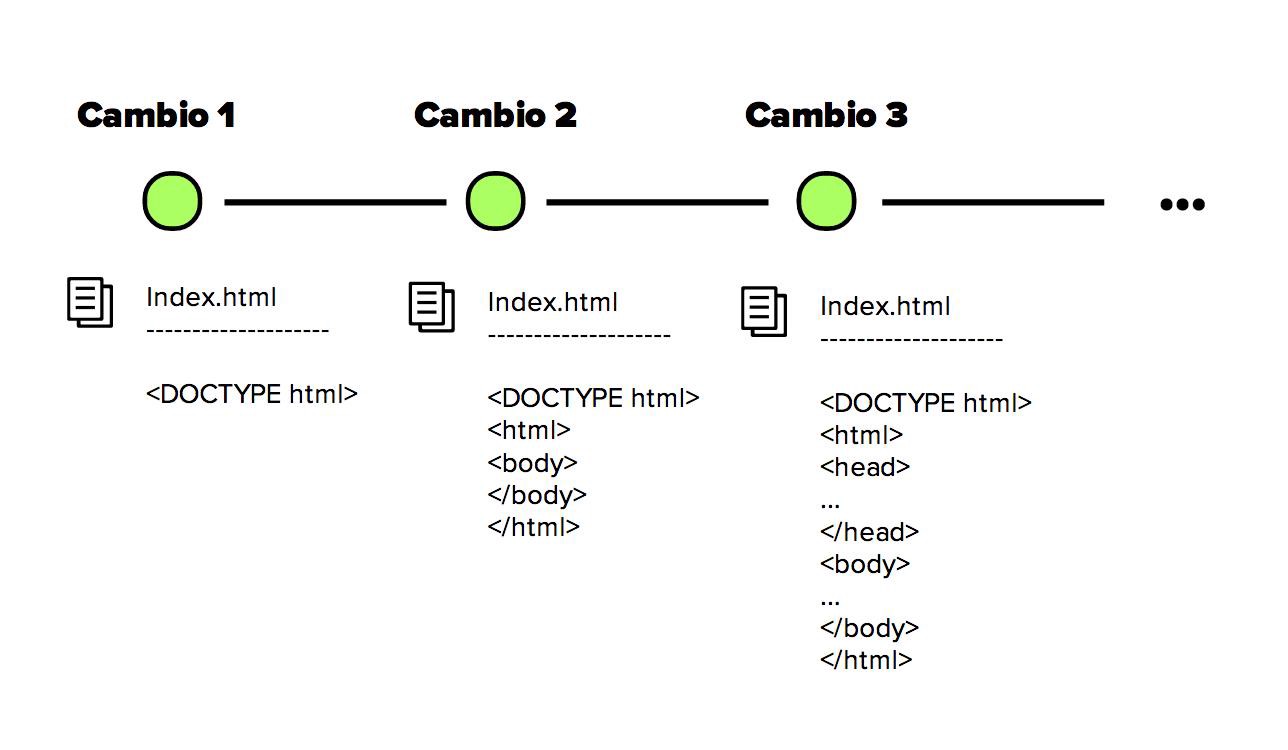
# Control De Versiones

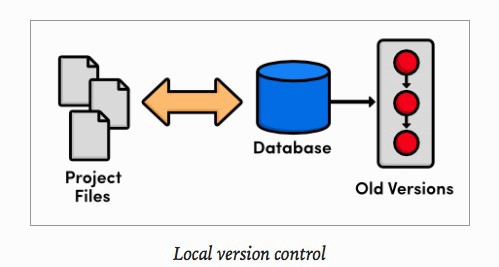
El control de versiones es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo de tal manera que sea posible recuperar versiones especificas más adelante.



Los sistemas de control de versiones han ido evolucionando a lo largo del tiempo y podemos clasificarlos en tres tipos: Sistemas de Control de Versiones Locales, Centralizados y Distribuidos.

**Sistemas de Control de Versiones Locales**

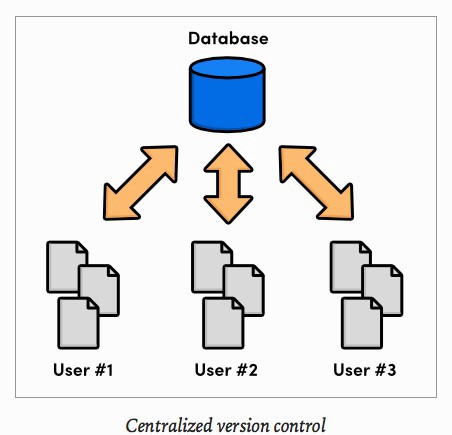
Los sistemas de control de versiones locales en vez de mantener las versiones como archivos independientes, los almacenaban en una base de datos. Cuando era necesario revisar una versión anterior del proyecto se usaba el sistema de control de versiones en vez de acceder directamente al archivo, de esta manera en cualquier momento solo se tenia una copia del proyecto, eliminando la posibilidad de confundir o eliminar versiones.



En este punto el control de versiones se llevaba a cabo en el computador de cada uno de los desarrolladores y no existía una manera eficiente de compartir el código entre ellos.

**Sistemas de Control de Versiones Centralizados**

Para facilitar la colaboración de múltiples desarrolladores en un solo proyecto los sistemas de control de versiones evolucionaron: en vez de almacenar los cambios y versiones en el disco duro de los desarrolladores, estos se almacenaban en un servidor.



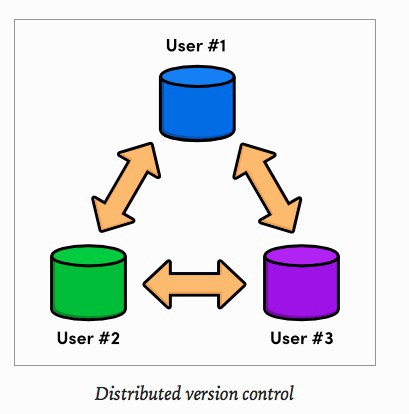
Sin embargo, aunque el avance frente a los sistemas de control de versiones locales fue enorme, los sistemas centralizados trajeron consigo nuevos retos: ¿Cómo trabajaban múltiples usuarios en un mismo archivo al mismo tiempo?

Los sistemas de control de versiones centralizados abordaron este problema impidiendo que los usuarios invalidaran el trabajo de los demás. Si dos personas editaban el mismo archivo y se presentaba un conflicto alguien debía solucionar este problema de manera manual y el desarrollo no podía continuar hasta que todos los conflictos fueran resueltos y puestos a disposición del resto del equipo.

Esta solución funcionó en proyectos que tenían relativamente pocas actualizaciones y por ende pocos conflictos pero resulto muy engorroso para proyectos con docenas de contribuyentes activos que realizaban actualizaciones a diario.

**Sistemas de Control de Versiones Distribuidos**

La siguiente generación de sistemas de control de versiones se alejo de la idea de un solo repositorio centralizado y optó por darle a cada desarrollador una copia local de todo el proyecto, de esta manera se construyo una red distribuida de repositorios, en la que cada desarrollador podía trabajar de manera aislada pero teniendo un mecanismo de resolución de conflictos mucho más elegante que un su versión anterior.



Al no existir un repositorio central, cada desarrollador puede trabajar a su propio ritmo, almacenar los cambios a nivel local y mezclar los conflictos que se presenten solo cuando se requiera. Cómo cada usuario tiene una copia completa del proyecto el riesgo por una caída del servidor, un repositorio dañado o cualquier otro tipo de perdida de datos es mucho menor que en cualquiera de sus predecesores.

# <2>

Los sistemas de control de versiones son herramientas que gestionan los diversos cambios que se realizan sobre los elementos de uno o más proyectos, guardando así versiones de nuestra aplicación en todas sus fases de desarrollo.

Cuando hablamos de una versión, revisión o edición de un producto, nos referimos al estado en el que se encuentra el mismo en un momento dado de su desarrollo o modificación. Una revisión donde se nos informa la fecha en la cual se ha creado, quien la ha ejecutado y por qué motivos lo ha hecho.

Estas versiones son como una instantánea / snapshot donde se registra el estado del proyecto en un momento determinado en el tiempo y se van guardando, a medida que se hacen, modificaciones al código fuente, se agregan assets, frameworks o librerías, etc.

Es como si congeláramos el tiempo cada vez que apliquemos un cambio importante para luego, en caso preciso, poder volver atrás y reintentarlo desde este punto.

Me imagino que una analogía también válida y con la que la mayoría hemos interactuado serían los checkpoint en algunos juegos, las revisiones que gestionan WordPress para con nuestras publicaciones o el undo y redo que muchas veces ejecutamos sobre la edición de una foto o video o más frecuente en un documento de Word o Pages.

Creo que ya vamos entendiendo que el control de versiones lo podemos aplicar a cualquier tipo de proyecto que estemos creando en nuestro ordenador, no necesariamente de software. Pudiera ser aplicado al proceso de escribir un libro o tesis de grado, la edición de una foto o video podemos definirlo:

En un sistema de control de versiones se registran los cambios realizados sobre un archivo o conjunto de estos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.

Además de todo lo antes comentado y haciendo un énfasis en el desarrollo de software: los sistemas de control de versiones dan soporte a esa necesidad que surge en todo proyecto de trabajar en distintas ramas al mismo tiempo, introduciendo cambios a nuestra aplicación, tanto en la rama de desarrollo como a la que tenemos en producción.

Teóricamente, las nuevas funcionalidades de nuestra aplicación las iremos trabajando dentro de la rama de desarrollo, pero constantemente uno se encuentra resolviendo bugs, tanto en la rama de producción como en la de desarrollo.

A esto me refería con movernos en el tiempo, podemos ir tanto hacia atrás como hacia adelante, podemos trabajar en varias ramas (branch) que existen en paralelo dentro de la misma línea de tiempo o historial de nuestra aplicación, la rama estable, testing o de desarrollo… ya esto lo determina cada equipo y la metodología de trabajo que se aplique (SCRUM, Modelo en Cascada… etc).

Evolución en el Control de Versiones

Como todo en este mundo, los sistemas de control de versiones han sufrido una evolución hasta nuestros días. Comenzaron a aparecer allá por los años setenta, aunque al principio eran bastante elementales. Para que se hagan una idea, en los primeros sistemas existía una restricción por la que sólo una persona podía estar a la vez tocando el mismo código.

Es evidente que esto provocaba un gran retraso en los equipos de trabajo, por lo que se convirtió junto a otros aspectos en esa fuerzas de cambio que a lo largo de los años impulsó el surgimiento de nuevos sistemas de control de versiones cada vez mejores y siempre en consecuencia con las necesidades de los equipos de desarrollo.

A continuación hablaremos sobre los tres tipos o variantes principales que han surgido:

Sistema de Control de Versiones Local

Un método de control de versiones que todos hemos aplicado es copiar archivos a otros directorios, especificando variaciones en los nombres (archivo1, archivo2, archivo-final… etc) o quizás indicando la fecha y hora en que estos fueron modificados.

Estos son meros ejemplos de una gestión de versiones vetusta, que aún hoy se siguen aplicando dada la simpleza de esta solución, pero claramente no es para nada óptima y sí tremendamente propensa a errores.

Es fácil olvidar en qué directorio te encuentras, y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías.

Para hacer frente a este problema, hace tiempo se crearon los primeros sistemas de versiones locales, estos contenían una simple base de datos en la que se llevaba un registro de todos los cambios realizados sobre los archivos.

Remarcar que estamos hablando de un sistema local que no contaba con ningún respaldo en la nube y que estaba expuesto a cualquier problema de hardware o sistema operativo por lo que hacer backup de este sistema y sus archivos era mandatory.

Una de las herramientas de control de versiones más popular fue un sistema llamadorcs, que todavía podemos encontrar en muchos de los ordenadores actuales. Hasta macOS incluye el comandorcscuando instalas las herramientas de desarrollo.

Esta herramienta funciona básicamente guardando conjuntos de parches (es decir, las diferencias entre archivos) de una versión a otra en un formato especial en disco; puede entonces recrear cómo era un archivo en cualquier momento sumando los distintos parches.

Sistemas de Control de Versiones Centralizados

Estos sistemas, como CVS, Subversion, y Perforce, tienen un único servidor que contiene todos los archivos versionados, y varios clientes descargan los archivos desde ese lugar central. Durante muchos años éste fue el estándar para el control de versiones y como sucede con todo en un inicio vino a resolver varios problemas y ofrecía sólidas ventajas frente a los sistemas de versiones locales.

Sin embargo, esta configuración también tiene serias desventajas. La más obvia es el punto único de fallo que representa el servidor centralizado. Si ese servidor se cae durante una hora, entonces durante esa hora nadie puede colaborar o guardar cambios versionados de aquello en que están trabajando.

Si el disco duro en el que se encuentra la base de datos central se corrompe, y no se han llevado copias de seguridad adecuadamente, pierdes absolutamente todo, toda la historia del proyecto salvo aquellas instantáneas que puedan haber en las máquinas locales de los desarrolladores.

Los sistemas de controles de versiones locales sufren de este mismo problema, cuando tienes toda la historia del proyecto en un único lugar, te arriesgas a perderlo todo.

Sistemas de Control de Versiones Distribuidos

Es aquí donde entran los sistemas de control de versiones distribuidos (Distributed Version Control Systems en inglés). En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no sólo descargan la última instantánea de los archivos: también replican completamente el repositorio.

Así, si un servidor muere, y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios de los clientes puede copiarse en el servidor para restaurarlo. Cada vez que se descarga una instantánea, en realidad se hace una copia de seguridad completa de todos los datos.

Es más, muchos de estos sistemas se las arreglan bastante bien teniendo varios repositorios con los que trabajar, por lo que puedes colaborar con distintos grupos de desarrolladores simultáneamente dentro del mismo proyecto. Esto te permite establecer varios flujos de trabajo que no son posibles en sistemas centralizados, como pueden ser los modelos jerárquicos.

# Tecnología GIT

En un nivel más alto, GitHub es un sitio web y un servicio en la nube que ayuda a los desarrolladores a almacenar y administrar su código, al igual que llevar un registro y control de cualquier cambio sobre este código. Para entender exactamente qué es GitHub, primero usted necesita conocer los dos principios que lo conectan:

* Control de versión
* Git

En este artículo, explicamos estos dos principios. Luego, nos profundizaremos más sobre GitHub y cómo puede hacer que funcione GitHub para trabajar con WordPress y/o Kinsta.

**¿Qué Es una Versión de Control?**

Una Versión de Control ayuda a los desarrolladores llevar un registro y administrar cualquier cambio en el código del proyecto de software. A medida que crece este proyecto, la versión de control se vuelve esencial. Tomemos WordPress por ejemplo…

En este punto, WordPress es un gran proyecto. Si un desarrollador importante deseara trabajar en una parte en específico de la base de datos del código de WordPress, no sería seguro o eficiente permitir que pueda editar directamente sobre el código fuente “oficial”

Por esto, la versión de control permite a los desarrolladores trabajar de forma segura a través de una **bifurcación** y una **fusión**.

Con la **bifurcación**, un desarrollador duplica parte del código fuente (llamado **repositorio**). Este desarrollador, luego puede, de forma segura, hacer cambios a esa parte del código, sin afectar al resto del proyecto.

Luego, una vez que el desarrollador logre que su parte del código funcione de forma apropiada, esta persona podría **fusionar** este código al código fuente principal para hacerlo oficial.

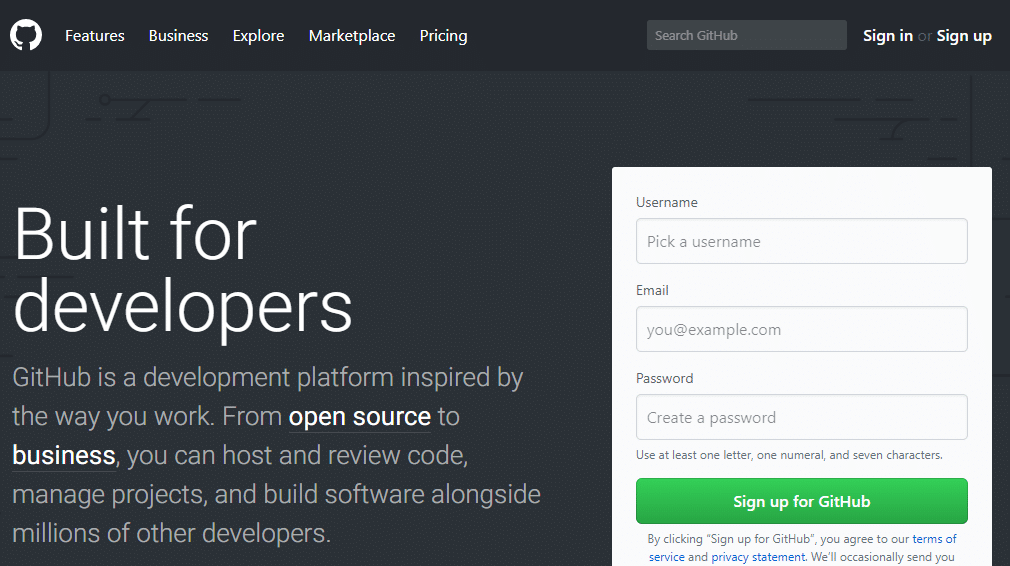
Todos estos cambios luego son registrados y pueden ser revertidos si es necesario.

**¿Qué Es Git?**

Git es un **sistema de control específico de versión de fuente abierta** creada por Linus Torvalds en el 2005.

Específicamente, Git es **un sistema de control de versión distribuida**, lo que quiere decir que la base del código entero y su historial se encuentran disponibles en la computadora de todo desarrollador, lo cual permite un fácil acceso a las bifurcaciones y fusiones.

Según la encuesta entre los desarrolladores de [Stack Overflow](https://insights.stackoverflow.com/survey/2018/" \l "work-version-control" \t "_blank), más de 87% de los desarrolladores usan Git.

[](https://kinsta.com/es/wp-content/uploads/2018/05/qu%C3%A9-es-github-2.png)

La página principal de GitHub

GitHub es una compañía sin fines de lucro que ofrece un servicio de hosting de repositorios almacenados en la nube. Esencialmente, hace que sea más fácil para individuos y equipos usar Git como la versión de control y colaboración.

La interfaz de GitHub es bastante fácil de usar para el desarrollador novato que quiera aprovechar las ventajas del Git. Sin GitHub, usar un Git generalmente requiere de un poco más de conocimientos de tecnología y uso de una línea de comando.

GitHub es tan fácil de usar, que incluso algunas personas usan GitHub para administrar otro tipo de proyectos – como [escribir libros](http://braythwayt.com/2015/01/29/how-i-write-books-with-github-and-leanpub.html).

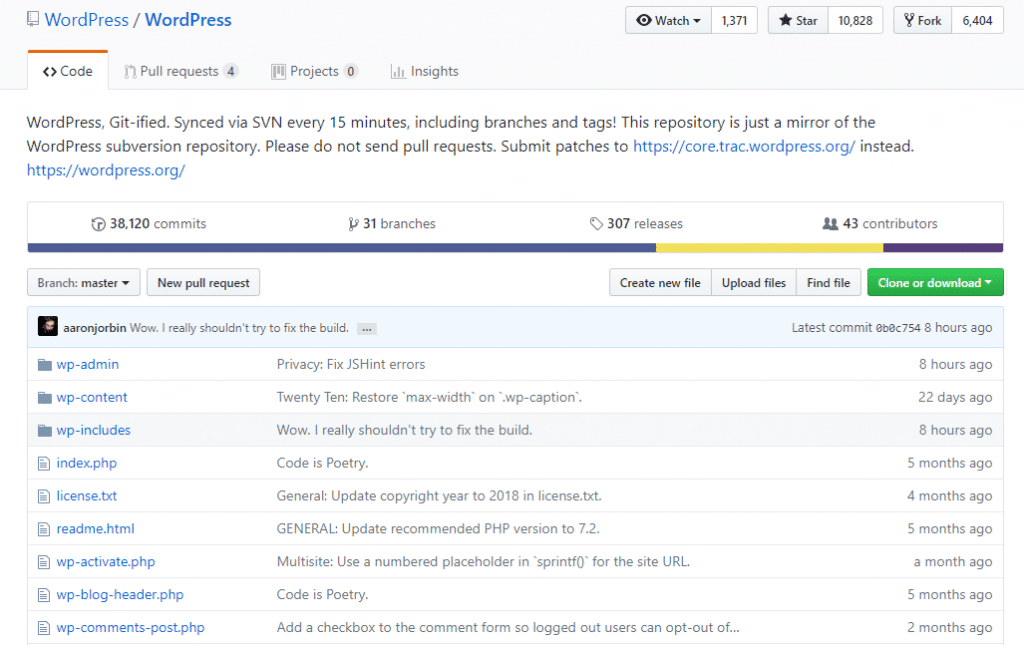
Además de esto, cualquier persona puede inscribirse y ser hospedar un repositorio de código público completamente gratuito, el cual hace que GitHub sea especialmente popular con proyectos de fuente abierta.

Como compañía, GitHub hace dinero vendiendo alojamiento para repositorios de código privado, al igual que otros planes enfocados para negocios, que hace más fácil que las organizaciones administren a los miembros de su equipo y su seguridad. Utilizamos GitHub de forma extensiva aquí en Kinsta, para administrar y desarrollar proyectos internos.

Final del formulario

**Explorando la Interfaz de GitHub**

Para tener un entendimiento básico de como luce la interfaz de GitHub, aquí hay un código fuente de WordPress [alojado en un repositorio de GitHub](https://github.com/WordPress/WordPress):

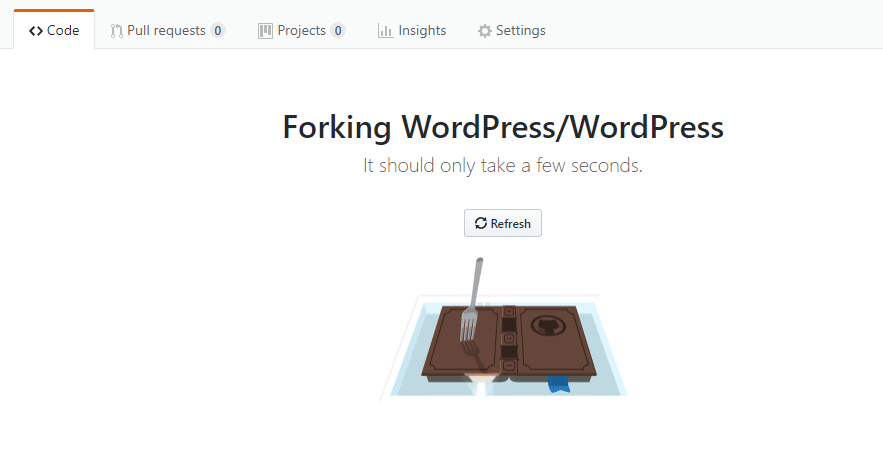
[](https://kinsta.com/es/wp-content/uploads/2018/05/qu%C3%A9-es-github-3.png)

El código de WordPress en GitHub

Desde ahí, podrá ver varias bifurcaciones que están siendo modificadas, al igual podrá ver cuando alguien hace un **commit** (una forma de “guardar” un archivo). Dependiendo de cómo haya sido establecido el repositorio, también podrá crear su propia **bifurcación** y sus propios **commits**.

Una vez que haya hecho algunos cambios, usted puede enviar ese código de nuevo a la bifurcación, con tan solo hacer una **solicitud de pull**. Una solicitud de pull es básicamente pedirle a la persona a cargo de la bifurcación que incluya su código. Y este también ayuda a que esta persona pueda ver exactamente qué fue lo que cambió en el código.

Si quisiera editar un poco o bastante del código fuente de WordPress en su propia cuenta de forma relativamente permanente. También podría hacerle **fork** con tan solo dar click en el botón de **Fork** (un fork es similar en concepto a la bifurcación, pero el fork es relativamente permanente):

[](https://kinsta.com/es/wp-content/uploads/2018/05/qu%C3%A9-es-github-4.png)

Un ejemplo de Fork en un código

WordPress por si mismo [fue originalmente un fork de b2/cafelog](https://kinsta.com/es/aprender/wordpress-historia/). Si quiere obtener un poco más de información de cómo puede usar GitHub, la [guía GitHub Hello World](https://guides.github.com/activities/hello-world/) ofrece un tutorial intuitivo para que usted pueda empezar.

**Cómo Usar Git y GitHub en Kinsta**

Kinsta ofrece un cliente Git para que pueda [usar SSH](https://kinsta.com/es/base-de-conocimiento/conectarse-a-ssh/) y descargar su propio repositorio Git de GitHub (o incluso servicios similares).

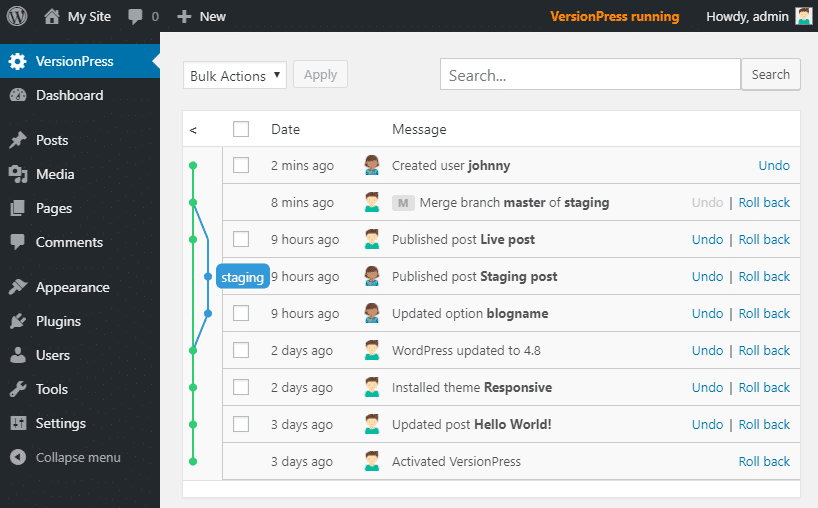
Si usted está interesado en usar Git o GitHub en Kinsta, por favor lea e[ste artículo de apoyo para poder ver las instrucciones completas](https://kinsta.com/es/base-de-conocimiento/git/).

**Otras Formas de Usar Git y GitHub con WordPress**

Muchos desarrollador de temas y plugins de WordPress alojan sus proyectos de código fuente en GitHub. Por ejemplo, usted puede ver el código constructor de la Página [Elementor](https://github.com/pojome/elementor" \t "_blank), el código fuente del tema [Zerif Lite](https://github.com/Codeinwp/zerif-lite" \t "_blank), y muchos más.

Pero también hay plugins de WordPress que le podrán ayudar a usar Git con WordPress de una forma más directa.

Por ejemplo, [VersionPress](https://versionpress.net/" \t "_blank) tiene como objetivo brindar poder del Git a las acciones regulares de WordPress, como actualizar un post o instalar un plugin. VersionPress agrega una versión de control para los archivos y las bases de datos de su WordPress.

[](https://kinsta.com/es/wp-content/uploads/2018/05/qu%C3%A9-es-github-5.png)

VersionPress

[WP Pusher](https://wppusher.com/) es otro plugin usando Git que le permite lanzar temas y plugins directamente desde GitHub a su sitio de WordPress.

**Cómo Empezar con GitHub**

Para empezar con GitHub:

* [Inscríbase](https://github.com/join) para obtener una cuenta gratis de GitHub
* Siga la [guía de GitHub](https://guides.github.com/activities/hello-world/) para entender las acciones más populares que es muy posible que usted quiera hacer.

# Repositorio En La nube GITHUB

Un repositorio es como una carpeta para tu proyecto. El repositorio de tu proyecto contiene todos los archivos de tu repositorio y almacena el historial de revisión de cada archivo. También puedes debatir y administrar el trabajo de tu proyecto dentro del repositorio.

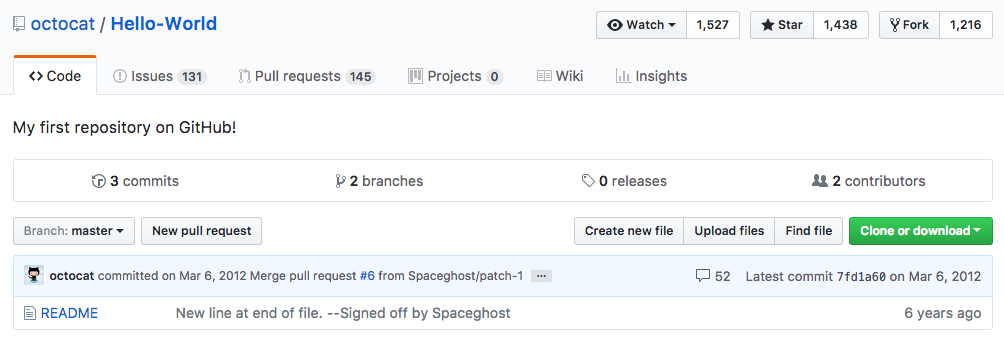
Puedes ser propietario de repositorios individualmente o puedes compartir la propiedad de los repositorios con otras personas en una organización.

You can restrict who has access to a repository by choosing the repository's visibility. For more information, see "[About repository visibility](https://help.github.com/es/github/creating-cloning-and-archiving-repositories/about-repository-visibility)."

Para los repositorios que son propiedad de un usuario, les puedes dar a otras personas acceso de colaborador para que puedan colaborar en tu proyecto. Si un repositorio es propiedad de una organización, les puedes dar a los miembros de la organización permisos de acceso para colaborar en tu repositorio. Para obtener más información, consulta "[Niveles de permiso para un repositorio de cuenta de usuario](https://help.github.com/es/articles/permission-levels-for-a-user-account-repository)" y "[Niveles de permiso de repositorio para una organización](https://help.github.com/es/articles/repository-permission-levels-for-an-organization)".

Cada persona y organización puede ser propietario de una cantidad ilimitada de repositorios públicos e invitar a una cantidad ilimitada de colaboradores a los repositorios públicos. Con GitHub Free, puedes usar una cantidad ilimitada de repositorios privados gratuitos con una configuración de características limitada y agregar hasta tres personas como colaboradores del repositorio. Para obtener repositorios privados ilimitados con colaboradores ilimitados, puedes actualizar a GitHub Pro, GitHub Team o GitHub Enterprise Cloud. For more information, see "[GitHub's products](https://help.github.com/es/articles/github-s-products)."

Puedes colaborar en tu proyecto con otros usando los tableros de proyecto, las solicitudes de extracción y las propuestas de tu repositorio.



# Gestores De Paquetes

Yarn

Frente a npm, posiblemente el gestor de paquetes JavaScript con más tirón dentro de la comunidad de desarrolladores JS, ha surgido alguna alternativa que puede tener cierto recorrido si miramos quiénes son sus padrinos dentro del mercado: [Facebook](https://www.facebook.com/), [Google](https://www.google.es/), [Exponent](https://getexponent.com/" \t "_blank) y Tilde. La alternativa se llama [Yarn](https://github.com/yarnpkg/yarn" \t "_blank) y ha despertado mucha atención no sólo por sus creadores, sino también por el cambio de enfoque en la descarga e instalación de los paquetes y en su gestión de las dependencias.

Final del formulario

Facebook usó durante años el cliente npm para la instalación y gestión de paquetes JavaScript en su código, pero reconoce que con el aumento del volumen de su código y el crecimiento del número de ingenieros y programadores que participaban en ese código, “los problemas de consistencia, seguridad y rendimiento” también se disparaban en paralelo. Esa fue la razón por la que la compañía de Menlo Park decidió desarrollar su propio gestor de paquetes JavaScript, que permitiera “gestionar nuestras dependencias con mayor fiabilidad”.

Diferencias clave entre npm y Yarn

Entre npm y Yarn existen algunas diferencias importantes que son fruto del trabajo realizado por Facebook y sus socios en este gestor de paquetes JavaScript:

* Gestión de las dependencias: dentro del ecosistema de nodos en el que suele trabajar un gestor de paquetes JavaScript, las dependencias se sitúan en el *node\_modules*del directorio del proyecto. En el caso de npm, el orden de instalación de los paquetes y la generación de las dependencias define la estructura de ese nodo del directorio. Por tanto, cuando un desarrollador trabaja con npm, lo más seguro es que su estructura del nodo de dependencias sea distinta al nodo de dependencias de otros programadores.

Yarn no funciona así. Este gestor de paquetes JavaScript usa ciertos ficheros de bloqueo que actúan sobre las dependencias instaladas en determinadas versiones del código, de tal forma que garantiza que el *node\_modules*tenga una misma estructura de archivos independientemente del proyecto y los desarrolladores que hayan ido accediendo a la base de programación. Yarn genera cierta unidad en todo el proceso, aunque se produzcan cambios. De todas formas, npm también dispone del comando *[npm shrinkwrap](https://docs.npmjs.com/cli/shrinkwrap" \t "_blank)*, que permite bloquear determinadas versiones de los distintos paquetes y sus descendientes dentro del directorio jerárquico *node\_modules*. El problema viene cuando se producen cambios en paquetes que ya [no se encuentran dentro de lo que en npm se conoce como versionado semántico](http://javascript.tutorialhorizon.com/2015/03/21/what-is-npm-shrinkwrap-and-when-is-it-needed/) ([semver](http://semver.org/" \t "_blank)).

El proceso de instalación y control que introduce Yarn tiene distintos pasos:

- *Resolution*: Yarn resuelve las dependencias entre paquetes o librerías JavaScript haciendo solicitudes al registro y revisando cada dependencia que se encuentre ya gestionada dentro del directorio.

- *Fletching*: después, Yarn revisa el directorio global almacenado en la memoria caché y comprueba que el paquete o librería JavaScript que se quiere descargar no fue instalado con anterioridad. Si Yarn comprueba que no lo tiene, descarga el paquete y lo instala en la caché para evitar instalar y gestionar en el futuro la misma dependencia.

- *Linking*: el último paso de Yarn es copiar todos los archivos de la memoria caché al *node\_modules* del directorio local para que el desarrollador pueda empezar a trabajar con el paquete JavaScript.

* Tareas en la instalación de paquetes: cuando un desarrollador quiere instalar un nuevo paquete JavaScript, eso lanza una serie de tareas previas. En el caso de npm, la tareas son ejecutadas por cada paquete y de forma progresiva y secuencial. Eso quiere decir que primero ejecuta las tareas para la instalación del primer paquete, después del segundo y así sucesivamente. En el caso de Yarn, las tareas por paquetes se ejecutan en paralelo y eso tiene una incidencia evidente en los tiempos de rendimiento. [En esta comparativa realizada por el sitio web Sitepoint](https://www.sitepoint.com/yarn-vs-npm/) pueden verse algunas diferencias de rendimiento en la instalación de los paquetes JavaScript [express](https://www.npmjs.com/package/express" \t "_blank) y [gulp](https://www.npmjs.com/package/gulp" \t "_blank) (con el que se gestionaron un total de 195 dependencias).
* Línea de comandos: los comandos utilizados por Yarn y npm son distintos, desde los usados para la propia instalación del gestor hasta para la búsqueda de paquetes o la gestión de los archivos de bloqueo para las dependencias.

# NPM

Si en alguna ocasión has utilizado en otro lenguaje de programación un gestor de paquetes o dependencias como **composer** de *PHP* o **gem** de *Ruby* ya tendrás una idea de cómo funcionan. Si nunca has utilizado alguno de estos no te preocupes, comenzaremos explicando desde cero para que se utilizan y como hacer uso del mismo.

Npm es el gestor de paquetes de NodeJS. Según la definición que le dan sus creadores en su web oficial:

Para los desarrolladores Javascript npm hace que sea fácil compartir y reutilizar código, y también facilita la actualización del código que compartes.

Desde que salió node y los desarrolladores se dieron cuenda de la gran utilidad y su rendimiento, comenzaron a desarrollar paquetes para todo tipo de usos. Todos estos paquetes son compartidos a través de npm el que a su vez es el encargado de gestionar cada dependencia de los mismos. Si en algún momento necesitas alguna funcionalidad para tu aplicación, te recomiendo que antes de comenzar a desarrollarla por ti mismo, busques en el sitio de [npm](https://www.npmjs.com/) donde casi seguro encontrarás esa funcionalidad para tu aplicación.

**Actualizando npm**

Como ya sabes **npm** viene incluido en la instalación de NodeJS. Pero como npm se actualiza con más frecuencia que node debes asegurarte cuando instales node, actualizar npm a su última versión. Para obtener la última versión de npm lo único que necesitamos es ejecutar un comando en la consola de Windows.

npm install npm -g

El comando anterior actualizara tu versión de npm a la última estable. Para comprobarlo puedes ejecutar en la misma consola de Windows.

npm -v

La versión ahora debe ser mayor que la que venía con la instalación de node.

**Como usar el comando npm**

Como habrás podido observar en el apartado de actualización, el comando npm recibe varias instrucciones con las cuales ejecuta las acciones. Para ver la lista de las instrucciones que puedes ejecutar con él, puedes ejecutar en la consola lo siguiente:

npm -h

El comando anterior imprimirá la ayuda de npm con todos los posibles comandos a ejecutar. Además para saber sobre cómo utilizar un comando específico podemos pedir ayuda para cada comando por separado de la siguiente forma:

node parámetro -h

Debes remplazar **«parámetro»** por uno de los comandos de node en el ejemplo anterior.

# Node.js

Node.js es un entorno JavaScript de lado de servidor que utiliza un modelo asíncrono y dirigido por eventos.

Si ya has escuchado algo acerca de Node.js, supongo que lo primero que te preguntarás es: ¿Qué es Node.js?, y espero que este artículo aclare tus dudas.

Node.js es un entorno JavaScript de lado de servidor que utiliza un modelo asíncrono y dirigido por eventos.

Es una Máquina Virtual tremendamente rápida y de gran calidad escrita por gente como Lars Bak, uno de los mejores ingenieros del mundo especializados en VMs(Virtual Machines). No olvidemos que V8 es actualizado constantemente y es uno de los intérpretes más rápidos que puedan existir en la actualidad para cualquier lenguaje dinámico. Además las capacidades de Node.js para I/O (Entrada/Salida) son realmente ligeras y potentes, dando al desarrollador la posibilidad de utilizar a tope la I/O del sistema.

Node soporta protocolos TCP, DNS y HTTP.  Y fue creado por Ryan Dahl en 2009 y su evolución está apadrinada por la empresa Joyent que además tiene contratado a Dahl.

Uno de los puntos fuertes de Node.js es su capacidad de mantener muchas conexiones abiertas y esperando. En Apache por ejemplo el parámetro MaxClients por defecto es 256. Este valor puede ser aumentado para servir contenido estático, sin embargo si se sirven aplicaciones web dinámicas en PHP u otro lenguaje es probable que al poner un valor alto el servidor se quede bloqueado ante muchas conexiones esto dependerá del trabajo que la aplicación web de lado del servidor y de su capacidad hardware.

Después de haber leído todo esto, espero que ahora tengas mucho más interés por adentrarte al mundo de Node.js y de descubrir todo lo grandioso que tiene para ti.

# ECMASCRIPT (ES6)

**Función Arrow**

¿Cuántas veces has programado un código con una estructura similar a la siguiente?

1*// ES5*

2*// Imaginemos una variable data que incluye un array de objectos*

3var data = [{...}, {...}, {...}, ...];

4data.forEach(function(elem){

5 *// Tratamos el elemento*

6 console.log(elem)

7});

Con la función *arrow* => de ES6, el código anterior se sustituiría por:

1*//ES6*

2var data = [{...}, {...}, {...}, ...];

3data.forEach(elem => {

4 console.log(elem);

5});

Mucho más limpio y claro. CoffeeScript (un metalenguaje que compila a JavaScript) usa algo parecido.

Incluso la podemos utilizar así:

1*// ES5*

2var miFuncion = function(num) {

3 return num + num;

4}

1*// ES6*

2var miFuncion = (num) => num + num;

**Clases**

Ahora JavaScript tendrá clases, muy parecidas las funciones constructoras de objectos que realizabamos en el estándar anterior, pero ahora bajo el paradigma de clases, con todo lo que eso conlleva, como por ejemplo, herencia. Aunque no deja de ser un *azúcar sintáctico (Sugar Syntax)* porque en JavaScript no tenemos clases, tenemos prototipos.

1class LibroTecnico extends Libro {

2 constructor(tematica, paginas) {

3 super(tematica, paginas);

4 this.capitulos = [];

5 this.precio = "";

6 *// ...*

7 }

8 metodo() {

9 *// ...*

10 }

11}

**This**

La variable this muchas veces se vuelve un dolor de cabeza. antíguamente teníamos que cachearlo en otra variable ya que solo hace referencia al contexto en el que nos encontremos. Por ejemplo, en el siguiente código si no hacemos var that = this dentro de la función document.addEventListener, *this* haría referencia a la función que pasamos por *Callback* y no podríamos llamar a foo()

1*//ES3*

2var obj = {

3 foo : function() {...},

4 bar : function() {

5 var that = this;

6 document.addEventListener("click", function(e) {

7 that.foo();

8 });

9 }

10}

Con ECMAScript5 la cosa cambió un poco, y gracias al método bind podíamos indicarle que this hace referencia a un contexto y no a otro.

1*//ES5*

2var obj = {

3 foo : function() {...},

4 bar : function() {

5 document.addEventListener("click", function(e) {

6 this.foo();

7 }.bind(this));

8 }

9}

Ahora con ES6 y la función *Arrow* => la cosa es todavía más visual y sencilla.

1*//ES6*

2var obj = {

3 foo : function() {...},

4 bar : function() {

5 document.addEventListener("click", (e) => this.foo());

6 }

7}

**let y const**

Ahora podemos declarar variables con let en lugar de var si no queremos que sean accesibles más allá de un ámbito. Por ejemplo:

1*//ES5*

2(function() {

3 console.log(x); *// x no está definida aún.*

4 if(true) {

5 var x = "hola mundo";

6 }

7 console.log(x);

8 *// Imprime "hola mundo", porque "var" hace que sea global*

9 *// a la función;*

10})();

11

12*//ES6*

13(function() {

14 if(true) {

15 let x = "hola mundo";

16 }

17 console.log(x);

18 *//Da error, porque "x" ha sido definida dentro del "if"*

19})();

Ahora con const podemos crear constantes que sólo se puedan leer y no modificar a lo largo del código. Veamos un ejemplo

1(function() {

2 const PI;

3 PI = 3.15;

4 *// ERROR, porque ha de asignarse un valor en la*

5 *// declaración*

6})();

7

8(function() {

9 const PI = 3.15;

10 PI = 3.14159;

11 *// ERROR de nuevo, porque es sólo-lectura*

12})();

**Template Strings**

Con ES6 podemos interpolar *Strings* de una forma más sencilla que como estábamos haciendo hasta ahora. Fíjate en este ejemplo:

1*//ES6*

2let nombre1 = "JavaScript";

3let nombre2 = "awesome";

4console.log(`Sólo quiero decir que ${nombre1} is ${nombre2`);

5*// Solo quiero decir que JavaScript is awesome*

También podemos tener String multilínea sin necesidad de concatenarlos con +.

1*//ES5*

2var saludo = "ola " +

3"que " +

4"ase ";

5

6*//ES6*

7var saludo = "ola

8que

9ase";

10

11console.log("hola

12que

13ase");

**Destructuring**

Tenemos nuevas formas de asignar valores a Arrays y a Objetos. Veamos unos ejemplos

1var [a, b] = ["hola", "mundo"];

2console.log(a); *// "hola"*

3console.log(b); *// "mundo"*

4

5var obj = { nombre: "Carlos", apellido: "Azaustre" };

6var { nombre, apellido } = obj;

7console.log(nombre); *// "Carlos"*

¿No te ha estallado el cerebro todavía? Pues mira esto:

1var foo = function() {

2 return ["175", "75"];

3};

4var [estatura, peso] = foo();

5console.log(estatura); *//175*

6console.log(peso); *//75*

**Valores por defecto**

Otra novedad es asignar valores por defecto a las variables que se pasan por parámatros en las funciones. Antes teníamos que comprobar si la variable ya tenía un valor. Ahora con ES6 se la podemos asignar según creemos la función.

1*//ES5*

2function(valor) {

3 valor = valor || "foo";

4}

5

6*//ES6*

7function(valor = "foo") {...};

**Módulos**

A esto lo llamo un browserify nativo. Ahora JavaScript se empieza a parecer a lenguajes como *Python* o *Ruby*. Llamamos a las funciones desde los propios Scripts, sin tener que importarlos en el HTML, si usamos JavaScript en el navegador.

1*//File: lib/person.js*

2module "person" {

3 export function hello(nombre) {

4 return nombre;

5 }

6}

*También se puede exportar así, según una aclaración de*[*Sergio Daniel Xalambrí*](https://twitter.com/sergiodxa)

*export function hello(nombre) {...}; o también, si solo es una función la que tiene el módulo: export default function(nombre) {...};*

Y para importar en otro fichero:

1*//File: app.js*

2import { hello } from "person";

3var app = {

4 foo: function() {

5 hello("Carlos");

6 }

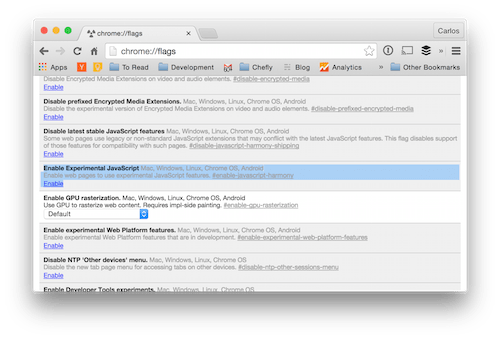
7}

8export app;

**Cómo empezar a usar ES6 hoy mismo**

Todo esto es muy bonito, pero ¿Cómo podemos empezar a utilizarlo hoy en día?. Lo primero más recomendable es que te instales la última versión de Chrome, que es uno de los navegadores que está implementando las nuevas *features* de ES6 más rápidamente. Te aconsejo incluso que instales [**Chrome Canary**](https://www.google.com/chrome/browser/canary.html), que es la versión de Chrome que prueba funcionalidades antes de lanzarlas en el Chrome original.

Para probar directamente código ES6 en la consola de tu navegador. Escribe en la barra de direcciones chrome://flags y tendrás una página como ésta:



Y activar el flag **Enable Experimental JavaScript**.

Esto te permitirá probar algunas *features* pero no todas porque algunas aún están en desarrollo.

La otra opción, más extendida, es escribir en ES6 en tus ficheros JavaScript y después "compilarlos" a la versión ES5. Existen herramientas como [babel](https://babeljs.io/) que te permiten lograrlo.

Por medio de [plugings y paquetes](https://babeljs.io/docs/en/plugins) podemos utilizar estas *features* incluso las que estean en desarrollo para próximas versiones si [elegimos el paquete correspondiente](https://babeljs.io/setup).

Desde 2015, cada año se introducen nuevas características en JavaScript. ES6 fue el cambio más radical y es normal encontrar referencias a esta versión como ES6 o ES2015, las siguientes versiones al principio se les llamaba ES7 y ES8, pero se ha optado por indicarlas con el año en el que son lanzados.

Por eso, hasta el momento podemos encontrar ES2015, ES2016, ES2017 y el más reciente ES2018, aunque como digo, desde la versión de 2015 los cambios son menores.

# Babel (Compilador De JavaScript)

ES6 funciona en todos los navegadores modernos, pero NO en en los más antiguos. He aquí una tabla detallada del [soporte en los navegadores de ES6](https://kangax.github.io/compat-table/es6/)C:\Users\Ferna\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\1A985008.tmp

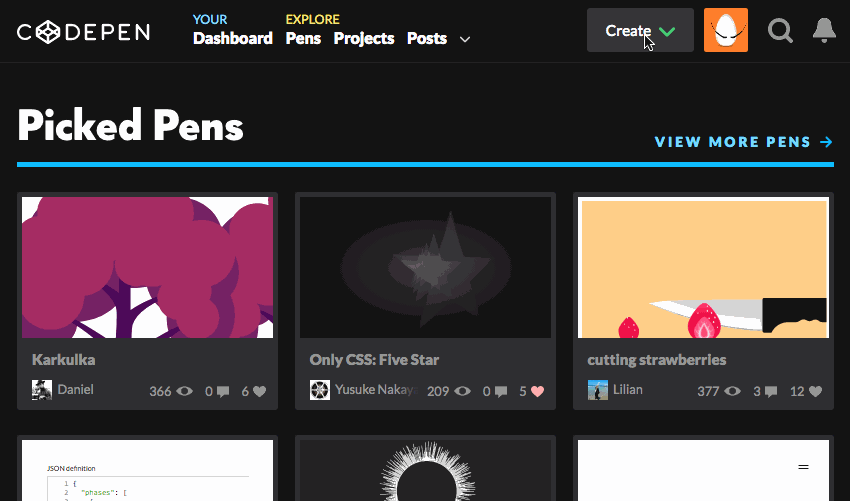
Si queremos que ES6 funcione también en los navegadores antiguos tenemos que utilizar polyfills, fragmentos de código que implementan las nuevas características de ES6 en estos navegadores.

**Donde encontrar los polyfills necesarios**

Si solo queremos implementar algunos nuevos métodos de ES6, por ejemplo el método codePointAt() de string, podemos encontrar el polyfill necesario a [MDN: String.prototype.codePointAt()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/codePointAt)C:\Users\Ferna\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\85291B56.tmp

**Babel - un compilador**

Una opción mejor es utilizar un compilador como por ejemplo [Babel](https://babeljs.io/)C:\Users\Ferna\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\62DF59D4.tmp, que "traduce" el código ES6 a ES5. Para entender que puede hacer Babel recomiendo utilizar [Codepen](https://codepen.io/" \t "_blank)C:\Users\Ferna\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\3C33E302.tmp donde podemos:  
crear un nuevo pen,  
escoger en Pen Settings "Babel" como JavaScript Preprocessor  
escribir algo de código en ES6  
y escoger la opción View Compiled JS que devuelve el código "traducido" a ES5.



También la página de [Babel](https://babeljs.io/)C:\Users\Ferna\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\41F4F6E1.tmp permite probar el ["traductor"](https://babeljs.io/repl/)C:\Users\Ferna\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\CBC13C97.tmp de ES6.

Todo esto está muy bien como demostración, pero si tenemos que escribir mucho código, no hay nada mejor que instalar Babel en el propio ordenador.

Para poder instalar Babel tenemos que instalar primero [Node](https://nodejs.org/es/" \t "_blank)C:\Users\Ferna\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\C678A9BD.tmp. Esta es una tarea bastante sencilla aunque a veces podemos tener problemas. Si todavía no lo tiene instalado, hay muchísimos [tutoriales](https://desarrolloweb.com/articulos/instalar-node-js.html)C:\Users\Ferna\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\7F195FD3.tmp que explican cómo hacerlo.

Después de instalar Node podemos instalar [Babel](https://babeljs.io/)C:\Users\Ferna\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\F1717C59.tmp

**Cómo instalar Babel – por pasos**

Babel aconseja instalar el compilador localmente, proyecto por proyecto. Para instalar Babel podemosseguir los siguientes pasos:

**1**. Creamos un directorio para el nuevo proyecto ( por ejemplo BabelTest )  
**2**. Dentro del nuevo directorio creamos un archivo .js ( por ejemplo script.js ).  
Es aquí es donde habrá que escribir el código ES6.  
**3**. Abrimos el terminal ( *command line*  o CLI )  
**4**. Cambiamos de directorio utilizando el comando cd.

cd /Users/. . . . . /BabelTest

Un pequeño truco: después de escribir el comando cd ( *change directory* ) podemos arrastrar el directorio directamente en el terminal para recuperar la dirección de este.

**5**. Iniciamos el npm ( *node package manager* o el gestor de paquetes de node ).

npm init

Esto genera un nuevo package.json en el directorio BabelTest. El proceso dura unos cuantos segundos, tiempo en el cual tenemos que contestar algunas preguntas o simplemente dar a enter para escoger los valores por defecto.

El aspecto del nuevo package.json generado es algo así:

{

  "name": "babeltest",

  "version": "1.0.0",

  "description": "",

  "main": "script.js",

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1"

  },

  "author": "",

  "license": "ISC"

}

**6**. Ahora podemos abrir este archivo y modificarlo:

{

  "name": " babeltest ",

  "version": "1.0.0",

  "description": "",

  "main": "script.js",

  "scripts": {

    "babel": "babel script.js --watch --out-file script-compiled.js"

  },

  "author": "",

  "license": "ISC"

}

En este caso queremos escribir el código ES6 en el script.js, y queremos que el código compilado aparezca en el script-compiled.js. O sea: script-compiled.js es el nombre del fichero de salida.

También necesitamos utilizar algunos indicadores ( *flags* ):  
--watch: utilizado para compilar el script.js cada vez que este cambia.  
--out-file el javascript compilado aparece en un fichero de salida: script-compiled.js en este caso.

Pero estos no son los únicos indicadores que podemos utilizar. [Puede leer más aquí](https://babeljs.io/docs/usage/cli/#babel-compile-files)C:\Users\Ferna\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\5F2C78CF.tmp

A continuación podemos instalar Babel. Para esto escribimos el siguiente comando en el terminal:

npm install --save-dev babel-cli

Después de darle a enter la instalación empieza y dura unos cuantos segundos. Cuando la instalación acaba, el directorio BabelTest tiene la siguiente estructura:  
- Una carpeta node\_modules y tres ficheros:  
- package-lock.json  
- package.json  
- script.js

El package.json también ha cambiado:

{

  "name": " babeltest ",

  "version": "1.0.0",

  "description": "",

  "main": "script.js",

  "scripts": {

    "babel": "babel script.js --watch --out-file script-compiled.js"

  },

  "author": "",

  "license": "ISC",

  "devDependencies": {

   "babel-cli": "^6.26.0"

 }

}

En este caso la versión de babel instalada es 6.26.0, pero esto cambiará en unas cuantas semanas.

**8**. A continuación tenemos que decidir que necesitamos: o sea los [plugins](https://babeljs.io/docs/plugins/" \t "_blank)C:\Users\Ferna\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\D6278AB5.tmp necesarios para transformar el código ES6 en ES5. Pero hay una opción mejor: utilizar presets: o sea paquetes de plugins, por ejemplo [es2015](https://babeljs.io/docs/plugins/preset-es2015/)C:\Users\Ferna\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\30A5038B.tmp ( este es un clásico ).

Un preset todavía mejor es el [env preset](https://babeljs.io/docs/plugins/preset-env/" \t "_blank)C:\Users\Ferna\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\ADB6B0D1.tmp que detecta automáticamente los plugins necesarios.

Para instalar este preset tenemos que modificar de nuevo el package.json de esta manera:

{

  "name": " babeltest ",

  "version": "1.0.0",

  "description": "",

  "main": "script.js",

  "scripts": {

    "babel": "babel script.js --watch --out-file script-compiled.js"

  },

  "author": "",

  "license": "ISC",

  "devDependencies": {

    "babel-cli": "^6.26.0"

  },

  "babel": {

 "presets": [

    ["env", {

      "targets": {

        "browsers": ["last 2 versions", "safari >= 7"]

      }

    }]

  ]

}

}

Puede encontrar este ejemplo aquí: [Preset env](https://babeljs.io/docs/plugins/preset-env/" \t "_blank)C:\Users\Ferna\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\C9F73C07.tmp

El código que acabamos de añadir al package.json dice que queremos utilizar el preset env, y que el JavaScript compilado funcione en las ultimas dos versiones de cada navegador ( *"last 2 versions"* ), y en las versiones de Safari >= 7.

Ahora podemos volver al terminal para instalar el preset env:

npm install babel-preset-env --save-dev

La instalación dura unos cuantos segundos.

**9**. Si todo ha ido bien, podemos ejecutar Babel escribiendo el siguiente comando en el terminal:

npm run babel

Esto crea un nuevo fichero js: script-compiled.js en este caso. Ahora al modificar el script.js, Babel lo compila automáticamente en el script-compiled.js

