

## Construyendo aplicaciones distribuidas con Erlang/OTP



Compartir

f 21



♥ GUARDAR    💬 2 COMENTARIOS

4 Diciembre 2017 - Actualizado 5 Diciembre 2017, 13:18 **RUBENFA**

SUSCRÍBETE A GENBETA DEV

Recibe un email al día con nuestros artículos:

Tu correo electrónico **SUSCRIBIR**

Síguenos



Cualquiera que se haya enfrentado a la construcción de un sistema distribuido, se habrá dado cuenta que no es tarea fácil. Ya sea porque estamos construyendo un sistema a base de microservicios, porque estamos repartiendo un problema en partes para solucionarlas de forma paralela, o porque nuestro sistema tiene una concurrencia muy alta, nos enfrentaremos a una serie de problemas que son de sobra conocidos. Y es que hay muchos factores a tener en cuenta, como el control de la concurrencia, la sincronización de los datos o la tolerancia a fallos. La buena noticia es que si somos programadores de [Elixir](#) o Erlang, lo tendremos mucho más fácil gracias a OTP.

que nos permiten gestionar procesos y concurrencia con mucha más facilidad. OTP fue creado pensando en centralitas telefónicas, que por aquella época (hablamos de mediados de los 90), eran de los pocos sistemas altamente concurrentes que existían. Con el paso del tiempo, fueron apareciendo más problemas que OTP podía resolver y es que sus creadores consiguieron crear un modelo capaz de lidiar con conceptos como distribuido, tolerante a fallos, escalable, que funciona en tiempo real y altamente disponible. ¿Qué significan estos términos?

- **Escalable:** cuando un sistema puede adaptarse a cambios de carga o recursos disponibles.
- **Distribuido:** se refiere a cuando podemos agrupar sistemas y como interactúan unos con otros. Podemos crear grupos de sistemas de forma horizontal, por ejemplo añadiendo más máquinas hardware, para tener más recursos o añadir capacidad de proceso de forma vertical haciendo más potentes nuestras máquinas hardware virtualizadas.
- **Tolerante a fallos:** todo el sistema se comportará de forma previsible cuando se produzcan fallos. Si el sistema es tolerante a fallos, la latencia y la capacidad de respuesta no se verán mermadas en exceso y el sistema podrá continuar funcionando de forma normal.
- **Funcionamiento en tiempo real:** el tiempo de respuesta y la latencia serán constantes, y seremos capaces de devolver una respuesta en un tiempo razonable y normalmente bajo. Independientemente de las peticiones concurrentes que recibamos, deberemos ser capaces de responder a todas ellas.
- **Alta disponibilidad:** da igual que tengamos un *bug* en nuestro código, el sistema debe seguir funcionando. Es decir, que las actualizaciones del código, los parches u otras operaciones típicas de mantenimiento, no deben parar el sistema, que debe seguir funcionando de forma continua.

Los creadores de Erlang/OTP consiguieron crear un modelo capaz de lidiar con conceptos como distribuido, tolerante a fallos, escalable, que funciona en tiempo real y altamente disponible

Con OTP, y utilizando tanto Erlang, como Elixir, podemos conseguir controlar todas estas características de los sistemas distribuidos de forma robusta. ¿Y cómo consigue OTP hacer sencillo (o abordable) lo que es complejo? Pues con una mezcla de las siguientes características.

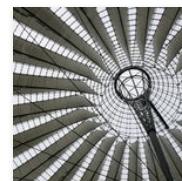
## Erlang/Elixir

Un lenguaje funcional es de ayuda a la hora de conseguir cierta seguridad a la hora de crear software distribuido, pero más importante es la inmutabilidad del mismo. **En otros lenguajes mutables, debemos recurrir a sistemas de sincronización de datos para evitar problemas acceso concurrente.** Semáforos, monitores, bloqueos etc. son palabras conocidas entre todos aquellos que nos hemos visto en la necesidad de programar alguna aplicación basada en hilos o procesos.

Con Erlang y Elixir es algo que tenemos solucionado desde la base, ya que al ser los datos inmutables, nos evitamos de un plumazo todos estos problemas. **Si las estructuras de datos de nuestros programas no pueden modificarse, no existirán problemas de concurrencia.**



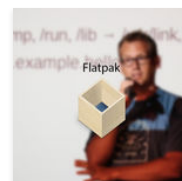
## TE RECOMENDAMOS



Principios de una arquitectura limpia: mantenible y testeable



Continuous Delivery en profundidad: problemas y soluciones alternativas comunes de los pipelines de Jenkins



Flatpak quiere revolucionar la instalación de apps en Linux con paquetes universales que funcionan en cualquier





## La máquina virtual BEAM

Otra de las patas importantes en OTP es la máquina virtual. Erlang y Elixir **corren sobre una máquina virtual conocida como BEAM**, que curiosamente son las siglas de Bogdan/Björn's Erlang Abstract Machine, nombres de dos programadores que trabajaban en Ericsson por la época.

En palabras de Joe Armstrong, uno de los coautores de Erlang *"Puedes emular la lógica de Erlang, pero si no corre sobre la máquina virtual de Erlang no puedes emular su semántica"*. Así que, por muy bonitos que sean los lenguajes de programación, sin una máquina virtual bien diseñada, no tendríamos muchas de las funcionalidades cubiertas.

"Puedes emular la lógica de Erlang, pero si no corre sobre la máquina virtual de Erlang no puedes emular su semántica". Joe Armstrong

---

El código que generamos con Erlang o Elixir (y algún lenguaje más) hay que compilarlo, para crear un archivo con extensión `.beam`. Ese archivo es al final el que se ejecuta sobre BEAM.

BEAM está optimizada para gestionar concurrencia, tiene un recolector de basura por cada proceso (haciendo que la recolección sea más sencilla y rápida) y que funciona de forma muy predecible y consistente en todos los casos.

## Herramientas y librerías

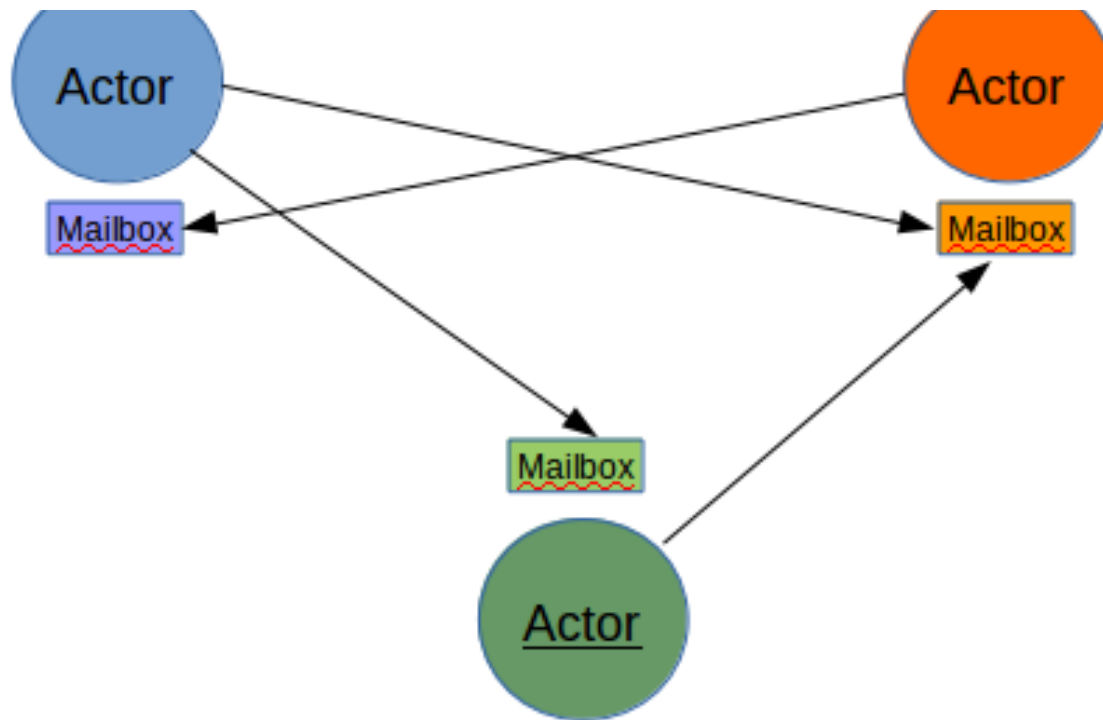
Además de Erlang y Elixir como lenguajes, y además de BEAM como máquina virtual, OTP incluye otra serie de añadidos que hacen toda la magia posible. Algunas de estas características son el Erlang runtime system (ERS), algunas librerías estándar (`stdlib`), bases de datos distribuidas como MNESIA, una colección de protocolos e interfaces para comunicarse con otros lenguajes de programación, como C o Java, herramientas de seguridad como SSL, sistemas de acceso a LDAP y un largo etc. así como un debugger gráfico y Observer para monitorizar procesos.

## Nodos

Los nodos son un conjunto de las herramientas anteriormente descritas, así como de herramientas de terceros, que funcionan sobre el sistema operativo. **Cada nodo, puede funcionar de forma independiente, pero se comunica con el resto de nodos de la red, permitiendo hacer nuestro sistema escalable de forma horizontal.**

Cada nodo puede conectarse a uno o varios nodos, de forma transitiva. Es decir, que si tenemos un nodo A, conectado a B, y conectamos B a C, C también estará conectado con A. Para gestionar la seguridad de los nodos se utiliza lo que se conoce como una **magic cookie**. Cuando se intenta una conexión entre nodos, se comprueba esta cookie y si coincide los nodos pueden conectarse. En otro caso se rechaza la conexión.

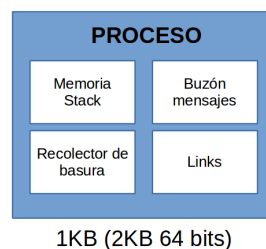
## Procesos



Aunque OTP está compuesta de muchas partes diferentes, podríamos decir que la parte principal son los procesos. Al final son los procesos los encargados de realizar las operaciones demandadas, y la gestión que hace OTP de ellos es parte fundamental en todo el sistema.

No debemos pensar en los procesos como si estuviéramos hablando de procesos del Sistema Operativo. En este caso **los procesos son mucho más livianos, lo que nos permite ejecutar muchísimos de forma concurrente sin que nuestro sistema se resienta**. De hecho un nodo puede ejecutar cientos de miles de procesos (incluso millones dependiendo de la potencia del hardware), sin afectar al rendimiento.

Un proceso en Erlang/Elixir está compuesto por su buzón de mensajes, su propio recolector de basura, un stack con la información necesaria y una zona para gestionar los enlaces a otros procesos. En conjunto, es probable que el tamaño no sea más que de 1Kb (2Kb en sistemas de 64 bits). Como veis los procesos son muy pequeños, lo cual hace que el cambio de contexto que tiene que realizar el procesador sea rapidísimo.



Pero la parte más importante es sin duda la comunicación entre procesos. Los procesos se comunican en base a un modelo de actores, o lo que es lo mismo, **los procesos no comparten memoria, y solo se comunican unos con otros a través del buzón de mensajes**. Una vez más esto nos evita muchos problemas de concurrencia.

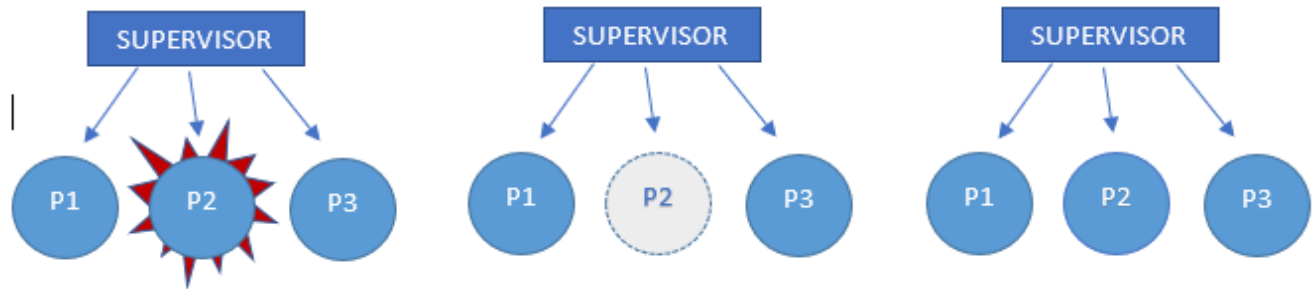
Si un proceso quiere comunicarse con otro, dejará un mensaje en el buzón del proceso destinatario, que el proceso receptor procesará cuando le sea posible. Gracias a este modelo de actores, nos evitamos los problemas relacionados con compartir memoria y hacemos mucho más sencillo el trabajo del recolector de basura.

Al tener la posibilidad de gestionar los procesos de forma independiente, se nos presentan interesantes opciones para crear estructuras jerárquicas de procesos de forma que sea mucho más sencillo gestionar los procesos. **Es aquí donde entran en**

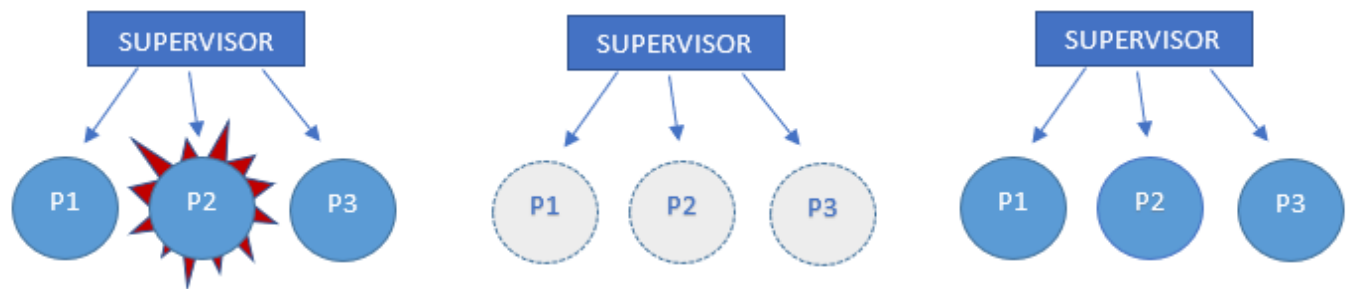
## Supervisores

Los supervisores son procesos que tienen el único objetivo de lanzar y monitorizar procesos *hijos*. Son capaces de detectar cuando un proceso que depende de él se ha detenido (por un fallo o por una ejecución normal), y dependiendo de su configuración, utilizar diferentes estrategias para su reinicio. Son las siguientes:

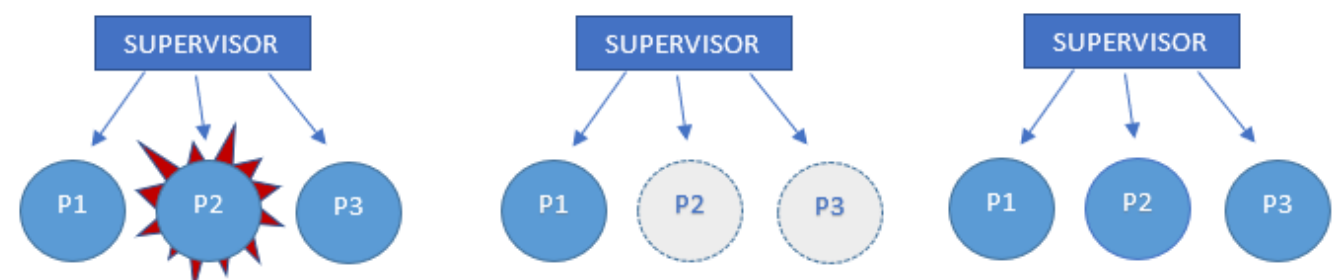
- **One for one:** si un proceso falla, se vuelve a reiniciar ese y sólo ese proceso.



- **One for all:** si un proceso falla, se detienen todos los procesos de ese supervisor y se vuelven a iniciar.



- **Rest for one:** si un proceso falla, además de él, se detienen todos los procesos que se hayan iniciado después y se vuelven a iniciar.



Por tanto la clave a la hora de usar supervisores, es asegurarse de que el orden de inicio está correctamente designado y la estrategia de reinicio elegida es la correcta.

Con todo esto podemos crear estructuras de supervisión más o menos complejas. Los fallos se pueden ir propagando hacia arriba en la jerarquía de supervisión. Si un proceso falla, su supervisor decidirá reiniciarlo. Si el problema se soluciona con ese reinicio, la ejecución continuará de forma normal. Pero si el proceso reiniciado vuelve a fallar, se seguirá intentando, hasta que se alcance un límite de intentos preconfigurado. Es ahí cuando el supervisor se detendrá y pasará el error a su propio supervisor. Si ningún reinicio soluciona el problema, es posible que se tomen medidas drásticas como reiniciar la máquina virtual, o incluso reiniciar la máquina.

**configuraciones y otros recursos.** Estos conjuntos son independientes unos de otros y es una forma de agrupar código para poder desplegarlo en cualquier parte. Por ejemplo podemos desplegar una aplicación en un nodo de Erlang y dicha aplicación podrá ser arrancada y detenida como un todo. Las aplicaciones pueden ser de tipo *normal* o de tipo *librería*. Las primeras arrancan un supervisor para poder gestionar los procesos dependientes, mientras que las de tipo librería no lo hacen, ya que no lo necesitan.

## GenServer

Un GenServer, implementa la típica estructura cliente servidor. Aunque con Erlang y Elixir pueden lanzarse procesos de forma manual, es mucho más sencillo crearlos a través de un GenServer. **Los GenServer se basan en comportamientos (behaviours en inglés), que definen una interface común para la comunicación entre procesos.** Esta interface utiliza llamadas `handle_call` (síncronas) y `handle_cast` (asíncronas), para realizar todas las operaciones requeridas. Los GenServer se pueden iniciar desde una función `start_link`, que suele ser utilizada, entre otras cosas, por los supervisores a la hora de arrancar el proceso.

Como hemos comentado antes, con OTP utilizamos un modelo de actores, y solo podemos comunicarnos con un proceso a través de su buzón de mensajes. Si utilizamos los `call`, nuestro proceso quedará a la espera de una respuesta del proceso remoto, mientras que si utilizamos `cast`, continuaremos la aplicación sin esperar ninguna respuesta.

## Actualización en caliente

Como decíamos, si un sistema que tiene que tener alta disponibilidad no puede detenerse para ser actualizado. **Debemos asegurar que el sistema es capaz de funcionar incluso cuando tenemos que aplicar parches para corregir *bugs* o para añadir nueva funcionalidad.**

Por suerte, con OTP, tenemos la posibilidad de utilizar la actualización de código en caliente. Para ello los módulos tienen que cargarse previamente, de lo que se encarga un componente de OTP conocido como *servidor de código*.

En el sistema puede haber hasta dos versiones de un mismo módulo, aunque inicialmente solo habrá una versión. Si realizamos algún cambio en el código, la versión existente pasará a ser la versión antigua, y la versión nueva pasará a ser la actual. Ambas versiones pueden seguir funcionando, ya que puede haber módulos que estén siendo utilizados por algún proceso en ejecución. Cuando sea posible, OTP irá actualizando los módulos de todos los procesos en ejecución. Si tenemos dos versiones y añadimos una tercera, la versión inicial será eliminada y los procesos que aun estén funcionando con ella serán detenidos.

## Tolerancia a fallos

Los nodos de Erlang/OTP entran en juego cuando queremos hacer que nuestra aplicación sea tolerante a fallos. Aunque al tener más de un nodo, nos encontramos con otros problemas típicos de la programación distribuida.

## Problemas en el paso de mensajes

Tenemos multitud de procesos en ejecución, que pueden ejecutarse en distintos nodos. Si hay fallos de red, sobrecarga de procesos o cualquier otro problema, cabe la posibilidad de que algún mensaje se pierda. En este caso podemos seguir tres estrategias distintas:

- **Al menos uno:** imagina que tenemos un servidor web en el que queremos iniciar sesión. Si la primera petición falla, podemos intentarlo en otro nodo. Si el segundo nodo funciona, nos quedamos con la sesión de este. Es posible que la primera petición acabe funcionando (aunque con retraso), pero nosotros la ignoraremos.
- **Como mucho uno:** imaginemos que un sistema que envía SMS. Si nuestro sistema envía millones de mensajes al día, es posible que la pérdida de algunos mensajes sea asumible y no nos preocupe. En ese caso realizamos la petición de envío de SMS y nos olvidamos.

error o incluso recibió la petición, pero lo que se ha perdido ha sido la respuesta. En cualquier caso, deberemos pensar en estos problemas a la hora de utilizar esta estrategia.

## Problemas con los datos compartidos

Si tenemos varios nodos funcionando, nos encontraremos con el problema de los datos compartidos entre ellos. En este caso podemos seguir varias estrategias:

- **No compartir nada:** los procesos de un nodo tienen cada uno su versión de los datos y de su estado actual y no lo comparten con ningún otro nodo. Esto hace que la escalabilidad del sistema sea predecible y lineal. El problema de esta estrategia, es que si perdemos el nodo, también perdemos los datos y el estado actual de los procesos.
- **Compartir una parte:** si queremos asegurar de que aunque un nodo falle, podamos conservar los datos más críticos, utilizamos esta estrategia. Los datos y el estado se irán copiando entre nodos, para asegurarnos de tener una copia en cada uno. Esto reduce algo el rendimiento, y nos crea el problema de que si un nodo se reinicia, tiene que volver a adquirir los datos de todos los procesos.
- **Compartir todo:** en este caso no nos podemos permitir que se pierda un solo dato, y debemos asegurarnos de que una transacción se ejecute una sola vez. Esta técnica es la más segura, pero también nos obliga a sacrificar escalabilidad.

## Consistencia vs disponibilidad

Aunque las soluciones reales no son tan simples, [el teorema de CAP](#) ya nos indica que **todo sistema distribuido tiene que elegir entre consistencia, disponibilidad y tolerancia a particiones**. Y cuando diseñamos nuestra aplicación distribuida, es algo que debemos tener en cuenta.

Por ejemplo en las estrategias para gestionar el paso de mensajes, podemos ver que dependiendo de la que elijamos tendremos que sacrificar o bien la consistencia o bien la disponibilidad. La estrategia *al menos uno* es muy escalable, pero no muy consistente, mientras que la estrategia *exactamente uno* es muy consistente, pero mucho menos escalable, por lo que la disponibilidad se resiente.

Lo mismo nos pasa con el tema de compartir datos. *Compartir todo* hace que seamos mucho más fiables, pero que nuestra disponibilidad sea menor. Si *no compartimos nada*, pasa justamente lo contrario, somos menos fiables, pero nos aseguramos una alta disponibilidad.

## Conclusión

**Erlang/Elixir y OTP nos proporcionan muchas herramientas para que construir sistemas distribuidos sea mucho menos doloroso que con otras plataformas y lenguajes de programación.** Aun así, no es tarea fácil y hay muchos aspectos que deberemos tener en cuenta para asegurar que nuestra aplicación sea escalable, tolerante a fallos, altamente disponible etc.

En definitiva, construir sistemas distribuidos es difícil, pero muy divertido.

Imagen | [Mathias.Pastwa](#)

Compartir

f 21  

Temas:

FRAMEWORKS  ERLANG



TAMBIÉN TE PUEDE GUSTAR



Por qué un editor de texto de hace 40 años machaca al "todopoderoso" Atom



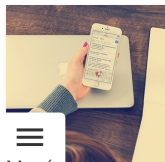
Experiencias en el desarrollo de aplicaciones SPA corporativas



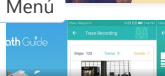
Hogar, dulce hogar (conectado)



Caos en la seguridad WiFi: un repaso a las vulnerabilidades de WEP, WPA, y WPA2




¿Por qué no te sirven las aplicaciones de productividad?





Compartir  
"Construyendo aplicaciones distribuidas con Erlang/OTP"

f 21  

 hace 15 minutos #2

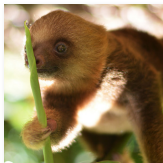
Muchas de estas aplicaciones de este programa acelerar el funcionamiento del dispositivo.

 [ESCRIBIR COMENTARIO](#)



## RECOMENDADO EN MAGNET



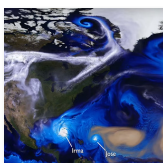
Ten cuidado con los likes que das en Instagram: podrías estar fomentando el maltrato animal



Tatsuo Horiuchi, el anciano japonés que pinta detalladísimos y coloridos cuadros con ¡Excel!



Muerte en Roma: el mapa que ilustra dónde nacieron y murieron todos los emperadores romanos



La flipante visualización de la NASA que ilustra cómo nacen y mueren los huracanes atlánticos



## RECIBE UN EMAIL AL DÍA CON LOS ARTÍCULOS DE GENBETA DEV:

Tu correo electrónico  [SUSCRIBIR](#)

Síguenos



## EN GENBETA DEV HABLAMOS DE...

Trabajar como desarrollador

Desarrollo web

Herramientas

Curiosidades

Frameworks

Eventos para Desarrolladores

Actualidad

JavaScript

Formación

Open Source

Herramientas de desarrollo

java

[VER MÁS TEMAS](#)

SUBIR ▲

TECNOLOGÍA	ESTILO DE VIDA	MOTOR	ECONOMÍA	OCIO
Xataka	Tendencias	Motorpasión	El Blog Salmón	Espinof
Xataka Móvil	Tendencias Belleza	Motorpasión Moto	Pymes y Autónomos	Diario del Viajero
Xataka Foto	Tendencias Hombre			
Xataka Android	Directo al Paladar			
Xataka Smart Home	Bebés y Más			
Xataka Windows	Vitónica			
Xataka Ciencia	Decoesfera			
Applesfera				
Vida Extra				
Genbeta				
Genbeta Dev				
Magnet				
Compradiccion				
Xataka eSports				

LATINOAMÉRICA				
Xataka México	Directo Al Paladar México	Motorpasión México		
Xataka Colombia				

PARTICIPAMOS EN				
Nobbot	Mi Mundo Philips	Circula Seguro	En Naranja	Bluemagazine
Tecnología de tú a tú	Muy Saludable de Sanitas	Circula Seguro PT	Sage Experience	
Blog Lenovo	Coca-Cola Journey España	Corriente Eléctrica	Bloggin Zenith	
eSports Unlocked by Orange	Coca-Cola Journey México		Seguros de tú a tú	
Inget by acer	Coca-Cola Journey Portugal			
	Zona Coca-Cola			
	Cervezas Alhambra			
	Mahou Rentabilibar			

