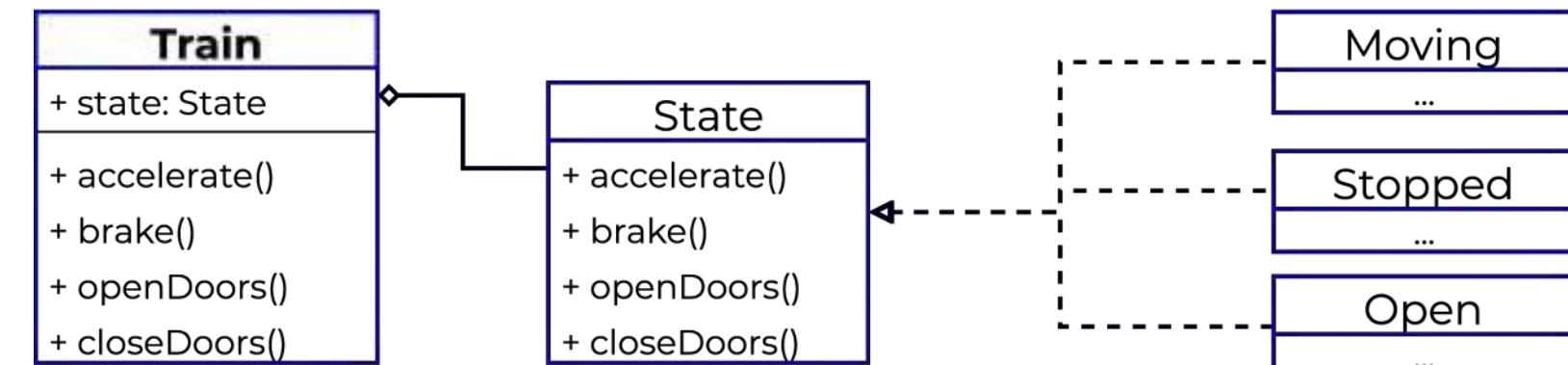
- El contexto **representa el estado actual**. Cuando un cliente invoque los métodos del contexto, se utilizará el comportamiento del estado asociado a él.
- Los estados son objetos que implementan una misma interfaz utilizada por el contexto. Cada estado contiene lógica de acuerdo a lo que representa.

También hay que considerar las **transiciones de estado**. Sin transiciones, no habría cambio de comportamiento. Si las transiciones se definen dentro de los estados, deberán modificar el estado actual del contexto.



APLICACIÓN

Supongamos que tenemos un tren, y este tren está siempre encendido:



1. Podemos **acelerar** el tren. Si el estado actual es **Detenido**, se pasa a **En Marcha**. No se puede acelerar si el tren está **Abierto**.
2. Podemos **frenar** el tren. Si el estado actual es En Marcha y la velocidad del tren llega a 0, el estado pasa a Detenido.
3. Cuando está En marcha, el tren **no podrá abrir o cerrar sus puertas**.
4. Cuando esté Detenido, **podrá abrir las puertas**. En ese caso, pasaría al estado **Abierto**.
5. Del estado Abierto se podrán cerrar las puertas y volvería al estado Detenido.

Con este patrón se pretende que el comportamiento del Contexto (el tren) varíe cuando su estado interno cambie: el objeto Tren se comportará en base a su estado actual.



Comportamiento - Visitor

¿Qué es?

Dada una estructura de objetos compuesta, este patrón **permite añadir funcionalidad sin necesidad de cambiar las clases de los elementos en los que va a ejecutarse.**



CONCEPTO

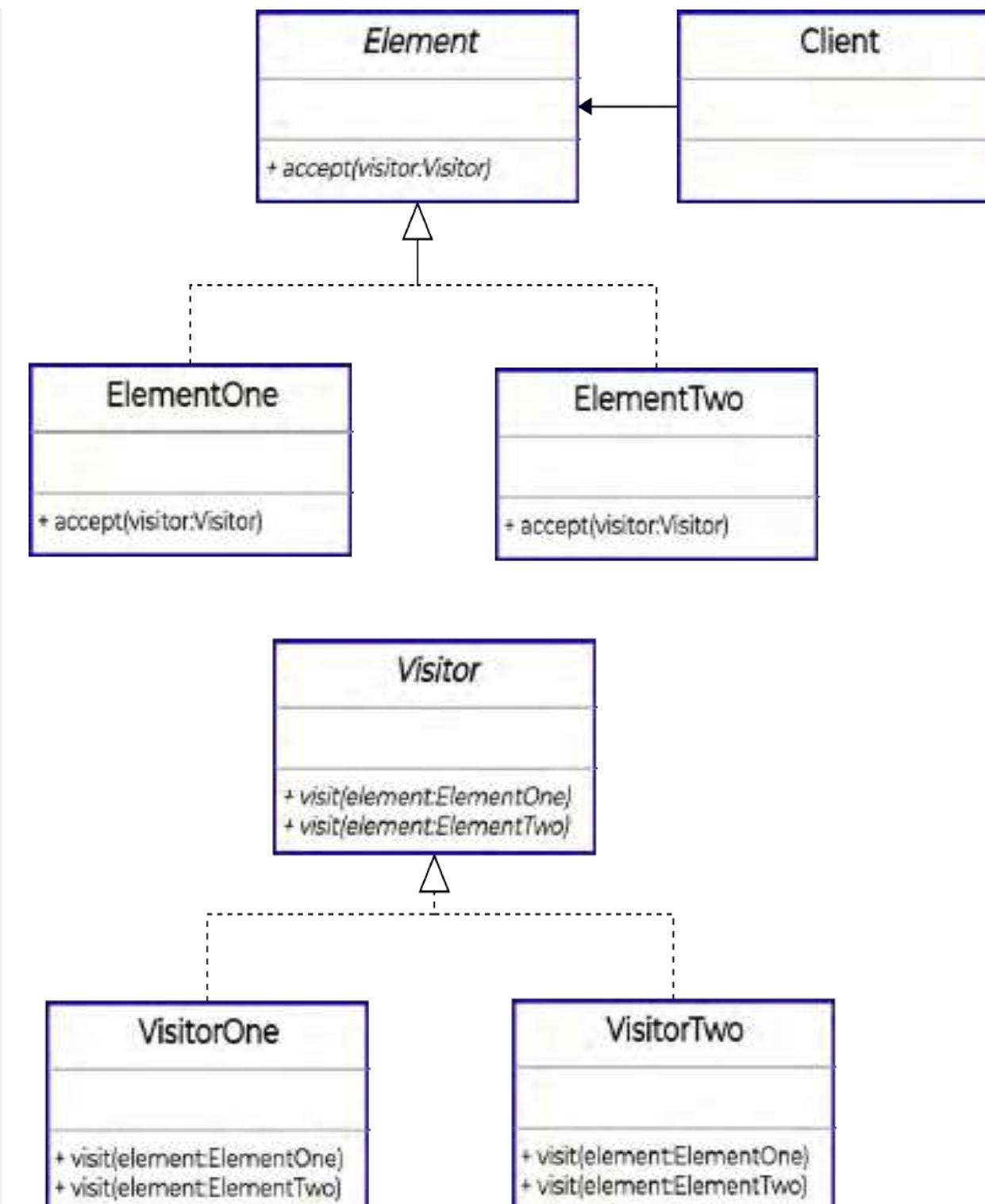
El patrón Visitor permite extender el comportamiento de las clases de una estructura de objetos, creando una jerarquía de clases separada. Estas nuevas clases son del tipo Visitor y recogen el nuevo comportamiento; **las clases cliente solo tienen que "aceptar" al visitante para delegarle las operaciones.**

Como se ve en la figura, la clase ElementOne no implementa la operación directamente, si no que con el método `accept(visitor)` delega la operación al objeto visitante (que invoca su método `visit()`). Si se quiere añadir una nueva operación, se añade una nueva clase que implemente Visitor, sin tener que modificar ElementOne o ElementTwo.

La implementación del método `visit()` usada, depende tanto del tipo del elemento como del tipo del Visitor. Esto es lo que se conoce como “double-dispatch”.

Este patrón resulta muy útil en los siguientes casos:

- Cuando necesitamos añadir nuevas operaciones frecuentemente.
- Tenemos un mismo algoritmo y queremos que funcione en distintas jerarquías de clases y se encuentre en un solo lugar.
- La estructura de objetos cliente no esperamos que cambie o no es nuestra.





¿En qué consiste?

Provee una forma de **acceder secuencialmente a los elementos de una colección sin necesidad de exponer su representación interna.**



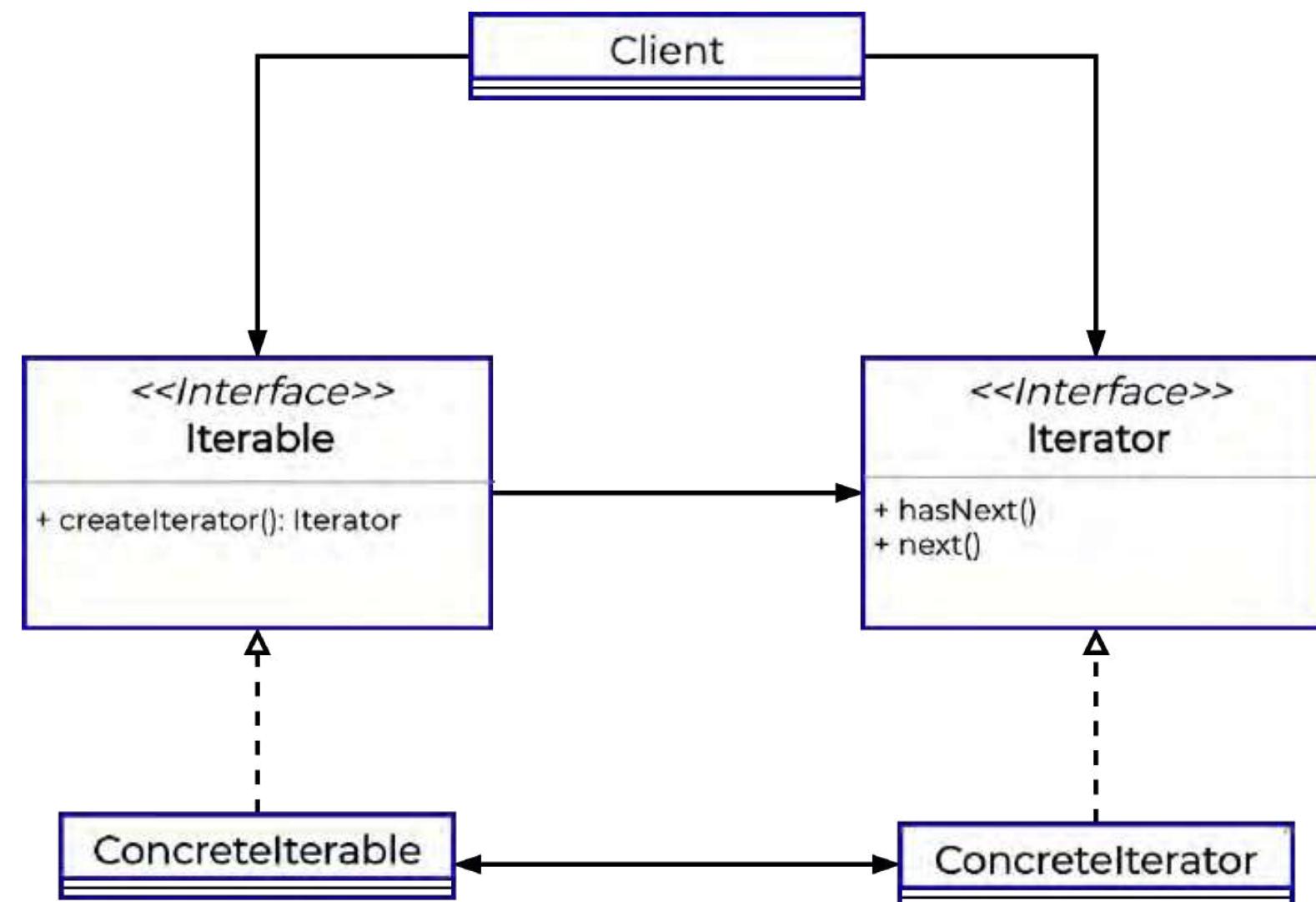
CONCEPTO

La idea principal es la de extraer el comportamiento de una colección en un objeto llamado *Iterator* que se encargará de tener toda la información necesaria para manipular la colección.

Se debe declarar una interfaz *Iterator* con los métodos necesarios (*hasNext*, *next*, *currentItem*, *first*, *last*, etc). Podemos definir distintas implementaciones una por cada algoritmo de recorrido que necesitemos.

Definimos también una interfaz *Iterable* que define un método para crear un iterator. Importante que el método devuelva la interfaz *Iterator* para no acoplarnos a una única implementación y depender de abstracciones. *Concretelterable* devuelve nuevas instancias de un iterator en concreto

El cliente al final está siempre trabajando con abstracciones a través de sus interfaces. Esto le permite hacer uso de varios tipos de colecciones e iteradores con el mismo código.





Implementación (1/2)



SOLUCIÓN

Definimos la interfaz Iterator con dos métodos (se podrían añadir más) y su correspondiente implementación. *MessageIterator* debería implementar el algoritmo específico para recorrer esa colección, además de otra lógica, como tener una propiedad que se encargue de gestionar la posición actual de iterator sobre el array/coleccion/lista, etc.

La interfaz Iterable define un método para instanciar iterator. En esta clase podríamos tener una lista con su respectivo método que vaya añadiendo nuevos mensajes y al instanciar *MessageIterator*, se podría pasar dicha lista por parámetro para que pueda ser recorrida posteriormente.

```
public class Message{  
    //...  
}  
  
public interface Iterable {  
    Iterator createIterator();  
}  
  
public class MessageIterable  
implements Iterable {  
    //...  
  
    @Override  
    public Iterator createIterator(){  
        return new  
MessageIterator(messageList);  
    }  
}  
  
public interface Iterator {  
    Boolean hasNext();  
    Object next();  
}  
  
public class MessageIterator implements Iterator{  
    private Message[] messageList  
    private int currentPosition;  
    //constructor  
  
    @Override  
    public Boolean hasNext() {  
        if(position >= messageList.length){  
            return false  
        }else {true}  
    }  
  
    @Override  
    public Object next() {  
        Message nextItem = messageList[currentPosition]  
        currentPosition += 1;  
        return nextItem;  
    }  
}
```



Implementación (2/2)



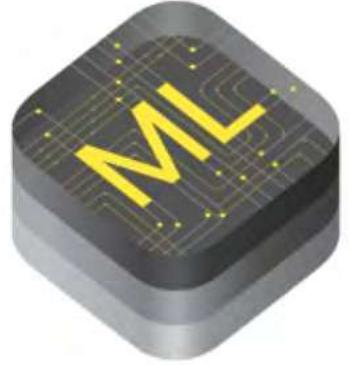
SOLUCIÓN

NotificationScreen será la clase encargada de imprimir los mensajes correspondientes usando los métodos de la clase iterator.

La clase main simplemente define que tipo de iterable desea usar. En este ejemplo, solo tenemos una implementación (MessageIterable) pero en un ejemplo real podríamos tener varios tipos.

```
public class NotificationScreen {  
  
    private Iterable messages;  
  
    public NotificationScreen(Iterable messages) {  
        this.messages = messages;  
    }  
  
    public void showMessages() {  
        Iterator iterator = messages.createIterator();  
        while (iterator.hasNext()) {  
            System.out.println("next item: " + iterator.next());  
            //...  
        }  
    }  
  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Iterable messages = new MessageIterable();  
        NotificationScreen notificationScreen = new NotificationScreen(messages);  
        notificationScreen.showMessages();  
    }  
}
```

ios



Definición

Es un framework de Apple para el desarrollo de **Machine Learning** en apps nativas. Con posibilidad de entrenamiento en el dispositivo. Xcode viene acompañado de la herramienta **CreateML** para generar los modelos y entrenarlos sin una sola línea de código.



VENTAJAS DE NEGOCIO

Conocimiento profundo del cliente, por ejemplo:

- Categorizar los clientes en función de su comportamiento para una correcta segmentación.
- Conocer y predecir la Navegación dentro de nuestra app para saber dónde hacer foco.
- Conocer y predecir las búsquedas del cliente concreto para ofrecerle de manera activa los productos demandados.

Recomendar los productos correctos:

- Conocer patrones ocultos en el comportamiento de un cliente para ofertar productos que le han interesado a otros clientes de comportamiento similar.
- Hacer estrategias de ventas y marketing basadas en patrones de comportamiento general o individual.



VENTAJAS DE DESARROLLO

- **Soporte desde iOS 11** y entrenamiento “on-device” desde iOS 13.
- **Rápida implementación** con una API simple y clara.
- CoreML es un framework **nativo de Apple**, lo que nos garantiza su continuidad y calidad.
- Mediante la herramienta de Xcode CreateML **podemos crear de forma rápida y sin código nuestros modelos y entrenarlos**.
- Mediante coreMLtools **podemos transformar modelos hechos en otras plataformas al formato de Apple**.

```
extensión MyModel {  
    func predict(_ value: MLFeatureValue) -> String {  
        guard let pixelBuffer = value.imageBufferValue,  
              let prediction = try? prediction(image:  
pixelBuffer).classLabel else {  
            return "unknown"  
        }  
        return prediction  
    }  
}
```



Definición

Es un framework de Apple para la construcción de **interfaces visuales** de forma declarativa, reduciendo la complejidad del código y optimizado para aprovechar las funciones nativas del sistema. Disponible desde macOS Catalina 10.15, iOS 13, iPadOS 13, tvOS 13 y watchOS 6.



VENTAJAS

- Sintaxis declarativa y programación reactiva.
- Diseño bidireccional con las previews: añades un objeto a la interfaz y se actualiza el código y añades código y se actualiza la interfaz.
- Con las Previews puedes ejecutar tu app en un dispositivo Apple conectado y hacer cambios en tiempo real.
- Soportado en todas las plataformas (iOS, iPadOS, macOS, watchOS y tvOS).
- Animaciones complejas con poca codificación.
- Favorece la implementación de una arquitectura limpia con MVVM o VIPER.
- Se puede mezclar con UIKit sin problema.
- Prescinde de Interface Builder y Autolayout.
- Es el futuro de la construcción de interfaces, tanto Apple como la comunidad apuesta fuertemente por esta tecnología.



EJEMPLO

```
struct MasterView: View {  
    @State private var showPopover: Bool = false  
  
    var body: some View {  
        VStack {  
            Button("Show popover") {  
                self.showPopover = true  
            }.popover( isPresented: self.$showPopover) {  
                Text("Popover")  
                    .bold()  
                    .underline()  
                    .foregroundColor(.red)  
            }  
        }  
    }  
}
```



¿Qué es?

Es un lenguaje de programación desarrollado por Apple para ser utilizado en sus plataformas con la intención de ser una alternativa a Objective-C. Inicialmente propietario pero liberado como código abierto en su versión 2.2.



¿PARA QUÉ SE USA?

Inicialmente, cuando fue presentado por Apple, su propósito era ser una alternativa a Objective-C para el desarrollo de aplicaciones nativas para las plataformas Apple (**iOS, iPadOS, macOS, watchOS y tvOS**).

Con la versión 2.2 y tras ser liberado como código abierto, ha sido portado a Linux y Windows y han surgido varios proyectos open source que hacen uso del mismo. Sobre el papel, podría ser utilizado para abordar cualquier desarrollo.

En la actualidad, también puede ser utilizado para el desarrollo de backend, por ejemplo con **VAPOR**. También fue adoptado por Google para usarlo con **TensorFlow** como alternativa o complemento a Python.

Recientemente se ha anunciado un proyecto para poder utilizar Swift también con AWS lambda.



CARACTERÍSTICAS

- Interoperable con Objective-C (y por extensión, con C y C++ mediante un wrapper). Posteriormente añadida interoperatividad con Python.
- Lenguaje fuertemente tipado.
- Compilado.
- Orientado a protocolos.
- Inferencia de tipos.
- Distinción entre tipos de datos opcionales y no opcionales para prevenir los problemas relacionados con la nulabilidad.
- Cuenta con características de programación funcional.
- Closures (conocidos como lambdas en otros lenguajes) y funciones first-class.
- Multiplataforma con soporte para macOS, Linux y Windows.



Definición

SPM es un **gestor de dependencias** basado en paquetes. Tiene soporte para aplicaciones en lado servidor y aplicaciones de front nativas de Apple. Se puede crear, ejecutar y desplegar por línea de comandos desde el lanzamiento de Swift 3.0, o a través de Xcode desde Swift 5.0.



¿QUÉ ES UN PAQUETE?

Un paquete es una colección de archivos de código en Swift que conforman una librería, también contiene un fichero llamado **Package.swift** que es el manifiesto en el que se declara la configuración del propio paquete. La configuración básica de un paquete contiene:

- El nombre del paquete.
- Las plataformas soportadas.
- Los productos que genera el paquete (las librerías).
- Las dependencias (puede contener dependencias con otros paquetes).
- Los targets que contiene (Módulos).



¿PARA QUÉ SIRVE?

Cuando necesitamos importar librerías propias o de terceros a nuestro proyecto, vamos a enfrentarnos al problema de ver cómo gestionar las actualizaciones y dependencias que éstas tienen.

A día de hoy, en el mundo del desarrollo iOS, existen soluciones como Cocoapods o Carthage que dan más problemas que soluciones. Con **SPM** vamos a tener las siguientes ventajas:

- Integración total en Xcode.
- Funciona en Linux.
- Linkado automático.
- Gestiona automáticamente las dependencias de los paquetes.
- De código abierto.



¿CÓMO LO USO?

Puedes crear un paquete por línea de comandos:

\$ swift package init

Con de Xcode:

File -> New -> Swift Package.

Una vez creado se sube a un repositorio público de Git.

Para importar tu paquete desde Xcode:

File -> Swift Packages -> Add Package Dependency

También puedes importar tu paquete como dependencia de otro paquete en el archivo **Package.swift**.



Definición

Es un proyecto de Apple para transformar las aplicaciones desarrolladas para las plataformas iOS/iPadOS en aplicaciones para Mac. Es importante tener en cuenta que solo las apps con versión para iPad van a poder adaptarse.



¿CÓMO?

La transformación se hace **traduciendo componentes** de UIKit (iOS) en AppKit (macOS).

Hay que tener en cuenta que hay frameworks de iOS que no tienen su contrapartida en Mac, por lo que si usas ARKit, HealthKit, la cámara o el giroscopio en tu aplicación, tendrás que tenerlo en cuenta en tu código:

```
#if !targetEnvironment(macCatalyst)
// Code to exclude from Mac.
#endif
```



¿CUÁNDO?

No todas las apps van a tener sentido en macOS, si tu aplicación se basa en la navegación GPS, o entrenar en un gimnasio, Apple no la va a aprobar para su publicación en el Store.

Cuando tu aplicación sea idónea para la transformación en una versión para Mac, vas a tener que implementar una serie de características que igual no tienes implementadas en iOS:

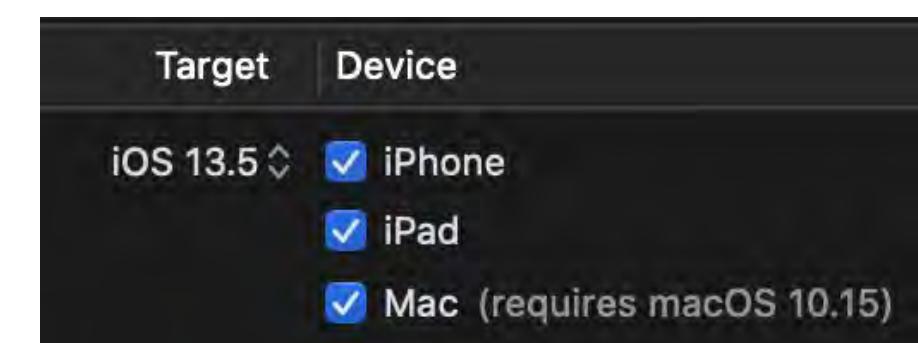
- Soporte para multitarea.
- Drag and drop.
- Atajos de teclado.



¿DÓNDE?

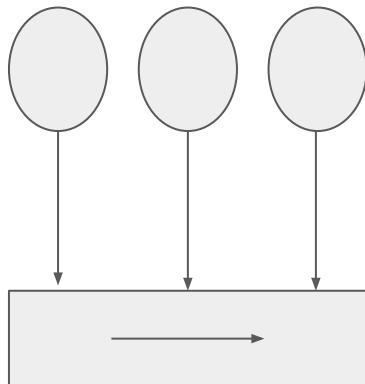
Para hacer tu app compatible con Mac basta con tener instalado macOS 10.15 y Xcode 11.

En la pestaña **General** de tu proyecto y en el apartado **Deployment Info** activar el checkbox de Mac:



Una vez activado podrás correr la app en el target de macOS desde tu Scheme.

Combine



Definición

Es un framework de Apple que nos proporciona una API declarativa para el proceso de valores a lo largo del tiempo (programación reactiva).



CARACTERÍSTICAS PRINCIPALES

Los conceptos básicos que hemos de entender en **Combine** son dos:

- **Publisher:** el publicador es un tipo de objeto **observable** que emite valores a lo largo del tiempo. Un publicador tiene operadores para actuar sobre valores recibidos de otros publicadores y volver a publicarlos.
- **Subscriber:** el suscriptor es un tipo de objeto que **escucha los valores** generados por los publicadores. Los publicadores solo emiten valores cuando el suscriptor los solicita.

Se puede combinar la salida de diferentes publicadores y coordinar su interacción.

```

let publisher = PassthroughSubject<Int, Never>()

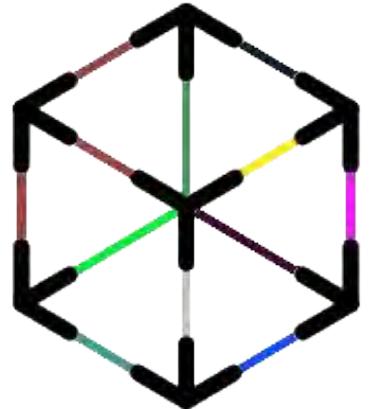
let subscription = publisher
    .sink { value in
        print("Subscription received integer: \(value)")
    }
  
```



COMBINE + FOUNDATION

Con el lanzamiento de Combine, Apple ha añadido muchas extensiones en su framework Foundation para añadir las capacidades y **ventajas de Combine** en muchas de sus clases. Por ejemplo:

- URLSession.
- Timer.
- NotificationCenter.
- Sequence.



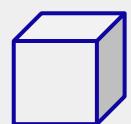
Definición

Es un framework de Apple para crear experiencias de **realidad aumentada**. Trabajando junto con **RealityKit** y la herramienta **Reality Composer**, dispondremos de todo lo necesario para desarrollar nuestras apps de AR.



LO QUE HAY QUE SABER

- **ARKit** es un framework para obtener información del mundo real. A partir de su versión 3, junto con RealityKit soporta oclusión automática en tiempo real, seguimiento facial de hasta 3 personas simultáneas y captura de movimiento en tiempo real.
- **RealityKit** es un framework que ofrece una API para renderizado en 3D, utiliza la información proporcionada por ARKit para integrar fácilmente objetos virtuales en entornos del mundo real, con escalado automático, renderizado fotorealista, física y animaciones.
- **Reality Composer** es una herramienta de Xcode para crear y editar escenas de AR sin código, permite importar archivos USDZ y animarlos, así como desencadenar eventos con distintos tipos de inputs.



LOS COMPAÑEROS PRINCIPALES

ARKit tiene 4 compañeros para el renderizado de gráficos:

- RealityKit.
- SceneKit.
- SpriteKit.
- Metal.



VENTAJAS

- Está **completamente integrado en el ecosistema de Apple** y se puede usar con otros frameworks de Apple como SceneKit o SpriteKit, así como en las apps Mac Catalyst.
- Es **bastante sencillo de usar** ya que tiene muchas cosas hechas e integradas por Apple como el rastreo de la cara, efectos especiales, etc.



Definición

Es un **framework de Apple para dibujar formas, partículas, texto, imágenes y vídeo en dos dimensiones**.

Aprovecha Metal para lograr alto rendimiento, al tiempo que ofrece una interfaz de programación simple para facilitar la creación de juegos y otras aplicaciones intensivas en gráficos. SpriteKit es compatible con iOS, macOS, tvOS y watchOS, y se integra bien con los frameworks como GameplayKit y SceneKit.

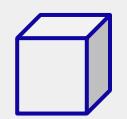


BLOQUES PRINCIPALES

SKView: una vista en la que se presentan *SKScenes*.

SKScene: una escena 2D que se presenta en un *SKView* y contiene uno o más *SKSpriteNodes*.

SKSpriteNode: una imagen 2D individual que puede ser animada alrededor de la escena.



BLOQUES DE CONSTRUCCIÓN

SKNode: un nodo más general que se puede usar en una escena para agrupar otros nodos para un comportamiento más complejo.

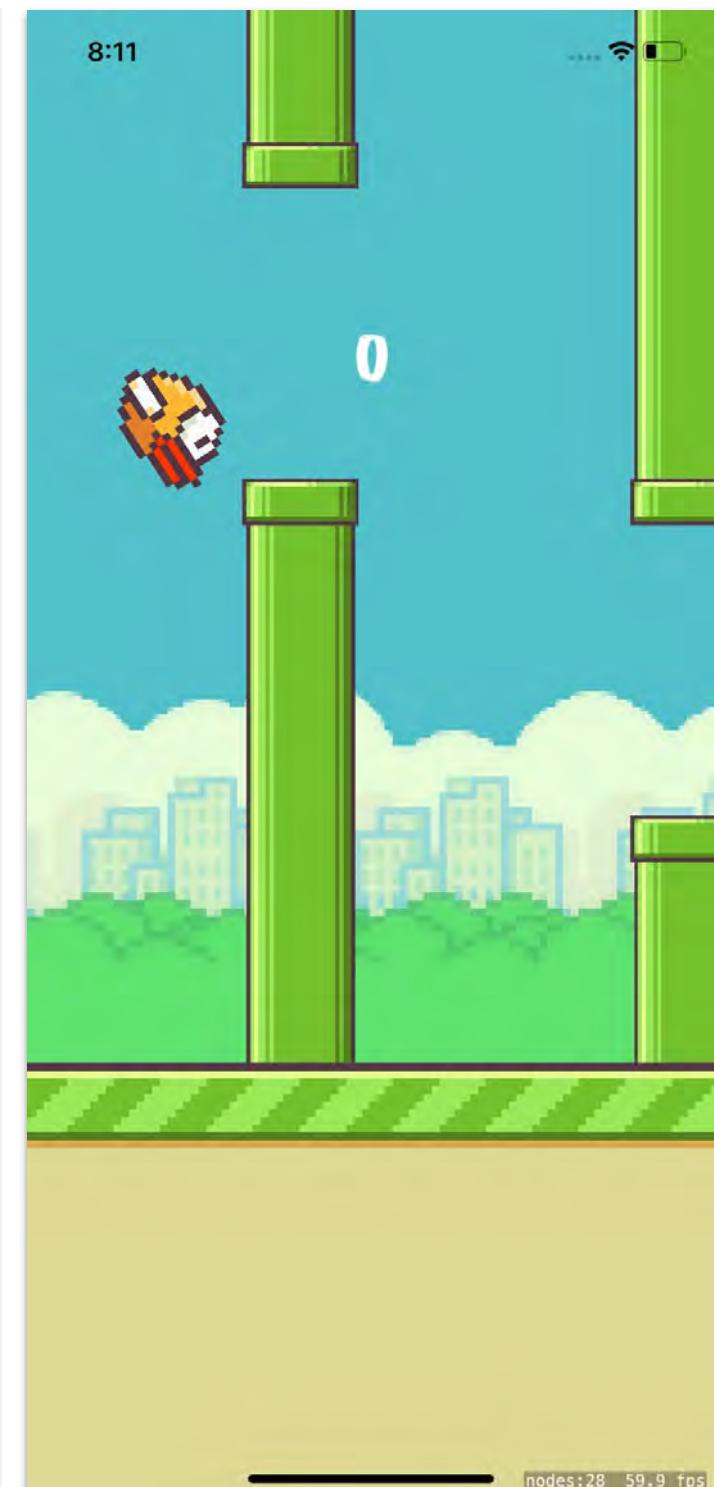
SKAction: acciones individuales o grupos de acciones que se aplican a *SKNodes* para implementar animaciones y otros efectos.

SKPhysicsBody: permite que la física se aplique a *SKNodes* para permitir que se comporten de una manera realista, incluida la caída por gravedad, rebotar entre sí y seguir trayectorias balísticas.



VENTAJAS

- Está integrado en el ecosistema de Apple.
- Es bastante fácil de utilizar.
- Es rápido gracias a Metal.
- Es gratuito.
- Puedes utilizarlo en todas las plataformas de Apple.
- Tiene el motor de física integrado.





SceneKit

Definición

Es un **framework de Apple para construir las aplicaciones de gráficos 3D de alto nivel** que ayuda a crear escenas y efectos animados en 3D. Incorpora un motor de física, un generador de partículas y formas fáciles de escribir las acciones de los objetos 3D para que puedan describir su escena en términos de su contenido (geometría, materiales, luces y cámaras) y luego animarla describiendo los cambios en esos objetos.

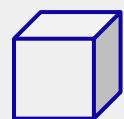


CARACTERÍSTICAS PRINCIPALES

El framework de SceneKit fue lanzado por primera vez por Apple junto con OS X 10.8 Mountain Lion y más tarde estuvo disponible en iOS con el lanzamiento de iOS 8.

El **propósito de éste framework es permitir a desarrolladores, integrar fácilmente gráficos 3D** en juegos y aplicaciones sin las complejidades de APIs para gráficos, tales como OpenGL y Metal. Todos los recursos, representados por nodos, son arreglados en un árbol jerárquico llamado **scene graph**.

Éste árbol funciona muy similar a un **view hierarchy** regular en UIKit. SceneKit te permite simplemente proporcionar una descripción de los recursos que deseas en su escena, con el propio framework que maneja todo el código de representación OpenGL para ti.



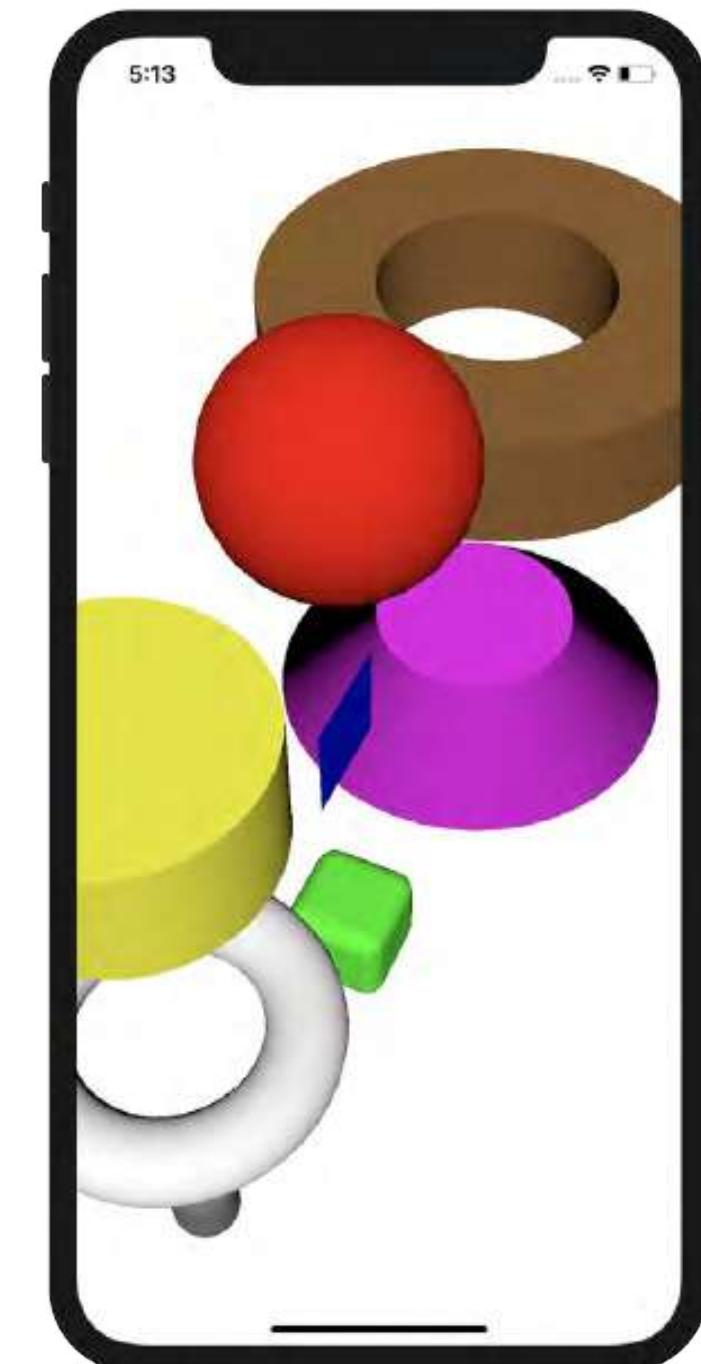
BLOQUES PRINCIPALES

SCNView: una vista en la que se presentan SCNScenes.

SCNScene: una escena 3D que se presenta en un SCNView

SCNVector: un vector de tres componentes que representa la posición de un nodo.

SCNNode: un elemento estructural de un gráfico de escena que representa una posición y se transforma en un espacio de coordenadas 3D, al que puede adjuntar geometría, luces, cámaras u otro contenido visualizable.





Definición

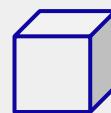
Es un framework de Apple que gestiona las peticiones de los usuarios para los servicios de las aplicaciones que se originan en Siri o Maps.



CARACTERÍSTICAS PRINCIPALES

SiriKit abarca los frameworks de *Intents* e *Intents UI*, que utiliza para implementar extensiones de aplicaciones que integran sus servicios con *Siri* y *Maps*. *SiriKit* admite dos tipos de extensiones de aplicación:

- **Una extensión de aplicación *Intents*** recibe solicitudes de los usuarios de *SiriKit* y las convierte en acciones específicas de la aplicación. Por ejemplo, el usuario puede pedirle a Siri que envíe un mensaje, reserve un viaje o comience un entrenamiento con su aplicación.
- **Una extensión de la aplicación *Intents UI*** muestra la marca u otro contenido personalizado en la interfaz Siri o Maps, después de que la extensión de la aplicación *Intents* cumple una solicitud del usuario. La creación de esta extensión es opcional.



BLOQUES PRINCIPALES

SiriKit define los tipos de solicitudes, conocidos como **Intents**, que los usuarios pueden realizar.

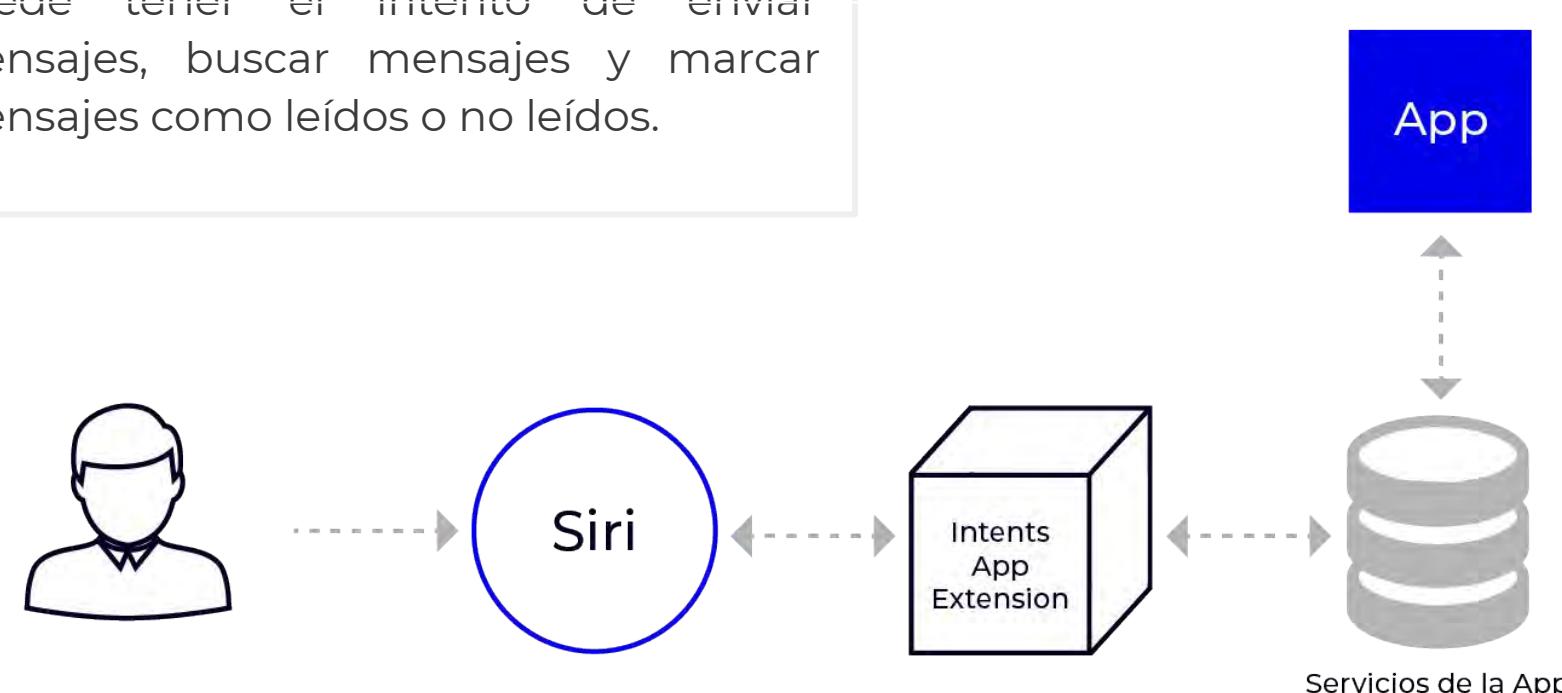
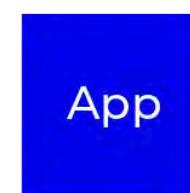
Los **dominios** (domains group) agrupan las intenciones relacionadas para dejar claro qué intenciones puede soportar la aplicación.

Por ejemplo, el dominio de mensajes puede tener el intento de enviar mensajes, buscar mensajes y marcar mensajes como leídos o no leídos.



RESTRICCIONES

- Hay que pedir el permiso al usuario.
- El sistema provee unos dominios predefinidos (envío de mensajes, establecer llamadas de voz sobre IP, pagos, etc.), en caso de que no se ajuste a lo deseado es necesario implementar un Intent personalizado.





Definición

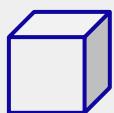
Es un **framework de Apple para persistir o almacenar en caché los datos en un solo dispositivo**. Abstira los detalles de mapear los objetos a un store (almacén), lo que facilita guardar datos de Swift y Objective-C sin administrar una base de datos directamente.



CARACTERÍSTICAS PRINCIPALES

Usa Core Data para guardar los datos permanentes de la aplicación para uso fuera de línea, almacenar datos temporales en caché y agregar funcionalidad de deshacer a su aplicación en un solo dispositivo.

- **Persistencia.**
- **Abstracción.**
- **Deshacer y rehacer** cambios individuales o por lotes.
- **Tareas de datos** en segundo plano (background tasks).
- **Versionado y Migración.**
- **Sincronización** de los datos con las vistas .



BLOQUES PRINCIPALES

- **Persistent store coordinator:** es un wrapper de nuestra base de datos, administra las conexiones (lecturas y escritura), lo podemos visualizar como un apuntador a nuestra base de datos.
- **NSManagedObjectModel:** describe el schema de nuestra base de datos, es decir, las tablas y relaciones.
- **NSManagedObjectContext:** nos permite crear, solicitar o actualizar objetos de nuestra base de datos.
- **Persistent container:** encapsula todos los componentes anteriores.



VENTAJAS

- Está integrado en el ecosistema de Apple
- El agrupar, filtrar u organizar nuestros datos.
- Evitar escribir SQL y lograr queries complejos mediante el uso de objetos Fetch Requests y Predicates.
- Reducir el impacto a nivel memoria (memory footprint) mediante el uso de Faulting.



Definición

Es un **framework de Apple para para incrustar los mapas de Apple directamente en tus propias vistas**. Proporciona la interfaz para añadir tus propias capas, anotaciones o texto encima del mapa.

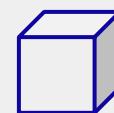


CARACTERÍSTICAS PRINCIPALES

Usa el framework **MapKit para incrustar mapas directamente en tus vistas**. Puedes agregar anotaciones y superposiciones al mapa, los puntos de interés, etc. También proporciona el mecanismo para buscar los puntos de interés.

Si tu aplicación ofrece indicaciones de tránsito, **puedes hacer que tus indicaciones estén disponibles en la app Mapas**. Además, puedes usar la app Mapas para complementar las instrucciones que proporcionas en tu aplicación.

Por ejemplo, si tu aplicación solo proporciona indicaciones para viajar en metro, puedes usar Maps para proporcionar indicaciones a pie desde y hacia las estaciones de metro.



BLOQUES PRINCIPALES

- **MKMapView:** una vista de mapa incrustable, similar a la proporcionada por la aplicación Mapas.
- **MKMapItem:** un punto de interés en el mapa.
- **MKAnnotationView:** la representación visual de uno de tus objetos de anotación.
- **MKDrections:** un objeto de utilidad que calcula direcciones e información de tiempo de viaje en función de la información de ruta que proporcionas.



VENTAJAS

- Está integrado en el ecosistema de Apple.
- Es bastante fácil de utilizar.



RESTRICCIONES

- La cobertura de los mapas aún es mejorable en muchos sitios.



Definición

Es un **framework de Apple para detectar etiquetas NFC**, leer la información guardada en ellas en el formato NDEF y modificarla.

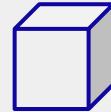


CARACTERÍSTICAS PRINCIPALES

El framework Core NFC sirve para leer etiquetas para dar a los usuarios más información sobre su entorno físico y los objetos del mundo real.

Con Core NFC, **puedes leer las etiquetas de comunicación de campo cercano (NFC) de los tipos 1 a 5 que contienen datos en el formato de intercambio de datos NFC (NDEF)** y conectar con etiquetas ISO7816, ISO15693, FeliCa y MIFARE.

Por ejemplo, la aplicación podría proporcionar a los usuarios información sobre los productos presentados en una tienda o los cuadros en un museo.



BLOQUES PRINCIPALES

- **NFCReaderSession:** una clase de sesión de lectura para detectar etiquetas de formato de intercambio de datos NFC (NDEF) con NFCNDEFReaderSession y conectar con el resto de tags con NFCTagReaderSession.
- **NFCNDEFReaderSessionDelegate:** un protocolo de delegado que sirve para leer los datos NDEF de una etiqueta NFC.
- **NFCTagReaderSessionDelegate:** un protocolo de delegado que sirve para gestionar la conexión a un tag.
- **NFCNDEFPayload:** el payload (carga útil) que contiene los datos, el nombre de tipo de formato de NDEF de un mensaje, etc.
- **NFCTag:** representa el tag NFC al que se ha conectado.



RESTRICCIONES

- Core NFC **no está disponible para su uso en extensiones de aplicaciones** y requiere de un dispositivo que admita la comunicación de NFC.
- Core NFC **no admite ID de aplicaciones relacionadas con el pago**.
- **El uso de Core NFC es restringido** y es apto solo para trabajar con las etiquetas NFC y no se permite la comunicación entre otros dispositivos que soportan NFC.



Definición

Es un framework de Apple que proporciona la infraestructura para crear aplicaciones watchOS.



CARACTERÍSTICAS PRINCIPALES

Con la ayuda de framework **WatchKit** podemos crear simples y sencillas aplicaciones para WatchOS.

Se construyen con los elementos predeterminados y dentro de la extensión delegado (*extension delegate*).

A pesar de su simplicidad tienen acceso a las tareas en segundo plano, intenciones de Siri, sesiones de entrenamiento, etc.

Los usos **más adecuados para apps WatchOS son mediciones de todo tipo**, las apps con solo una función principal (como Shazam), rastreadores de ejercicios, mostrar la información en el formato compacto, etc.



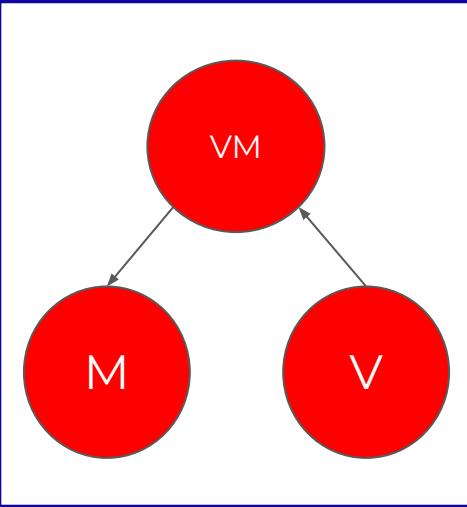
BLOQUES PRINCIPALES

- **WKExtension:** una clase de la extensión que gestiona los comportamientos de los controladores de interfaz de tu aplicación.
- **WKInterfaceController:** una clase que proporciona la infraestructura para administrar la interfaz en una aplicación watchOS (al igual que UIViewController en UIKit app).
- **WKExtensionDelegate:** el protocolo que contiene una colección de métodos que administran el comportamiento a nivel de aplicación de una extensión WatchKit.



RESTRICCIONES

- **Se necesita la app-compañero** (a partir de WatchOS 6 se puede crear las apps independientes).
- **Las animaciones están restringidas.** Puedes usar solo un conjunto de imágenes para crear una especie de GIF animado.
- **Las interfaces se crean con los elementos simples**, poco flexibles (una lista, un botón, un label, etc.).
- **Las notificaciones pueden ser de dos tipos:** Long and Short.
- **Los gestos están predeterminados**, no se soporta multitouch (multitáctil).
- **No puedes crear tus propios componentes personalizados.**



Definición

El patrón de arquitectura MVVM **consiste en separar nuestra aplicación en tres capas**: la lógica de negocio, la interfaz gráfica y la lógica de presentación.



CARACTERÍSTICAS PRINCIPALES

El **principal objetivo de este patrón es sacar el estado de la vista y la lógica de presentación de la vista**. De este modo, la vista solo contiene los elementos visuales.

El ViewModel **representa el estado de la vista y maneja los componentes de la vista y sus estados a través de “binding”** (atadura). El binding se puede hacer como un cierre (closure), a través del mecanismo Key-Value Observing o con la ayuda de las librerías de terceros, así como RxSwift/ReactiveCocoa, etc.

SwiftUI ya tiene los mecanismos integrados de “binding” como los property wrappers `@State` o `@Binding`.



BLOQUES PRINCIPALES

- **Model:** representa los conceptos y las entidades de negocio, también la lógica de negocio.
- **ViewModel:** contiene la lógica de presentación y el estado de la vista. Recupera los datos del negocio desde el Modelo, aplicando la lógica de presentación y guardando el estado de la vista. El estado se puede hacer global (a la Redux) o guardando cada viewModel su propio estado.
- **View:** pinta la interfaz gráfica y pasa las interacciones de usuario a ViewModel. Recupera los datos preparados desde ViewModel a través de binding.



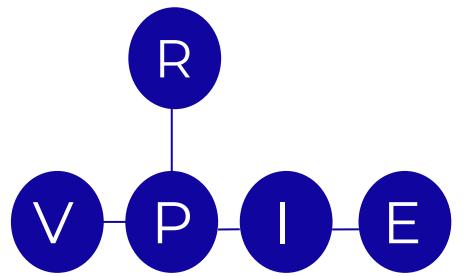
PROBLEMAS

- Con el crecimiento de la lógica de la app el ViewModel puede convertirse en un mega-objeto poco sostenible.
- El problema de la gestión del estado de la vista es bastante complejo.



VENTAJAS

- Es un patrón bastante “sencillo” para implementar sobre todo para las apps simples.
- Alta posibilidad de reutilización de los ViewModels para diferentes Views.



Definición

Viper **es un patrón arquitectónico que implementa los principios de Clean Architecture**, aplicándolos al desarrollo de aplicaciones iOS. Consiste en separar la lógica de la aplicación en cinco capas: View, Interactor, Presenter, Entity y Router.



CARACTERÍSTICAS PRINCIPALES

La implementación del patrón Viper en el desarrollo de aplicaciones iOS **consiste en dividir la lógica de cada módulo (pantalla) de la aplicación en cinco capas principales** con las diferentes responsabilidades.

De este modo, **aislamos las dependencias y podemos testear más fácilmente** tanto las interacciones entre capas como las capas propias.

Las **interacciones entre capas se ejecutan a través de los protocolos** para poder sustituir los objetos reales con los mocks en los tests unitarios.



BLOQUES PRINCIPALES

- **View:** pinta lo que le manda el Presenter y pasa las entradas de usuario al Presenter. Es la capa más sencilla de todas.
- **Interactor:** contiene la lógica de negocio.
- **Presenter:** contiene la lógica de presentación y procesa las entradas de usuario.
- **Entity:** contiene los modelos básicos (entities) de negocio que se usan por Interactor.
- **Router:** contiene la lógica de navegación.



DESVENTAJAS

- Existe cierta “sobrecarga” en crear cinco clases diferentes con los diferentes protocolos por cada módulo.
- Es un patrón complejo que requiere cierto aprendizaje y costumbre de usar.



VENTAJAS

- Cada capa es reutilizable y testeable.
- Con el crecimiento, la aplicación sigue estando estructurada y testeable a pesar de su tamaño.



Definición

La accesibilidad en las aplicaciones móviles significa hacerlas manejables y cómodas para toda la población, independientemente de sus capacidades técnicas o físicas.



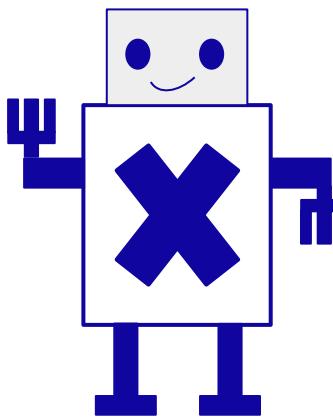
LAS HERRAMIENTAS PARA DESARROLLADORES

- **Accessibility Inspector:** es una herramienta que nos proporciona la información sobre la accesibilidad de cada elemento de nuestra aplicación. Además, puede hacer la auditoría de toda nuestra aplicación en términos de accesibilidad.
- **Accessibility Labels:** cada vista UIView tiene la propiedad accessibilityLabel que facilita la información sobre la vista al lector de pantalla VoiceOver.
- **Accessibility Traits:** para los elementos customizados es importante poner un rasgo(trait) para transmitir el sentido del elemento.
- **Accessibility Notifications:** puedes crear las notificaciones que avisan al usuario sobre los cambios en la interfaz, scroll o cualquier anuncio en la voz alta.
- **DynamicTypes:** escala los textos en la aplicación según los ajustes del móvil.



LAS HERRAMIENTAS PRINCIPALES PARA USUARIO

- **VoiceOver:** el lector de pantalla que pronuncia en voz alta todo lo que sucede en la pantalla y se manipula con los gestos.
- **Zoom:** escala el contenido de la pantalla.
- **Lupa:** utilizando la cámara trasera, escala el espacio alrededor de usuario.
- **Los tamaños del texto dinámicos:** se puede cambiar el tamaño del texto en las apps que lo soportan.
- **Lectura de pantalla:** lee el contenido de pantalla.
- **Acceso guiado:** limita el acceso de usuario a solo una app.
- **AssistiveTouch:** ayuda a los usuarios que tienen dificultades para tocar la pantalla. Muestra un menú adicional en blanco y negro con los botones que permiten manipular el contenido de pantalla fácilmente.



Bots de XCode

Definición

Es un sistema de integración continua de Apple para los proyectos de iOS y Mac.



CARACTERÍSTICAS PRINCIPALES

Para usar el sistema se necesita instalar XCode Server. Está integrado en XCode a partir de la versión 9.

Es una **plataforma de integración continua que está integrada en XCode** y que permite ejecutar los test **unitarios**, los test de la interfaz gráfica, archivar los builds y mandar los correos electrónicos de estado.

Se encarga de crear y mantener los certificados y los perfiles de tu aplicación. También, muestra la cobertura de los test unitarios, ejecuta los test en diferentes dispositivos.

Además, **permite instalar los build OTA** (Over-the-Air) a través de navegador en tu dispositivo.



BLOQUES PRINCIPALES

- **Bot:** es un conjunto de tareas que quieres hacer con el código de repositorio.
- **XCode Server:** es una herramienta que permite ejecutar los Bots tanto localmente como en otro ordenador.
- **MacOS Server:** es una aplicación Mac que complementa MacOS con las funcionalidades de servidor de diferentes tipos como MTA servidor, AFP y SMB servidor, el servidor web, servidor wiki, servidor de mensajería, etc.
Está disponible en App Store como la aplicación separada. Originalmente era necesario instalarlo pero actualmente ya no es necesario para poder utilizar los bots.



DESVENTAJAS

- Se puede usar solo para proyectos iOS/Mac.
- La funcionalidad está bastante limitada comparando con otras soluciones (Jenkins, Bamboo, etc).
- Tiene pequeños bugs.



VENTAJAS

- Se puede usar localmente en tu ordenador.
- Está integrado en XCode y no hace falta instalar nada más.
- Se ejecutan los test UI en diferentes dispositivos iOS/Mac.



¿Qué es?

Con App Clips **podemos crear un versión «reducida» de nuestra app que se ejecutará con un disparador como un código QR o una etiqueta NFC.** Están pensados para crear una experiencia limitada y contenida de lo que la app principal puede ofrecer, como vender algún producto o mostrar algún tipo de información.



VENTAJAS DE NEGOCIO

Los Clips son una **buenas herramienta para aumentar la conversión de los potenciales usuarios** al no tener que buscar nuestra app en el App Store.

Los Clips pueden mostrarse con los siguientes disparadores:

- Escanear una etiqueta NFC o código visual en una ubicación física.
- Al tocar una sugerencia basada en la ubicación de Siri Suggestions.
- Tocando un enlace en la aplicación Mapas.
- Al tocar un banner de aplicación inteligente en un sitio web.
- Tocando un enlace que alguien ha compartido en la aplicación Mensajes.



LIMITACIONES

Los Clips presentan algunas limitaciones a la hora de desarrollarlos que se deben tener en cuenta:

- Disponibles con iOS 14.
- No pueden pesar más de 10 megas.
- Ha de ser nativo 100%.
- La construcción ha de ser 100% con SwiftUI.
- No puede acceder a los datos de usuario, por lo que los frameworks relacionados con éste no están disponibles (Salud, Contactos...).
- Al cabo de un tiempo de inactividad, el clip es borrado por el sistema.



DESARROLLO

Para añadir un App Clip a tu aplicación principal basta con crear un Target nuevo tal y como lo hacemos con las extensiones existentes, no hay limitación al número de clips que podemos añadir.

Los clips permiten programar una notificación tras 8 horas después de su descarga para recordar algo al usuario, al pulsarla, se vuelve a abrir el clip.

Hay una nueva API para verificar que el usuario se encuentra en el lugar donde debe estar el Clip.

Los permisos que demos al Clip los hereda la app principal en caso de descarga de ésta.



Definición

Con la ayuda de WidgetKit **puedes crear los widgets (mini-vistas) para tres plataformas de Apple: iOS, iPad OS y macOS**. A partir de iOS 14, puedes poner los widgets en el escritorio de tu iPhone, en TodayView en iPad OS y en el centro de notificaciones de macOS Big Sur (11.0).



CARACTERÍSTICAS PRINCIPALES

Los widgets son una manera dinámica y directa para demostrar la información útil de tu aplicación en el escritorio de tu móvil, tablet o portátil. El widget tiene que ser relevante, personalizable y entendible de un solo vistazo. La extensión de WidgetKit es una extensión de fondo (background), que proporciona las vistas del widget de antemano para cada momento específico. Así, se evita el problema de “carga” y “espera”. Los widgets se configuran a través de Intents. Además, se pueden actualizar las jerarquías de vistas ya preparadas desde tu aplicación o establecer un horario de actualización. Se puede crear una pila de widgets que va cambiando el widget que se muestra arriba de todos.



LIMITACIONES

- Está disponible solo a partir de iOS 14 y macOS Big Sur(11.0).
- Los widgets no pueden contener los elementos interactivos (como los campos de texto o los botones).
- Tienen solo tres tamaños (pequeño, medio y grande).
- Tienen que ser escritos con SwiftUI.
- Los widgets no tienen el estado.
- No soportan el scrolling.
- No soportan la reproducción de los videos o las imágenes animadas.
- La única interacción que está permitida es a través de tocar el widget que está vinculado a un enlace profundo (deepLink) de tu aplicación o creando subenlace dentro del widget con Link API.

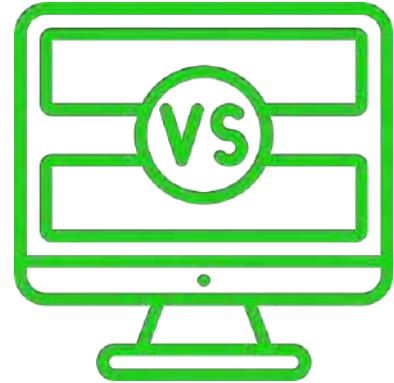


DESARROLLO

WidgetKit soporta SwiftUI, AppKit y las apps de macOS basadas en Catalyst. El widget se define a través de:

- **kind:** el identificador de tipo String de widget.
- **configuration:** el widget puede ser estático (*Static Configuration*) o configurable a través de Intent (*IntentConfiguration*).
- **supportFamilies:** el tamaño de widget: *systemSmall*, *systemMedium* y *systemLarge*.
- **placeholder:** la vista que muestra la estructura de widget sin mostrar los datos de usuario.
- **provider:** la estructura que proporciona el conjunto de vistas para cada momento de tiempo y la primera entrada de datos (*snapshot*).

Back: Introducción al Backend y Java



¿Qué les diferencia?

Los lenguajes de alto nivel utilizan una sintaxis más parecida a la natural, a cómo nos comunicamos los humanos, mientras que los de bajo nivel utilizan unas instrucciones y sentencias más parecidas al lenguaje que hablan las máquinas.



COMPARATIVA

Ventajas de los lenguajes de alto nivel:

- El **código** es **muchísimo más mantenible** ya que si está bien escrito, es posible ,a veces a simple vista, detectar errores.
- Al ser mucho más utilizados, existen muchas más herramientas de apoyo al desarrollo y cuentan con una comunidad mucho más amplia. Esto repercute en **mayores facilidades y herramientas** de desarrollo (IDEs, sistemas de debug...)
- **Desarrollar** programas complejos **es muchísimo más rápido** ya que una sentencia de alto nivel puede englobar decenas de bajo nivel.

Ventajas de los lenguajes de bajo nivel:

- Normalmente **son más rápidos** que los lenguajes de alto nivel, y se suelen utilizar cuando el rendimiento extremo es una necesidad primordial.
- Suelen ocupar menos espacio y tienen **menos requisitos para su ejecución**, por lo que son ideales para dispositivos con poco espacio o poca memoria.



EJEMPLOS DE BAJO NIVEL

- Lenguaje de máquina.
- Lenguaje ensamblador.



EJEMPLOS DE ALTO NIVEL

- Java.
- Python.
- Swift.
- Kotlin.
- Javascript.
- C#.
- Go.
- y muchos más menos conocidos.

01100
10110
11110

¿En qué consiste?

Los lenguajes de programación **compilados** son aquellos que requieren de un paso previo, en el que un compilador traduce todas las instrucciones del programa a código máquina. Un lenguaje **interpretado** realiza dicha traducción en tiempo de ejecución instrucción a instrucción.



CONSIDERACIONES

A día de hoy es raro encontrar lenguajes que se puedan considerar puramente interpretados. Lenguajes como Java, JavaScript, Python, etc., que históricamente fueron considerados como interpretados, hoy en día en realidad se podría decir que son un híbrido, ya que la mayoría realizan un paso previo en el cual traducen el código a **bytecode**, el cual es agnóstico a la plataforma en la cual se ejecuta el código y se suelen apoyar en máquinas virtuales que interpretan dicho **bytecode** y lo traducen a código máquina (por ejemplo la JVM para Java o el motor V8 para Javascript). De esta forma consiguen mayor eficiencia.

Por otra parte, en el caso de lenguajes de scripting, como bash y otros tipos de *shell* es más fácil considerarlos lenguajes interpretados.



EJEMPLOS

Lenguajes compilados:

- Swift.
- C.
- C++.
- C#.
- Objective-C.
- Kotlin/Native.

Lenguajes interpretados:

- Bash, sh, zsh...
- Java.
- JavaScript.
- Python.
- Perl.
- Kotlin (JVM).

Compilado

Interpretado





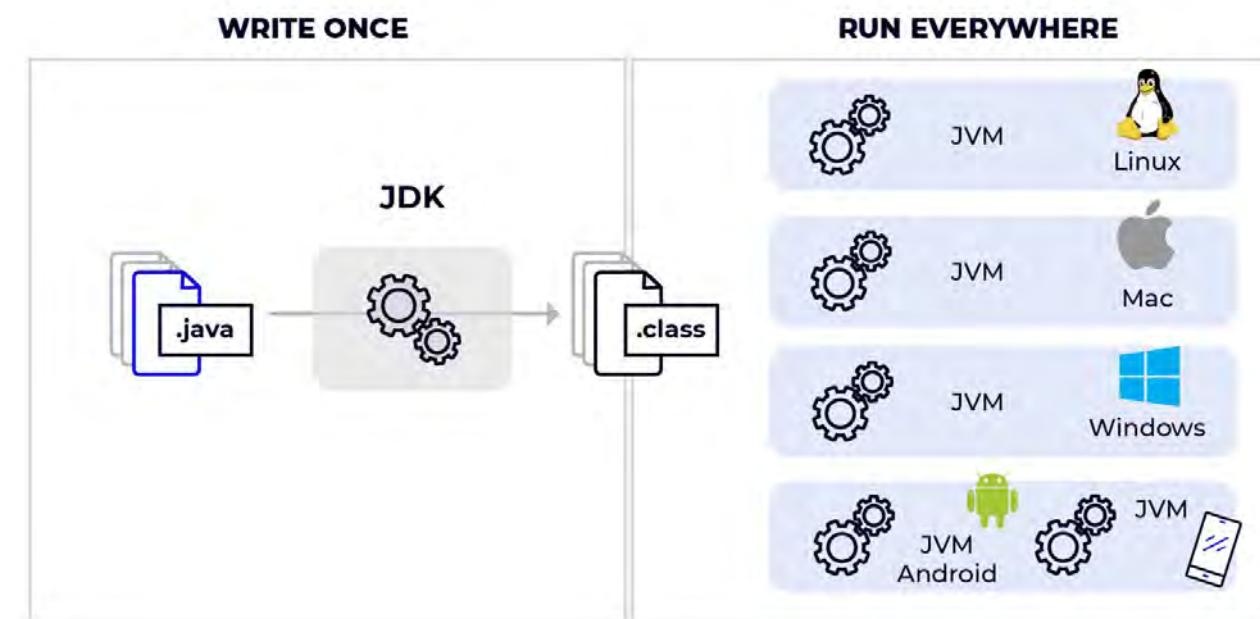
¿Qué es?

Máquina virtual que **permite ejecutar código desarrollado en Java en cualquier sistema operativo**. JVM interpreta y compila a código nativo (de la plataforma concreta de ejecución) las instrucciones expresadas en un código binario especial (**bytecode**), el cual es generado por el compilador del JDK (Java Development Kit).



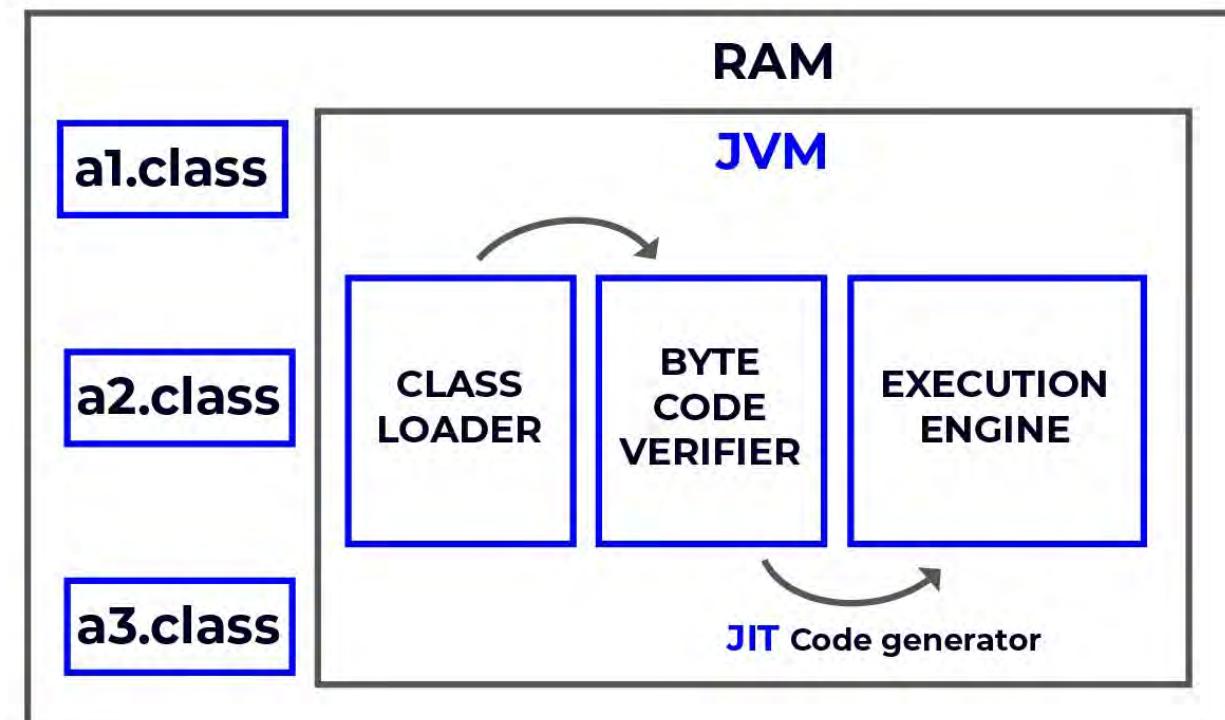
OBJETIVO

La idea cuando se desarrolló Java era **poder ejecutar el código en cualquier plataforma (Write Once Run Anywhere)**. Esto significa que un desarrollador puede escribir código Java en un sistema específico y sabe que se va a poder ejecutar en cualquier otro sin ninguna configuración adicional. La JVM es la que permite diversificar su uso y cada plataforma tiene su propia JVM, que se adapta a cada arquitectura/SO.



¿CÓMO FUNCIONA?

1. Un fichero `example.java` se compila (gracias al compilador del JDK) y se genera un fichero `example.class` que contiene el bytecode capaz de ser interpretado por la JVM.
2. Durante la ejecución del código, **Class Loader** se encarga de llevar los ficheros `.class` a la JVM que reside en la RAM del ordenador y **ByteCode Verifier** se encarga de verificar que el código en bytecode procede de una compilación válida.
3. El **compilador Just in Time** (esto se hace en tiempo de ejecución) compila el bytecode a código nativo de la máquina y se ejecuta directamente.





¿Qué es?

La Programación Orientada a Objetos (POO) es un paradigma de programación, una manera de programar específica que vino a revolucionar la visión que hasta entonces se tenía en los años 80. La POO supuso un cambio a la hora de programar más cercano a un lenguaje natural que los lenguajes existentes hasta el momento. En la POO se agrupa el código en objetos individuales que poseen información y funciones.



ENTENDIENDO LOS OBJETOS

Para entender qué son los **objetos** en la POO vamos a pensar en un teléfono móvil. Cualquier teléfono móvil tiene una serie de características o **propiedades** como puede ser el color, la marca, el modelo, su memoria interna, etc.

Pero además, estos tienen una serie de funcionalidades como son: enviar mensajes, hacer llamadas, tomar fotos, instalar aplicaciones. A estas funcionalidades las llamamos **métodos**.

Por último, para crear un teléfono móvil se utiliza una plantilla previamente diseñada con las características que lo definen. A esta plantilla vacía que nos va a permitir crear cualquier número de objetos "teléfono móvil", en la POO, la llamamos **clase**.

Pero en la POO, no sólo podemos definir como clases objetos físicos, sino conceptos más abstractos como por ejemplo conceptos matemáticos como una integral o una fracción. Para generar un objeto en Java:

Clase Teléfono

Atributos:
Color
Marca
Modelo
Memoria Interna

Métodos:
Enviar mensaje()
Tomar foto()
Hacer llamada()

Teléfono Nexus

Color: Azul
Marca: Google
Modelo: Nexus 4
Memoria interna: 32GB

Teléfono Iphone

Color: Negro
Marca: Apple
Modelo: Iphone 7
Memoria interna: 32GB



CARACTERÍSTICAS

Las principales características más relevantes de la POO son:

- **Abstracción:** Representación de las características esenciales de algo sin incluir antecedentes o detalles irrelevantes.
- **Encapsulamiento:** Consiste en agrupar los elementos que corresponden a una misma entidad en el mismo nivel de abstracción.
- **Principio de ocultación:** Capacidad de ocultar los detalles dentro de un objeto. Protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas.
- **Herencia:** Mecanismo para compartir automáticamente métodos y atributos entre clases y subclases. De esta forma se relacionan las clases entre sí y generan jerarquías de organización.
- **Polimorfismo:** Característica que permite implementar múltiples formas de un mismo método, dependiendo cada una de ellas de la clase sobre la que se realice la implementación.
- **Modularidad:** Propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.
- **Recolección de basura:** en inglés garbage collection, es la técnica por la cual el entorno de objetos se encarga de destruir automáticamente, y por tanto desvincular la memoria asociada, los objetos que hayan quedado sin ninguna referencia a ellos.



¿En qué consiste?

Paradigma de programación **declarativa** en la que las **funciones** son ciudadanas de primera clase.



CARACTERÍSTICAS

- **Funciones de primera clase y de orden superior**
 - Las funciones pueden recibir y devolver otras funciones.
 - Una función puede asignarse a una variable.
- **Funciones puras**
 - No tienen efectos secundarios.
 - Dado un parámetro de entrada devuelven siempre el mismo resultado.
- **Programación declarativa**
 - Expresamos qué queremos hacer y no el cómo.
- **No hay estructuras de control**
 - Se utiliza la recursividad para resolver problemas en los que se utilizarían estas estructuras tradicionalmente.
- **Inmutabilidad**
 - Los lenguajes puramente funcionales simulan el estado pasando datos inmutables entre las funciones.



LENGUAJES

Algunos lenguajes adoptan este paradigma por completo y todas las funciones son **puras**, lo que significa que no hay estado mutable como tal, ni se permiten efectos secundarios.

Por otra parte, algunos lenguajes, llamados **impuros** adoptan parcialmente la programación funcional, permitiendo el uso de características propias funcionales junto con las de otros paradigmas. En los impuros, podremos escribir parte del código en un estilo funcional.

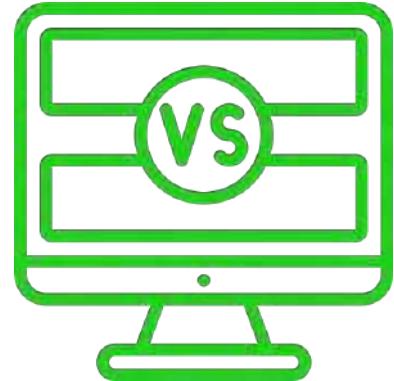
Algunos ejemplos de lenguajes funcionales son:

Puros:

- Haskell.
- Miranda.

Impuros:

- Scala.
- Python.
- Java (a partir del 8).
- Kotlin.
- Rust.



¿Qué es?

La programación reactiva es un paradigma basado en el desarrollo declarativo por contra al tradicional, que es imperativo. Se trata de funcionar de una forma **no bloqueante, asíncrona y dirigida por eventos**.



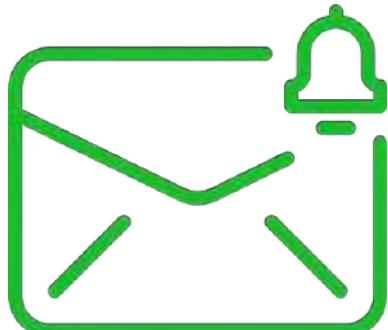
¿QUÉ PROBLEMA RESUELVE?

Se trata de evitar ciertos problemas que se han visto que pueden suceder con las arquitecturas tradicionales, que suelen funcionar de una forma bloqueante. Esto puede en ocasiones desaprovechar la CPU, ya que los hilos se encuentran bloqueados por entrada salida (ir a base de datos, consultar el API de un tercero...), incluso puede llegar a la saturación del sistema y finalmente su caída con volúmenes de carga altos. La programación reactiva se sustenta en la base de **NIO**. Con NIO en vez de un pool de hilos en la que cada hilo procesa una petición de inicio a fin, vamos a tener hilos workers. Estos workers van a coger las peticiones de los usuarios y en vez de esperar cuando se bloqueen, van a utilizarse para servir otras peticiones, aprovechando al máximo nuestra CPU. Es decir, vamos a tener un escalado vertical óptimo. Tanto en on premise como en cloud esto puede **suponer un ahorro en costes de infraestructura**. Lo recomendable es que el número de workers sea igual que el número de cores que tenga nuestra CPU.

Con las nuevas arquitecturas, como por ejemplo los microservicios, nos podemos ver en la necesidad de hacer múltiples llamadas entre ellos, lo que puede desembocar en una maraña de mensajes que tienen que ir en cierto orden para al final producir una respuesta. Si a eso le sumamos el protocolo de comunicaciones HTTP (que es síncrono) podemos tener problemas de rendimiento.

En el 2014 surge el [manifiesto de los sistemas reactivos](#). Podemos ver conceptos como que los sistemas tienen que ser **responsivos** (responder rápido en la medida de lo posible), **resilientes** (tolerante a fallos), **elásticos** (adaptable a carga) **y orientados a mensajes** (de forma asíncrona). La programación reactiva nos ayuda solo en el último de estos pasos y en algunos casos en el primero. Uno de los conceptos claves será establecer un **mecanismo de backpressure**, que nos va a permitir limitar los mensajes por parte de los productores para que los consumidores no se saturen.

Se está viendo cada vez una adopción más de este paradigma y solo hace falta mirar lenguajes como Java, que desde la versión 9 incorpora [Reactive Streams](#) o frameworks como Spring que también lo incluyen en su módulo [webflux](#). La programación reactiva no debe usarse para todo. Se recomienda para escenarios con tiempos de respuesta superiores al segundo cuyo tiempo se pierde casi en la totalidad en peticiones bloqueantes.



Reaccionando a cambios de estado

El patrón observador (*observer* en inglés) describe una solución que se asemeja al manejo de eventos. Principalmente es utilizado para permitir que ciertos objetos puedan reaccionar a los cambios que suceden en un momento dado en otros objetos.



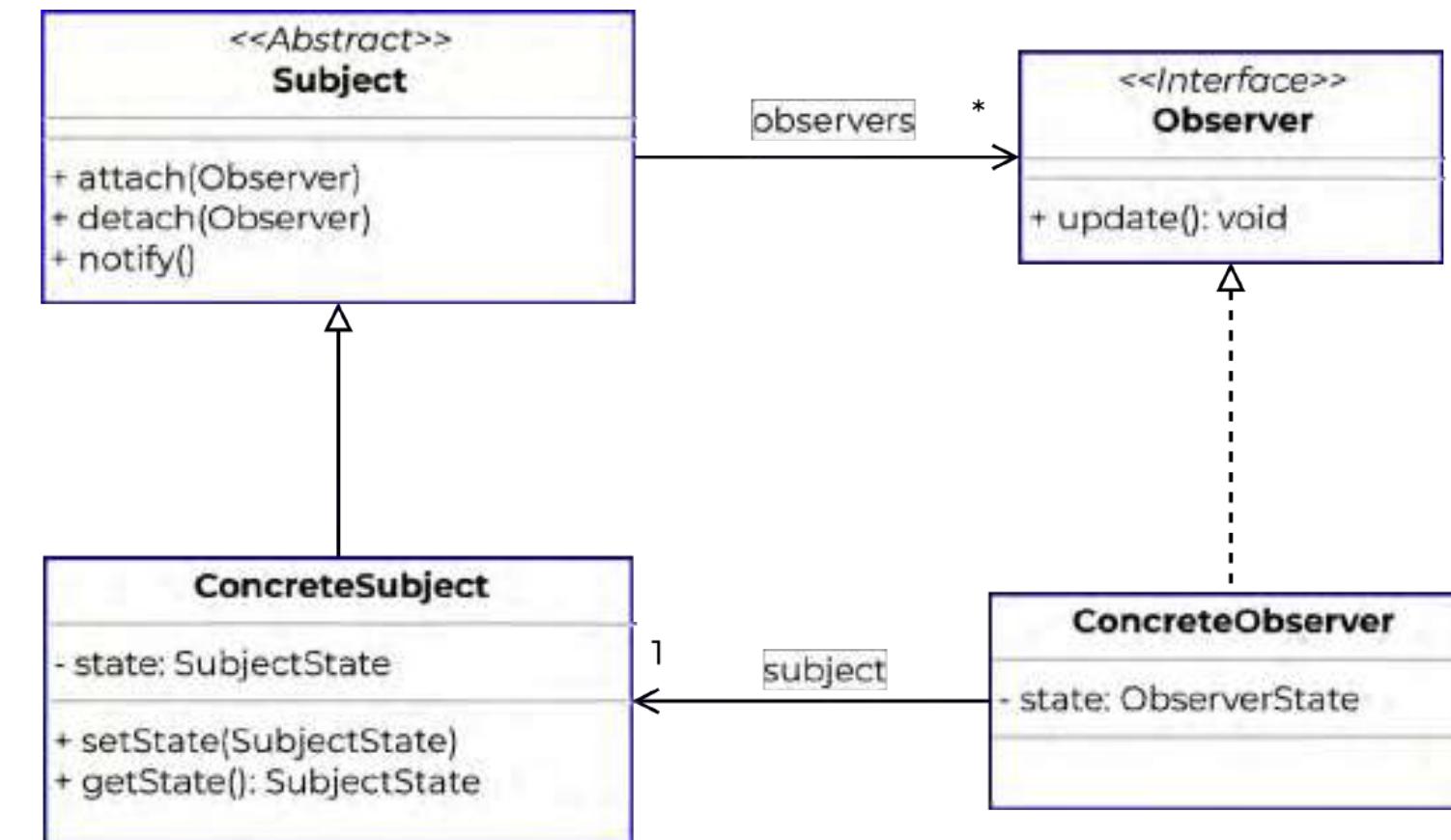
CONCEPTO

Dado el diagrama, tenemos una clase abstracta, **Subject**, que implementa una serie de métodos para enlazar observadores a la clase. Cuando el estado del subject cambie, notificará a sus observadores, invocando el método *update* en cada una de ellas.

La interfaz **Observer** expone un solo método, cuya invocación dependerá del Subject. La lógica de este método es una reacción al cambio sucedido en Subject.

Las clases **ConcreteSubject** y **ConcreteObserver** denotan una manera de implementar el patrón. Se observa que el **ConcreteObserver** tiene una referencia al **ConcreteSubject**, lo cual le permite obtener su estado. De esta manera, cuando el Subject notifique a sus Observers, esta implementación en particular, tendrá una referencia directa al Subject para poder reaccionar en base a los cambios sucedidos.

Esta no es la única forma de diseñar el patrón Observer, pero sí es la clásica descrita por el GoF (Gang of Four).





El sistema de publicaciones

Oracle con la publicación de Java SE 9 en Septiembre de 2017 no sólo introdujo como mejora la Modularidad en Java, también decidió apostar por las versiones de Java menores que no disponen de soporte extendido (LTS) y que desde entonces se publican cada 6 meses, siendo la última Java SE 14. La próxima versión LTS será Java SE 17 en Septiembre de 2021. A continuación, destacamos las novedades de las últimas versiones publicadas:



NOVEDADES JAVA 12

Las novedades más destacadas son:

El recolector de basura Shenandoah: con la pretensión de mejorar el rendimiento de las aplicaciones.

230: Microbenchmark Suite

Las expresiones Switch: Se trata de una mejora que nos permite eliminar varias sentencias *if else* encadenadas en la expresión Switch.

Colectores Teeing: para enviar un elemento de un Stream a dos Streams.

Formato de número compacto: Ahora se puede expresar un número en formato compacto.

Suite de Microbenchmark: añade una suite básica de Microbenchmark al código fuente del JDK que facilita a los desarrolladores ejecutar Microbenchmark existentes o crear nuevos.

Fuente: <https://docs.oracle.com/en/java/javase/12/>



NOVEDADES JAVA 13

Las novedades más destacadas son:

Bloques de texto: utilizando la triple comilla doble ("") se permite identificar grandes bloques de texto que facilitan la visualización del código.

Expresiones Switch mejoradas: ahora las expresiones switch pueden también devolver un valor en lugar de sentencias.

Archivos CDS Dinámicos: Extiende el uso de la aplicación CDS para permitir el archivado dinámico de clases al final de la ejecución de la aplicación Java.

Uncommit de la memoria no utilizada: Mejora ZGC para devolver la memoria de almacenamiento dinámico no utilizada al sistema operativo.

Reimplementación de la API Socket Legacy: Permite reemplazar la implementación subyacente utilizada por las API con una implementación más simple y moderna que sea fácil de mantener y depurar.

Fuente: <https://docs.oracle.com/en/java/javase/13/>



NOVEDADES JAVA 14

Las novedades más destacadas son:

Records. Sin duda alguna, la novedad más importante de esta versión de Java. Los registros son clases que no contienen más datos que los públicos declarados y permite reducir considerablemente el código en algunas clases.

Excepciones NullPointerException más útiles

Pattern Matching para el operador instanceof

Bloques de texto: se definen nuevos caracteres de escape.

Recolector de basura ZGC para Windows y MacOS

Expresiones switch en modo vista

Herramienta de Packaging

Actualización de MappedByteBuffer para soportar el acceso a la memoria no volátil (NVM).

Fuente: <https://docs.oracle.com/en/java/javase/14/>



¿Qué es?

Es una **JVM** y **JDK** basada en HotSpot/OpenJDK e implementada en Java. Soporta lenguajes de programación adicionales y modos de ejecución, como la compilación ***ahead-of-time*** que permite un **tiempo de arranque más rápido** en aplicaciones Java, resultando en ejecutables que **ocupan menos memoria**.



OBJETIVOS

- Mejorar el **rendimiento** de los lenguajes basados en la máquina virtual de Java haciendo que tengan un rendimiento similar a los lenguajes nativos.
- **Reducir el tiempo de arranque** de las aplicaciones de la JVM mediante la compilación *ahead-of-time* (antes de tiempo) con GraalVM Native image.
- Permitir la integración de GraalVM en Oracle Database, OpenJDK, Node.js, Android/iOS y otros similares.
- Permitir utilizar código de cualquier lenguaje **en una única aplicación**.
- Incluir una colección de herramientas, fácilmente extensible, para la programación de **aplicaciones políglotas**.



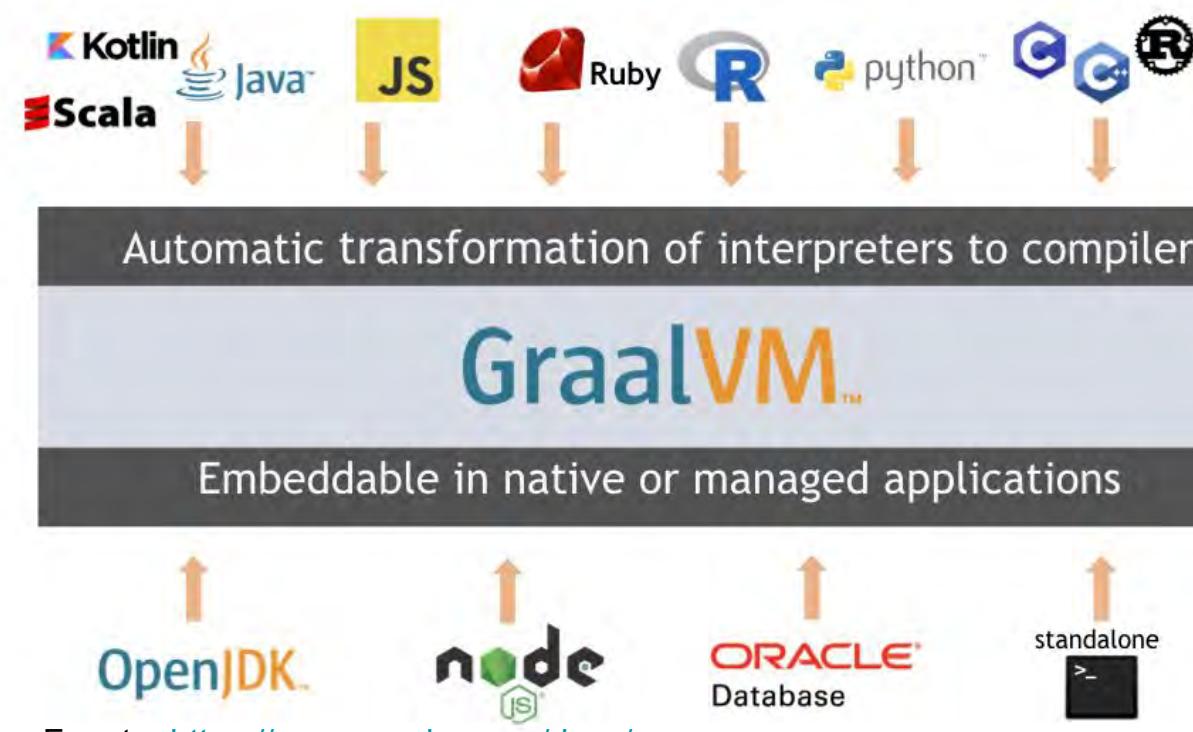
LENGUAJES Y RUNTIMES

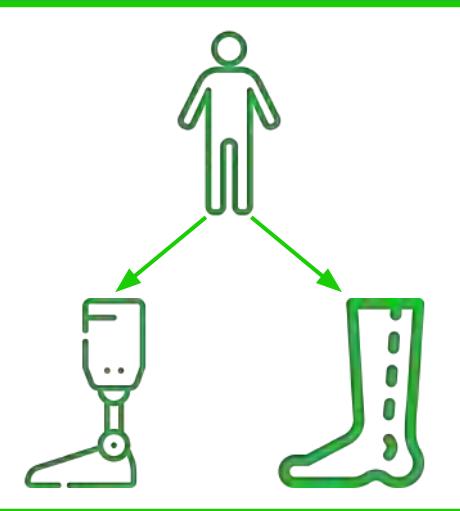
- GraalVM JavaScript: runtime de **JavaScript** ECMAScript 2019, con soporte para Node.js.
- TruffleRuby: implementación de **Ruby** con soporte preliminar para Ruby on Rails.
- FastR: implementación del lenguaje **R**.
- GraalVM Python: implementación de **Python 3**.
- GraalVM LLVM Runtime (SuLong): implementación de un interprete de bitcode **LLVM**.
- GraalWasm: implementación de **WebAssembly**.



COMPONENTES

- GraalVM Compiler: se trata de un compilador **JIT** para Java
- GraalVM Native Image: permite la compilación ***ahead-of-time***
- Truffle Language Implementation Framework: depende de GraalVM SDK y permite implementar **otros lenguajes** en GraalVM
- Instrumentation-based Tool Support: soporte para **instrumentación dinámica**, que es agnóstica del lenguaje.





¿En qué se diferencian?

Las interfaces en Java definen comportamientos, ofreciendo un catálogo de acciones posibles a los clientes que la utilizan. Una clase puede implementar esta interfaz, definiendo internamente cómo realizará las acciones ofrecidas.



¿CÓMO DEFINO UNA INTERFAZ?

Una interfaz se define así:

```
interface <nombre-interfaz> {
    // Definir firma de métodos aquí.
}
```

Luego, cualquier clase la puede implementar. Se verá obligada a implementar los métodos de la interfaz:

```
class <nombre-clase> implements <nombre-interfaz> {
    // Implementar los métodos de <nombre-interfaz>
}
```

Una clase puede implementar varias interfaces. A su vez, una interfaz puede extender de otras interfaces, agrupando varias firmas en una sola.

Los métodos de una interfaz pueden definir una implementación por defecto:

```
public default void greet() {
    System.out.println("Hola")
};
```



¿PARA QUÉ NOS SIRVE?

Las interfaces abstraen comportamientos que luego cualquier otra clase puede definir. Al momento de desarrollar, nos desliga de saber qué clase específica estamos utilizando, preocupándonos sólamente por los comportamientos que nos ofrece la interfaz.

```
interface Legs {
    void walk(NerveSignal signal);
}

class HidraulicLegs implements Legs {
    public void walk(NerveSignal signal) {
        // Activar mecanismos hidráulicos para caminar
    }
}

class BiologicalLegs implements Legs {
    public void walk(NerveSignal signal) {
        // Activar músculos para caminar
    }
}
```

```
Human humanA = new Human(BiologicalLegs());
Human humanB = new Human(HidraulicLegs());

humanA.walk();
humanB.walk();
```

```
class Human {
    private Legs legs;

    public Human(Legs legs) {
        this.legs = legs
    }

    public void walk() {
        this.legs.walk()
    }
}
```



¿Qué son?

Las anotaciones son una característica de Java que nos permite asociar un elemento de nuestro código a una serie de metadatos. Estos metadatos son utilizados por el compilador o durante la ejecución del programa, donde otros frameworks y librerías se pueden aprovechar para generar código o realizar otras operaciones.



¿CÓMO DECLARAR UNA ANOTACIÓN?

Una anotación se declara con `@` seguido por su nombre. **Debe preceder el elemento que queremos anotar.** Por ejemplo, un caso sencillo es la anotación `@Override`, indicando que el elemento siguiente sobreescribirá un elemento de la superclase donde está definida.

Algunas anotaciones pueden tener elementos asignables. En este caso se definen entre paréntesis después del nombre de la anotación:

```
@Author(nombre="José", apellido="Palacios")
```

Otras características:

- Se pueden declarar varias anotaciones para un mismo elemento.
- Normalmente una anotación ocupará su propia línea, pero es meramente una convención de estilo.

```
@Override  
@Author(nombre="Juan", apellido="Palacios")  
private String title;
```



EJEMPLO

Lombok es una librería que ofrece anotaciones para generar código. Una de sus anotaciones, `@Getter`, se asocia a una variable dentro de una clase:

```
public class LombokExample {  
  
    @Getter  
    private int someField;  
  
}
```

Al compilarla, se añadiría un método a la clase compilada:

```
public class LombokExample {  
  
    private int someField;  
  
    public int getSomeField() {  
        return this.someField;  
    }  
  
}
```



¿En qué consiste?

Es una interfaz para manipular fechas y horas, reemplazando las APIs de Java antiguas de las clases *Date* y *Calendar*. Sus principales ventajas con respecto a la antigua API son: la inmutabilidad, la seguridad en hilos concurrentes y el manejo de husos horarios.



Clases Principales

1. **LocalTime**: Representa una hora.
2. **LocalDate**: Representa una fecha.
3. **LocalDateTime**: Denota una fecha y una hora.
4. **ZonedDateTime**: Agrega el huso horario dentro de la representación.
5. **Period**: Abarca un espacio de tiempo, como los años o los días. Como ejemplo, se puede utilizar con las fechas para determinar rangos.
6. **Duration**: Especifica las duraciones de tiempo en nanosegundos, y se puede representar como segundos, horas, etc. Como ejemplo, esta clase se puede utilizar para extraer rangos de tiempo a partir de las clases que ofrecen una hora.

Cada una de estas clases contienen un abanico de operaciones y constructores para trabajar con fechas y horas. Se pueden crear fechas a partir de texto:

```
LocalDate.parse("2015-02-20");
```

Incluso se pueden añadir o restar días a la fecha resultante:

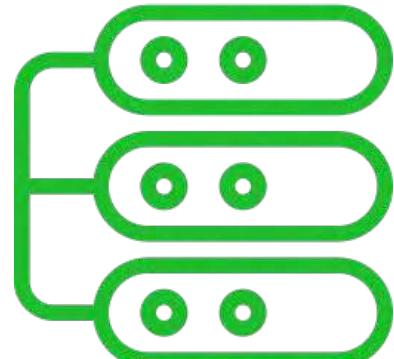
```
LocalTime.parse("06:30").plus(1, ChronoUnit.HOURS);
```



Características

- **Inmutable**: No se puede modificar una instancia de estas clases. El método *with* sirve como un “modificador” de una hora o fecha, devolviendo una instancia nueva con los nuevos valores deseados.
- **Seguridad de hilos**: Como consecuencia de la inmutabilidad, instancias de esta API se pueden utilizar dentro de hilos sin preocuparse de problemas de concurrencia que podrían ocurrir con instancias mutables.
- **Husos horarios**: La implementación previa de fechas y horas no ofrecían el manejo de husos horarios. El programador tenía que buscarse o implementarse uno. La nueva API ofrece esta capacidad de manejar horas y fechas con la clase *ZonedDateTime*.
- **Métodos consistentes**: Cada clase tiene un grupo de métodos con los mismos nombres, facilitando su uso. Por ejemplo, el método *of* se especifica en cada clase como la manera de inicializar un objeto de esa clase.
- **Estandarizado**: El diseño de la API se centra en el estándar ISO 8601 para fechas y horas.

Concurrencia



¿Qué es?

La **computación concurrente** es una forma de computación en la que varias tareas se ejecutan sin esperar a que la anterior haya terminado, aunque no necesariamente tienen que estar ejecutándose en el mismo instante.



SECUENCIAL, CONCURRENTE Y EN PARALELO

La ejecución de un programa suele realizarse siempre de manera **secuencial**. Si tenemos que realizar las tareas A y B, primero se realiza por completo la tarea A y cuando ha acabado se realiza la tarea B.

Hay ocasiones en las que no nos importa el orden en que se ejecuten las tareas. Esto puede hacerse usando **conurrencia**.

Un procesador mononúcleo solo puede ejecutar una operación a la vez, de manera que no se pueden ejecutar las tareas A y B a la vez. Si estamos usando concurrencia, nuestro procesador puede decidir ejecutar parcialmente la tarea A, luego cambiar y ejecutar la tarea B e ir ejecutando partes de estas tareas de manera intercalada. De esta manera conseguimos que las tareas A y B se estén ejecutando durante el mismo periodo de tiempo a pesar de que en cada instante de tiempo sólo se esté realizando una tarea.

Si tenemos más de un procesador o un procesador multinúcleo podemos ejecutar la tarea A en un núcleo y la tarea B en otro diferente de manera que ambas tareas se ejecutan en el mismo instante en **paralelo**.



DESVENTAJAS

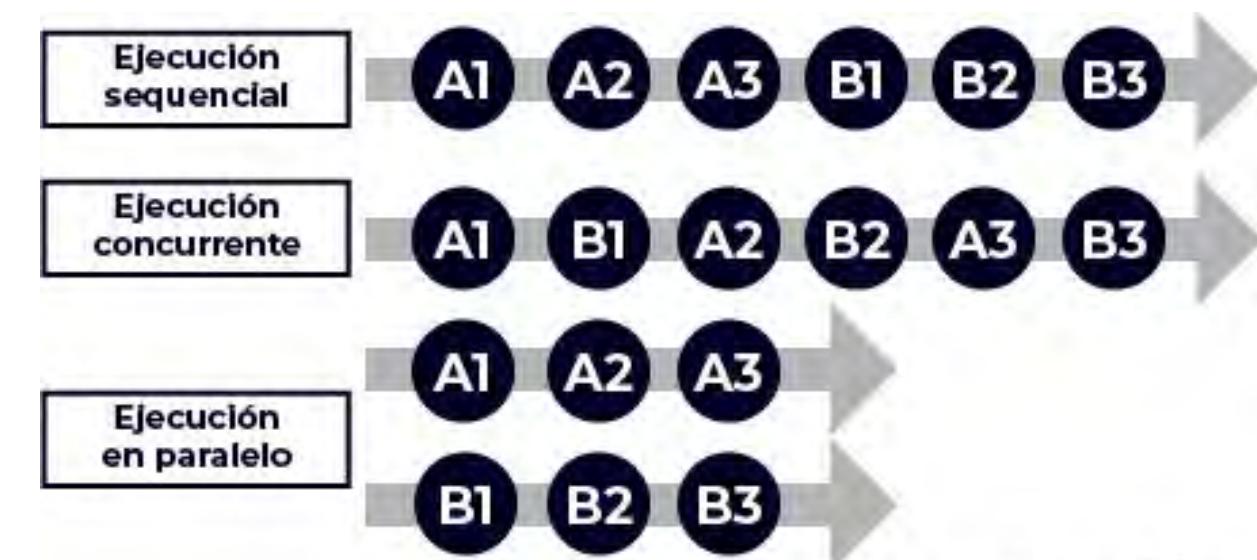
Al tener que gestionar la concurrencia, nuestro código será más complejo y más difícil de escribir y de mantener. Además, necesitamos tener en cuenta que el código ya no se ejecuta de manera secuencial y, por lo tanto, tenemos que tener especial cuidado con los recursos compartidos. Podemos tener, entre otros, condiciones de carrera, interbloqueos o inanición de procesos.

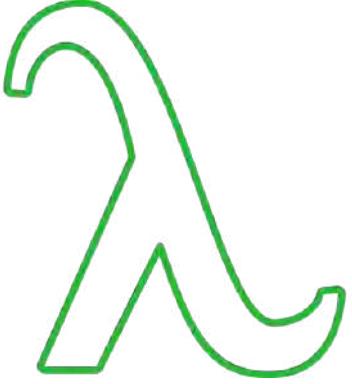
- **Condiciones de carrera:** Se dan cuando dos o más hilos acceden a un mismo recurso e intentan modificarlo a la vez. El resultado dependerá del orden en que se ejecuten los hilos.
- **Interbloqueo:** Se da cuando un hilo quiere acceder a un recurso que está siendo bloqueado por otro hilo y entra en estado de espera. Si ese otro hilo también está en estado de espera ambos hilos nunca saldrán del bloqueo..
- **Inanición:** Se da cuando un hilo tiene prioridad baja y nunca llega a ejecutarse porque se están ejecutando hilos con más prioridad.



VENTAJAS

- Aumento de la **eficiencia**.
- Disminución de **tiempos de espera**.
- Disminución de **tiempos de respuesta**.





¿Qué son?

Son expresiones que actúan como **funciones anónimas**. Pueden incluirse en cualquier lugar que acepte una **interfaz funcional**.



SINTAXIS

(Tipo1 param1, TipoN paramN) -> {cuerpo de la lambda}

Sin argumentos (paréntesis obligatorios)	<code>() -> System.out.println("Hola mundo!")</code>
1 argumento (paréntesis opcionales)	<code>cadena -> System.out.println(cadena)</code>
2 o más argumentos (paréntesis obligatorios)	<code>(x, y) -> x + y</code>
Con tipos explícitos (no es necesario incluir tipos)	<code>(int x, int y) -> x + y</code>
Con múltiples sentencias (se utilizan las llaves y return)	<code>(x, y) -> { System.out.println(x); System.out.println(y); return x + y; }</code>



INTERFACES FUNCIONALES

En Java, se considera interfaz funcional a toda interfaz que contenga **un único método abstracto**. Es decir, interfaces que tienen métodos estáticos o por defecto (default) seguirán siendo funcionales si solo tienen un único método abstracto.



¿DÓNDE SE USAN?

Donde se acepten **interfaces funcionales**. Por ejemplo:

- Al retornar:

```
private Predicate<Integer> isOddPredicate() {  
    return n -> n % 2 != 0;  
}
```

- En variables:

```
Predicate<Integer> isOdd = n -> n % 2 != 0;
```

- En llamadas a métodos:

```
IntStream.range(1, 11)  
.mapToObj(i -> String.format("i = %s", i))  
.forEach(System.out::println);
```



REFERENCIAS A MÉTODOS

Si la firma de la **interfaz funcional** coincide con la firma de otro método cualquiera, podemos usar una referencia al método en vez de una expresión lambda.

`System.out::println
this::miMetodo`

`super::metodoDeSuper
unObjeto::suMetodo`



¿En qué consiste?

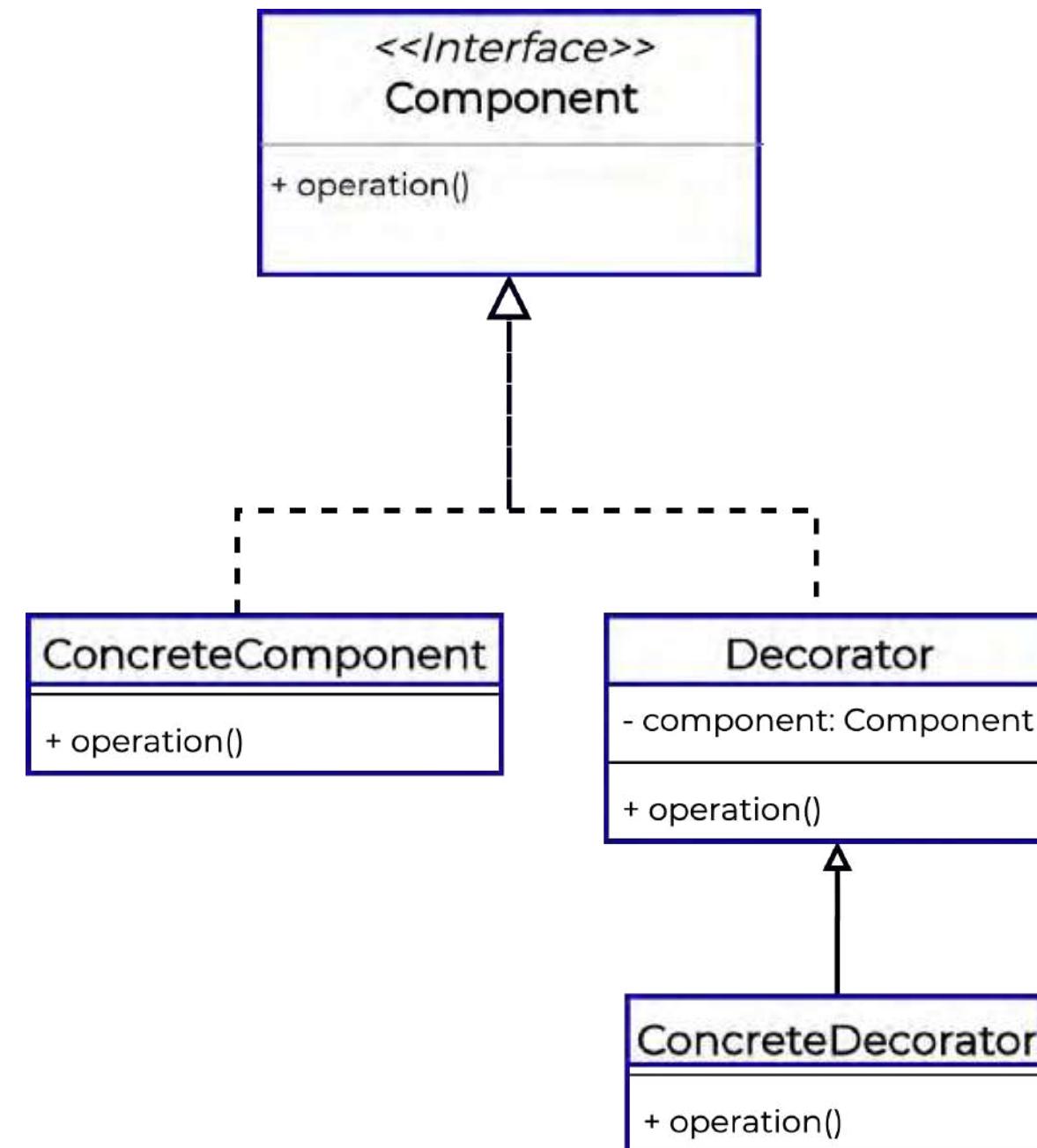
Patrón que **permite añadir nuevas funcionalidades a un objeto en tiempo de ejecución sin modificar su estructura** y a través de una envoltura (wrapper). El decorador envuelve la clase original sin cambiar la firma de los métodos existentes.



CONCEPTO

Decorator ofrece una alternativa cuando no es posible extender el comportamiento de un objeto a través de la herencia.

Normalmente tenemos una interfaz con varias implementaciones. Para aplicar este patrón, debemos crear una nueva ‘implementación’ que será nuestro *Decorator*. A partir de esta clase, creamos clases concretas de *Decorator* con las nuevas funcionalidades que se desean añadir.



Bibliografía

- <https://medium.com/@disenio2016/patr%C3%B3n-de-dise%C3%BAo-prototype-8447ee519165>
- <http://migranitodejava.blogspot.com/2011/05/prototype.html>
- <https://refactoring.guru>
- https://ikastaroak.birt.eus/edu/argitalpen/backupa/20200331/1920k/es/DAMDAW/ED/ED04/es_DAMDAW_ED04_Contenidos/website_13_patrones_de_refactorizacin_ms_habituales.html
- <http://www.lsi.us.es/docencia/get.php?id=5687>
- <http://rmlalves.blogspot.com/2014/01/cheat-sheet-fail-anti-patterns-and.html>
- <https://medium.com/@agrawall.lokesh/antipatterns-cheat-sheet-bcf820892e17>
- <https://www.adictosaltrabajo.com/2003/12/22/grasp/>
- http://www.cvc.uab.es/shared/teach/a21291/temes/object_oriented_design/slides/handouts/GRASP_patterns.pdf
- <https://jbravomontero.files.wordpress.com/2012/12/solid-y-grasp-buenas-practicas-hacia-el-exito-en-el-desarrollo-de-software.pdf>
- Head First Design Patterns (A Brain-Friendly Guide), Eric Freeman y Elisabeth Robson
- <https://blog.avenuecode.com/responsive-web-design-and-mobile-first-5-basic-technics>
- <https://web.dev/responsive-web-design-basics/>
- <https://blog.froont.com/9-basic-principles-of-responsive-web-design/>
- <https://desarrolloweb.com/articulos/conceptos-basicos-css-grid>
- https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
- https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript
- <https://smartbear.com/blog/test-and-monitor/understanding-the-open-web-stack/>
- <http://www.lesliesikos.com/open-web-platform/>
- Web Workers:**
 - <https://html.spec.whatwg.org/multipage/workers.html#navigator.hardwareconcurrency>
 - https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers
 - <https://www.html5rocks.com/en/tutorials/workers/basics/>
- React:**
 - <https://reactjs.org/>
- Angular:**
 - <https://angular.io/docs>
- Vue**
 - <https://www.adictosaltrabajo.com/2017/04/27/introduccion-a-vue-js/>

**Estas fichas son un recurso vivo,
¡anímate a hacernos sugerencias y comentarios!**



Puedes descargar la última versión en:

<https://autentia.com/libros/>

**Además encontrarás libros gratuitos, guías
y recursos útiles para tu día a día.**

