

Manual de Angular



Alberto Basalo
Miguel Angel Alvarez

Introducción: Manual de Angular

Comenzamos la redacción a nuestro Manual de Angular, la evolución del framework Javascript más popular para el desarrollo de aplicaciones del lado del cliente.

En esta página irás viendo los artículos del Manual de Angular según los vayamos publicando a lo largo de los próximos meses.

Nuestro objetivo es recorrer las piezas principales de este potente framework y estudiar los mecanismos para el desarrollo de aplicaciones web universales con Angular 2. De momento encontrarás una estupenda introducción al desarrollo con el nuevo Angular, y abordaremos muchos más detalles en breve.

Este manual aborda el framework Angular, en sus versiones 2 en adelante. Cabe aclarar que, poco después de salir la versión de Angular 2, el framework cambió de nombre a Angular. A partir de aquí se han ido publicando diversas entregas con números basados en el versionado semántico, pero siempre bajo la misma base tecnológica. Por tanto, este manual es válido tanto si quieres conocer Angular 2 o Angular 4 o las que vengan, ya que el framework en sí sigue manteniendo sus mismas bases.

Encuentras este manual online en:

<http://desarrolloweb.com/manuales/manual-angular-2.html>

Autores del manual

Las siguientes personas han participado como autores escribiendo artículos de este manual.

Miguel Angel Alvarez

Miguel es fundador de DesarrolloWeb.com y la plataforma de formación online EscuelaT. Comenzó en el mundo del desarrollo web en el año 1997, transformando su hobby en su trabajo.



Alberto Basalo

Alberto Basalo es experto en Angular y otras tecnologías basadas en Javascript, como NodeJS y MongoDB. Es director de Ágora Binaria, empresa dedicada al desarrollo de aplicaciones y a la formación a través de Academia Binaria.



Introducción al desarrollo con Angular

En esta primera parte del manual vamos a dar los primeros pasos para desarrollar aplicaciones con Angular. Veremos cómo crear el esqueleto de la aplicación básica y conoceremos los principales componentes de todo proyecto.

Introducción a Angular

Qué es Angular. Por qué se ha decidido escribir desde cero el popular framework Javascript y qué necesidades y carencias resuelve con respecto a su antecesor.

Angular 2 ha cambiado tanto que hasta el nombre es distinto. Lo conocíamos como "AngularJS" y ahora es sólo "Angular". No deja de ser una anécdota que hayan eliminado el "JS" hasta del nombre del dominio, pero es representativo. No porque ahora Angular no sea Javascript, sino porque es evolución radical.

Angular 2 es otro framework, no simplemente una nueva versión. A los que no conocían Angular 1 ésto les será indiferente, pero los que ya dominaban este framework sí deben entender que el conocimiento que necesitan adquirir es poco menos que si comenzasen desde cero. Obviamente, cuanto más experiencia en el desarrollo se tenga, más sencillo será lanzarse a usar Angular 2 porque muchas cosas sonarán de antes.

En este artículo encontrarás un poco de historia relacionada con Angular 1 y 2, junto con los motivos por los que Angular 2 es otro framework totalmente nuevo, que rompe compatibilidad hacia atrás.

Nota: Cabe decir que, aunque en este artículo nos refiramos constantemente a Angular como Angular 2, lo correcto es nombrarlo simplemente "Angular". De hecho, la versión 4 o la próxima versión 5 sigue llamándose "Angular", a secas. En estos momentos Angular se ha acogido al versionado semántico, por lo que el número de la versión importa menos, ya que siempre se mantiene el nombre del framework como "Angular" y la base tecnológica sigue siendo la misma. Lo que aprendas en este manual aplica a Angular 2, 4, 5... y en adelante.



Retos y necesidades de la nueva versión de Angular

Desde su creación hace ya más de 4 años, Angular ha sido el framework preferido por la mayoría de los desarrolladores Javascript. Este éxito ha provocado que los desarrolladores quieran usar el framework para más y más cosas.

De ser una plataforma para la creación de Web Apps, ha evolucionado como motor de una enorme cantidad de proyectos del ámbito empresarial y de ahí para aplicaciones en la Web Mobile Híbrida, llevando la tecnología al límite de sus posibilidades.

Es el motivo por el que comenzaron a detectarse problemas en Angular 1, o necesidades donde no se alcanzaba una solución a la altura de lo deseable. Son las siguientes.

Javascript: Para comenzar encontramos problemas en la creación de aplicaciones debido al propio Javascript. Es un lenguaje con carácter dinámico, asíncrono y de complicada depuración. Al ser tan particular resulta difícil adaptarse a él, sobre todo para personas que están acostumbradas a manejar lenguajes más tradicionales como Java o C#, porque muchas cosas que serían básicas en esos lenguajes no funcionan igualmente en Javascript.

Desarrollo del lado del cliente: Ya sabemos que con Angular te llevas al navegador mucha programación que antes estaba del lado del servidor, comenzando por el renderizado de las vistas. Esto hace que surjan nuevos problemas y desafíos. Uno de ellos es la sobrecarga en el navegador, haciendo que algunas aplicaciones sean lentas usando Angular 1 como motor.

Por otra parte tenemos un impacto negativo en la primera visita, ya que se tiene que descargar todo el código de la aplicación (todas las páginas, todas las vistas, todas las rutas, componentes, etc), que puede llegar a tener un peso de megas.

Nota: A partir de la segunda visita no es un problema, porque ya están descargados los scripts y cacheados en el navegador, pero para un visitante ocasional sí que representa un inconveniente grande porque nota que la aplicación tarda en cargar inicialmente.

Los intentos de implementar Lazy Load, o carga perezosa, en el framework en su versión 1.x no fueron muy fructíferos. Lo ideal sería que no fuese necesario cargar toda tu aplicación desde el primer instante, pero es algo muy difícil de conseguir en la versión precedente por el propio inyector de dependencias de Angular 1.x.

Otro de los problemas tradicionales de Angular era el impacto negativo en el SEO, producido por un renderizado en el lado del cliente. El contenido se inyecta mediante Javascript y aunque se dice que Google ha empezado a tener en cuenta ese tipo de contenido, las posibilidades de posicionamiento de aplicaciones Angular 1 eran mucho menores. Nuevamente, debido a la tecnología de Angular 1, era difícil de salvar.

Soluciones implementadas en el nuevo Angular 2

Todos esos problemas, difíciles de solucionar con la tecnología usada por Angular 1, han sido los que han impulsado a sus creadores a desarrollar desde cero una nueva versión del framework. La nueva herramienta está pensada para dar cabida a todos los usos dados por los desarrolladores, llevar a Javascript a un nuevo nivel comparable a lenguajes más tradicionales, siendo además capaz de resolver de una manera adecuada

las necesidades y problemas de la programación del lado del cliente.

En la siguiente imagen puedes ver algunas de las soluciones aportadas en Angular 2.



TypeScript / Javascript: Como base hemos puesto a Javascript, ya que es el inicio de los problemas de escalabilidad del código. Ayuda poco a detectar errores y además produce con facilidad situaciones poco deseables.

Nota: Con ECMAscript 6 ya mejora bastante el lenguaje, facilitando la legibilidad del código y solucionando diversos problemas, pero todavía se le exige más. Ya puestos a no usar el Javascript que entienden los navegadores (ECMAscript 5), insertando la necesidad de usar un transpilador como [Babel](#), podemos subir todavía un poco de nivel y usar TypeScript.

Angular 2 promueve el uso de TypeScript a sus desarrolladores. El propio framework está desarrollado en TypeScript, un lenguaje que agrega las posibilidades de ES6 y el futuro ES7, además de un tipado estático y ayudas durante la escritura del código, el refactoring, etc. pero sin alejarte del propio Javascript (ya que el código de Javascript es código perfectamente válido en TypeScript).

La sugerencia de usar TypeScript para desarrollar en Angular es casi una imposición porque la documentación y los generadores de código están pensados en TypeScript. Se supone que en futuro también estarán disponibles para Javascript, pero de momento no es así. De todos modos, para la tranquilidad de muchos, TypeScript no agrega más necesidad de procesamiento a las aplicaciones con Angular 2, ya que este lenguaje solamente lo utilizas en la etapa de desarrollo y todo el código que se ejecuta en el navegador es al final Javascript, ya que existe una transpilación previa.

Nota: Puedes saber más sobre este superset de Javascript en el artículo de [introducción a TypeScript](#).

Lazy SPA: Ahora el inyector de dependencias de Angular no necesita que estén en memoria todas las clases o código de todos los elementos que conforman una aplicación. En resumen, ahora con Lazy SPA el framework puede funcionar sin conocer todo el código de la aplicación, ofreciendo la posibilidad de cargar más adelante aquellas piezas que no necesitan todavía.

Renderizado Universal: Angular nació para hacer web y renderizar en HTML en el navegador, pero ahora el renderizado universal nos permite que no solo se pueda renderizar una vista a HTML. Gracias a ésto, alguien podría programar una aplicación y que el renderizado se haga, por ejemplo, en otro lenguaje nativo para un dispositivo dado.

Otra cosa que permite el renderizado universal es que se use el motor de renderizado de Angular del lado del servidor. Es una de las novedades más interesantes, ya que ahora podrás usar el framework para renderizar vistas del lado del servidor, permitiendo un mejor potencial de posicionamiento en buscadores de los contenidos de una aplicación. Esta misma novedad también permite reducir el impacto de la primera visita, ya que podrás tener vistas "precocinadas" en el servidor, que puedes enviar directamente al cliente.

Data Binding Flow: Uno de los motivos del éxito de Angular 1 fue el data binding, pero éste tenía un coste en tiempo de procesamiento en el navegador, que si bien no penalizaba el rendimiento en todas las aplicaciones sí era un problema en aquellas más complejas. El flujo de datos ahora está mucho más controlado y el desarrollador puede direccionarlo fácilmente, permitiendo optimizar las aplicaciones. El resultado es que en Angular 2 las aplicaciones pueden llegar a ser hasta 5 veces más rápidas.

Componentes: La arquitectura de una aplicación Angular ahora se realiza mediante componentes. En este caso no se trata de una novedad de la versión 2, ya que en la versión de Angular 1.5 ya se introdujo el [desarrollo basado en componentes](#).

Sin embargo, la componetización no es algo opcional como en Angular 1.5, sino es una obligatoriedad. Los componentes son estancos, no se comunican con el padre a no ser que se haga explícitamente por medio de los mecanismos disponibles, etc. Todo esto genera aplicaciones más mantenibles, donde se encapsula mejor la funcionalidad y cuyo funcionamiento es más previsible. Ahora se evita el acceso universal a cualquier cosa desde cualquier parte del código, vía herencia o cosas como el "Root Scope", que permitía en versiones tempranas de Angular modificar cualquier cosa de la aplicación desde cualquier sitio.

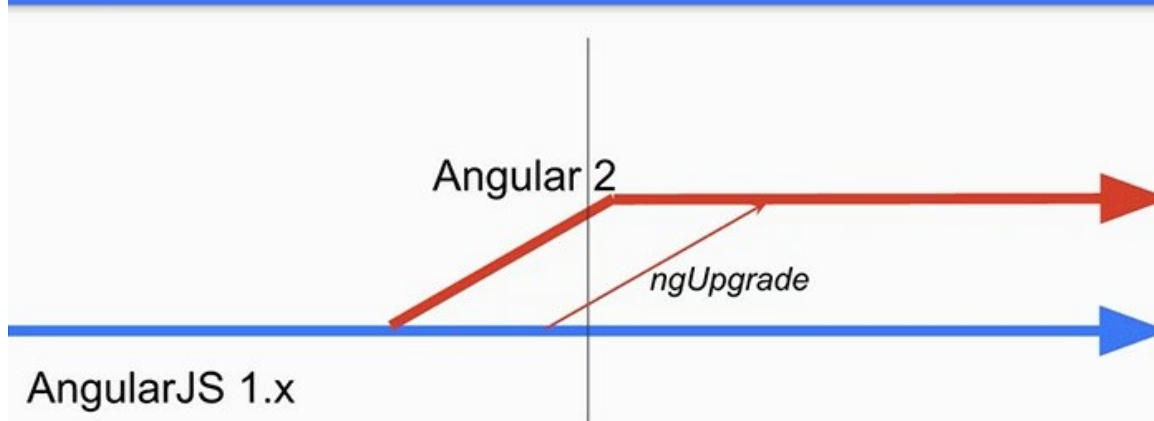
Evolución de las versiones 1 y 2 de Angular

Las versiones de AngularJS 1.x siguen vivas y continuarán dando soporte desde el equipo de Angular. Por tanto, se prevé que después de la actual 1.5 seguirán lanzando actualizaciones, nuevas versiones, etc.

La novedad es que ahora comienza en paralelo la vida de Angular 2 y seguirá evolucionando por su camino. Obviamente, la primera noticia que esperamos todos es que se presente la versión definitiva (ten en cuenta que en el momento de escribir estas líneas Angular 2 está solo en versiones RC, Release Candidate).

De este modo, debe quedar claro, para las personas que tengan aplicaciones en Angular 1.x, que no necesitan actualizarlas en una fecha determinada. Todavía existe un buen tiempo por delante en el que sus proyectos van a estar perfectamente soportados y el código de las aplicaciones perfectamente válido.

Evolución



Han anunciado además que en algún momento habrá algún sistema para hacer el upgrade de las aplicaciones de Angular 1.x a la 2.x. Esto podría permitir Incluso una convivencia, en una misma aplicación, de partes desarrolladas con Angular 1 y otras con la versión 2. La fecha de lanzamiento de ese hipotético "ngUpgrade" está sin confirmar y obviamente tampoco será magia. Veremos en el próximo próximo año lo que sucede, una vez que esa evolución de las versiones 1 y 2 estén conviviendo.

Conclusión

Se espera un futuro muy prometedor a Angular 2. Sus novedades son importantes y permitirán afrontar el futuro sobre una base tecnológica capaz de resolver todas las necesidades y retos actuales. En el Manual de Angular estaremos explicando en los próximos meses cómo usar este framework para desarrollar todo tipo de aplicaciones basadas en Javascript. Si además quieres aprender ya mismo Angular 2, tutorizado por nosotros te recomendamos también el [curso completo de Angular 2 que encuentras en Escuela T.](#)

Para aquel desarrollador que empieza desde cero en Angular 2 será un bonito camino que le permitirá crecer profesionalmente y ampliar seriamente sus capacidades y el rango de proyectos que sea capaz de acometer. El recorrido puede ser difícil al principio, pero la recompensa será grande.

Por su parte, quien ya tenga experiencia en Angular 1.x siempre le será positiva, sobre todo para los que (a partir de la 1.5) comenzaron a usar componentes. Aunque en Angular 2 cambie la manera de realizar las cosas, le resultará todo más familiar y por tanto su curva de aprendizaje será más sencilla.

Este artículo es obra de Alberto Basalo
Fue publicado por primera vez en 09/ 06/ 2016
Disponible online en <http://desarrolloweb.com/articulos/introduccion-angular2.html>

Angular CLI

Qué es Angular CLI, el intérprete de línea de comandos de Angular 2 que te facilitará el inicio de proyectos y la creación del esqueleto, o scaffolding, de la mayoría de los componentes de una aplicación Angular.

Después de la [introducción a las características de Angular 2](#) del pasado artículo, en esta ocasión te presentamos una introducción Angular CLI, una de las herramientas esenciales para desarrollar con el nuevo framework Angular. Es un paso esencial y necesario antes de comenzar a ver código, puesto que necesitamos esta herramienta para poder iniciar nuestra primera aplicación Angular 2, 4 y versiones venideras.

Debido a la complejidad del desarrollo con Angular, aunque el ejemplo que deseemos desarrollar se trate de un sencillo "Hola mundo", comenzar usando Angular CLI nos ahorrará escribir mucho código y nos permitirá partir de un esquema de aplicación avanzado y capaz de facilitar los flujos de desarrollo. Además nos ofrecerá una serie de herramientas ya configuradas y listas para hacer tareas como, depuración, testing o deploy. Y dicho sea de paso, el CLI también nos ahorrará caer en errores de principiante que nos provoquen frustración en los momentos iniciales. Así que vamos con ello.

Nota: Mencionamos la complejidad del desarrollo con Angular y la necesidad de usar su CLI, pero es que el propio desarrollo para la web se ha sofisticado bastante. Necesitamos compilar o transpilar el código, gestores de dependencias, sistemas de empaquetado y compactado de código, sistemas que nos revisen la sintaxis y nos ayuden a escribir un código limpio, etc. Cada una de esas tareas las debes de realizar con una herramienta definida, pero configurarlas todas y conseguir que trabajen al unísono no es una tarea trivial. Aparte de tiempo necesitarás bastante experiencia y es algo que no siempre se dispone. Por ello, la mayoría de los frameworks actuales ofrecen interfaces por línea de comandos para hacer las más diversas tareas. Por todo ello, no hay un framework o librería que se precie que no ofrezca su propio CLI.



Qué es Angular CLI

Dentro del ecosistema de Angular 2 encontramos una herramienta fundamental llamada "Angular CLI" (Command Line Interface). Es un producto que en el momento de escribir este artículo todavía se encuentra en fase beta, pero que ya resulta fundamental para el trabajo con el framework.

Angular CLI no es una herramienta de terceros, sino que nos la ofrece el propio equipo de Angular. En resumen, nos facilita mucho el proceso de inicio de cualquier aplicación con Angular, ya que en pocos minutos te ofrece el esqueleto de archivos y carpetas que vas a necesitar, junto con una cantidad de herramientas ya configuradas. Además, durante la etapa de desarrollo nos ofrecerá muchas ayudas, generando el "scaffolding" de muchos de los componentes de una aplicación. Durante la etapa de producción o testing también nos ayudará, permitiendo preparar los archivos que deben ser subidos al servidor, transpilar las fuentes, etc.

Node y npm

Angular CLI es una herramienta NodeJS, es decir, para poder instalarla necesitaremos contar con NodeJS instalado en nuestro sistema operativo, algo que podemos conseguir muy fácilmente yendo a la página de <https://nodejs.org> y descargando el instalador para nuestro sistema.

Además se instala vía "npm". Por npm generalmente no te tienes que preocupar, pues se instala al instalar NodeJS. No obstante es importante que ambas versiones, tanto la de la plataforma Node como el gestor de paquetes npm, se encuentren convenientemente actualizados. En estos momentos como requisito nos piden tener Node 4 o superior.

Actualizado: En estos momentos, septiembre de 2017 y para Angular en su versión 4, es necesario tener al menos NodeJS versión 6.9.x y npm 3.x.x.

Nota: Puedes saber la versión de Node que tienes instalada, así como la versión de npm por medio de los comandos:

```
node -v
```

```
npm -v
```

No tienes que saber Node para desarrollar con Angular, pero sí necesitas tenerlo para poder instalar y usar Angular Angular CLI, además de una serie de herramientas fantásticas para desarrolladores.

Instalar Angular CLI

Esto lo conseguimos desde el terminal, lanzando el comando:

```
npm install -g @angular/cli
```

Durante el proceso de instalación se instalará el propio Angular CLI junto con todas sus dependencias. La instalación puede tardar varios minutos dependiendo de la velocidad de tu conexión a Internet.

Una vez instalado dispondrás del comando "ng" a partir del cual lanzarás cualquiera de las acciones que se pueden hacer mediante la interfaz de comandos de Angular. Puedes comenzar lanzando el comando de ayuda:

```
ng --help
```

Nota: ng (que se lee "enji") es el apelativo familiar de "Angular" que se conoce desde el inicio del framework.

También encontrarás una excelente ayuda si entras en la [página de Angular CLI](#), navegando por sus secciones, o bien en el propio [repositorio de GitHub angular-cli](#).

Crear el esqueleto de una aplicación Angular 2

Uno de los comandos que puedes lanzar con Angular CLI es el de creación de un nuevo proyecto Angular 2. Este comando se ejecuta mediante "new", seguido del nombre del proyecto que queremos crear.

```
ng new mi-nuevo-proyecto-angular
```

Lanzado este comando se creará una carpeta igual que el nombre del proyecto indicado y dentro de ella se generarán una serie de subcarpetas y archivos que quizás por su número despisten a un desarrollador que se inicia en Angular. Si es así no te preocupes porque poco a poco nos iremos familiarizando con el código generado.

Además, como hemos dicho, se instalarán y se configurarán en el proyecto una gran cantidad de herramientas útiles para la etapa del desarrollo front-end. De hecho, gran cantidad de los directorios y archivos generados al crear un nuevo proyecto son necesarios para que estas herramientas funcionen. Entre otras cosas tendremos:

- Un servidor para servir el proyecto por HTTP
- Un sistema de live-reload, para que cuando cambiamos archivos de la aplicación se refresque el navegador
- Herramientas para testing
- Herramientas para despliegue del proyecto
- Etc.

Una vez creado el proyecto inicial podemos entrar en la carpeta con el comando `cd`.

```
cd mi-nuevo-proyecto-angular
```

Una vez dentro de esa carpeta encontrarás un listado de archivos y carpetas similar a este:

```
▸ e2e
▸ node_modules
▸ src
{} .angular-cli.json
⚙ .editorconfig
📄 .gitignore
📄 karma.conf.js
{} package-lock.json
{} package.json
📄 protractor.conf.js
📄 README.md
{} tsconfig.json
{} tslint.json
```

Nota: Esta lista de carpetas y archivos se ha actualizado a la versión de Angular 4, en septiembre de 2017. Con nuevas versiones del framework nos podemos encontrar con ligeras diferencias del contenido de la aplicación básica generada. En principio no debería preocuparnos demasiado, puesto que a veces se reorganizan las cosas o se usan dependencias distintas que no alteran mucho el funcionamiento o las prácticas necesarias para el desarrollo con Angular.

Servir el proyecto desde un web server

Angular CLI lleva integrado un servidor web, lo que quiere decir que podemos visualizar y usar el proyecto sin necesidad de cualquier otro software. Para servir la aplicación lanzamos el comando "serve".

```
ng serve
```

Eso lanzará el servidor web y lo pondrá en marcha. Además, en el terminal verás como salida del comando la ruta donde el servidor está funcionando. Generalmente será algo como esto (pero te sugerimos verificar el puerto en la salida de tu terminal):

```
http://localhost:4200/
```

En la siguiente imagen ves la salida del terminal nuestro.

```
mcMiguel:test-angular2 midesweb$ ng serve
Could not start watchman; falling back to NodeWatcher for file system events.
Visit http://ember-cli.com/user-guide/#watchman for more info.
Livereload server on http://localhost:49152
Serving on http://localhost:4200/
```

```
Build successful - 831ms.
```

Slowest Trees	Total
-----+-----	
BroccoliTypeScriptCompiler	405ms
vendor	337ms
Slowest Trees (cumulative)	Total (avg)
-----+-----	
BroccoliTypeScriptCompiler (1)	405ms
vendor (1)	337ms

Podrías modificar el puerto perfectamente si lo deseas, simplemente indicando el puerto deseado con la opción `--port`:

```
ng serve --port 4201
```

Una vez hayas abierto en navegador y accedido a la URL de localhost, con el puerto indicado, verás la pantalla de inicio de la aplicación que se acaba de crear. Tendrá un aspecto similar al de la siguiente imagen (ten en cuenta que esta página es solo una especie de "hola mundo" y que su aspecto puede cambiar dependiendo de la versión de Angular que estés usando).



Welcome to app!



Here are some links to help you start:

- [Tour of Heroes](#)
- [CLI Documentation](#)
- [Angular blog](#)

Scripts de npm

Para quien no lo sepa, npm es el "Node Package Manager", el gestor de paquetes de NodeJS. El CLI de Angular lo hemos instalado vía npm. Nuestra propia aplicación Angular se puede gestionar vía npm, para instalar sus dependencias. Iremos conociendo las mecánicas en el [Manual de Angular](#), no obstante, queremos detenernos a hablar de una herramienta de utilidad a la que puedes sacar ya mismo algo de valor: los scripts de npm.

Los scripts de npm son como subcomandos que npm te acepta para automatizar diferentes tareas. Éstos se definen en el archivo "package.json", en la sección "scripts".

Nota: "package.json" es una especie de resumen de las cosas que se han instalado en un proyecto vía npm. Nuestra aplicación Angular recién instalada tiene un archivo package.json que puedes abrir para dar un primer vistazo. En un archivo package.json encontramos diversas cosas, como el nombre del proyecto, la licencia, autores, las dependencias de un proyecto, tanto las que importan para la fase de desarrollo como para el proyecto en general. Este archivo tiene formato JSON, por lo que es un simple texto plano que no te costará de leer.

Como scripts de npm en nuestro proyecto recién creado encuentras varias alternativas como "start", "build", "test", etc. Esos scripts se pueden poner en marcha desde la consola, mediante el comando "npm", por ejemplo, para lanzar el script "start", tienes que escribir el comando de consola:

```
npm start
```

En el archivo package.json encuentras el comando que se ejecutará al hacer cada uno de estos scripts npm. Por ejemplo "start" lo que provoca es que se lance el comando "ng serve". Por lo tanto los comandos "npm start" o "ng serve" en principio son equivalentes. Sin embargo, nosotros podemos modificar los scripts npm para ajustarlos a nuestras necesidades, o incluso crear nuevos comandos npm. Para probar simplemente vamos a poner "--o" al final del comando correspondiente con el script "start". Este "--o" lo que hace es que, al iniciarse el servidor para correr la aplicación Angular, se abra el navegador directamente y nos la muestre.

```
"scripts": {  
  "ng": "ng",  
  "start": "ng serve --o",  
  [...]  
}
```

Ahora, al hacer un "npm start" observarás cómo se inicia el servidor web, nos indica la URL donde está disponible, pero nos abre el navegador automáticamente mostrando esa URL. Así te ahorras el trabajo de lanzar tú mismo el browser y escribir a mano la dirección del servidor que se ha abierto para servir nuestra app de Angular.

Problema Angular CLI con Broccoli en Windows

El problema más típico que nos podemos encontrar al usar Angular CLI es que no tengamos permisos de administrador. Al intentar poner en marcha el servidor recibirás un error como este:

```
TheBroccoli Plugin: [BroccoliTypeScriptCompiler] failed with: operation not permitted.
```

Es muy sencillo de solucionar en Windows, ya que simplemente necesitamos abrir el terminal en modo administrador (botón derecho sobre el icono del programa que uses para línea de comandos y "abrir como administrador". Eso permitirá que Angular CLI disponga de los permisos necesarios para realizar las tareas que requiere.

ACTUALIZACIÓN: Esto lo han cambiado en una versión beta de Angular CLI (Beta 6), por lo que ya no hace falta privilegios de administrador para usar las herramientas de línea de comandos de Angular 2.

En Linux o Mac, si te ocurre algo similar, simplemente tendrías que lanzar los comandos como "sudo".

Angular CLI tiene mucho más

En esta pequeña introducción solo te hemos explicado cómo iniciar un nuevo proyecto de Angular 2 y cómo servirlo después por medio del comando `serve`. Pero lo cierto es que detrás de Angular CLI hay muchas otras instrucciones de gran utilidad. Principalmente, como hemos comentado, encontrarás una gran cantidad de comandos que permiten crear el esqueleto de componentes, directivas, servicios, etc.

A medida que vayamos adentrándonos en el desarrollo con Angular 2 iremos aprendiendo de una forma sencilla y práctica todas las posibilidades que permite esta herramienta. De momento te recomendamos documentarte en el mencionado repositorio de Github.

Este artículo es obra de Alberto Basalo
Fue publicado por primera vez en 14/ 06/ 2016
Disponible online en <http://desarrolloweb.com/articulos/angular-di.html>

Análisis de las carpetas de un proyecto básico con Angular

Angular 2 exige un marco de trabajo más concreto y avanzado. Vemos los archivos y carpetas que nos hacen falta para comenzar un proyecto básico.

En el pasado artículo abordamos la herramienta [Angular CLI](#) y ahora vamos a introducirnos en el resultado que obtenemos mediante el comando `"ng new"`, que vimos servía para generar el esqueleto básico de una aplicación Angular 2.

No obstante, antes de entrar en materia, vamos a traer dos puntos interesantes. Uno de ellos es una reflexión previa sobre la complejidad de un proyecto "vacío" para una aplicación Angular 2. El otro es una de las preguntas típicas que se hacen cuando una persona se inicia con cualquier herramienta nueva: el editor que se recomienda usar.



Profesionalización

Todas las mejoras que nos ofrece Angular 2 tienen un coste a nivel técnico. Anteriormente (versiones 1.x) podíamos comenzar una aplicación de Angular con un código de inicio muy sencillo y sin necesidad de configurar diversas herramientas frontend. Básicamente podíamos enlazar Angular desde su CDN y con un script muy elemental empezar a usarlo.

Esto ya no es así. Con Angular 2 el número de piezas que necesitamos integrar, incluso en aplicaciones tan sencillas como un "hola mundo", es mucho mayor.

En aplicaciones grandes, donde existía un entorno bien definido por parte del equipo de desarrollo de esa aplicación (linter, transpiler, loader, tester, deployer...) se equilibra. Es decir, Angular no nos exige nada nuevo que ya no estuvieran usando equipos de desarrollo de aplicaciones grandes. Pero lo cierto es que aplicaciones pequeñas tendrán bastante más complejidad. Todo esto nos lleva a que desarrolladores ocasionales encontrarán más difícil el uso del framework en sus pasos iniciales, pero el esfuerzo sin duda merecerá la pena porque los pondrá a un nivel muy superior y eso redundará en su beneficio profesional.

¿Qué editor de código usar con Angular 2?

Se puede usar cualquier editor de código. Es una aplicación HTML y Javascript / TypeScript, por lo que puedes usar cualquier editor de los que vienes usando para cualquiera de estos lenguajes.

Como recomendación se sugiere usar un editor ligero, pero que te facilite la programación. Entre los que encontramos de código libre y gratuitos para cualquier uso están Brackets, Atom o Visual Studio Code. Éste último es quizás el más indicado, porque ya viene configurado con una serie de herramientas útiles y clave para desarrollar con Angular 2 como es el "intellisense" de TypeScript. De todos modos, a través de plugins podrás hacer que tu editor preferido también sea capaz de mostrarte las ayudas en tiempo de programación del compilador de TypeScript.

Archivos y carpetas con un proyecto de Angular 2.1 y Angular 4

Comenzaremos explicando los archivos y carpetas que encuentras en una aplicación recién creada con el [Angular CLI](#), tal como lo encontrarás si usas Angular 4, y en versiones más adelantadas de Angular 2. Más adelante en este mismo artículo explicaremos también carpetas que aparecían en las primeras versiones de Angular 2, que no son exactamente los mismos. El motivo principal de estas diferencias es que el tooling que se usa en Angular cambió poco después de liberar la versión 2, por ejemplo con la incorporación de WebPack. Con ello también la estructura de las carpetas básicas de un proyecto Angular cambió. No es una diferencia muy grande, y creemos que es útil que mantengamos en el manual ambas alternativas.

Archivos sueltos:

Encontrarás varios archivos sueltos en la carpeta raíz de tu proyecto. Te señalamos algunos importantes que debes conocer:

index.html Este archivo es información básica del proyecto recién creado. Te puede dar una idea inicial de qué es lo que encontrarás en estas carpetas y cómo usar el proyecto. Complementará sin duda las explicaciones de este [manual de Angular](#). Encontrarás algunos comandos explicados del CLI, como "ng serve" el cuál ya hemos tratado, para servir el proyecto. También comentará que tienes que hacer para realizar el build, comando "ng build" y llevar a producción una aplicación.

.angular-cli.json Es un archivo oculto (en Linux o Mac, pues comienza por ".") en el que se almacenan configuraciones del CLI de Angular.

package.json Este es el archivo que resume las dependencias del proyecto, las librerías sobre las que nos apoyamos, que se gestionan por medio de npm.

tslint.json Este archivo sirve para configurar el linter, el programa que nos alertará cuando tengamos problemas de estilo en el código.

.editorconfig Este es un archivo que sirve para definir la configuración para el editor de código que estemos utilizando. Permite centralizar la configuración, de modo que sea común para todos los desarrolladores que vayan a trabajar en el proyecto.

Carpeta src

Es la carpeta donde están las fuentes del proyecto. Esta carpeta es la que usaremos para desarrollar la aplicación y donde iremos colocando componentes y otro tipo de artefactos necesarios para poner en marcha nuestras ideas.

Entro de "src" encuentras muchos archivos que seguramente te serán familiares, como el index.html, que hace de raíz del proyecto. Es interesante que abras ese archivo para comprobar cómo es la raíz de una aplicación Angular. Te debería llamar la atención el código que encontrarás en el body:

```
<app-root></app-root>
```

Ese "app-root" es el componente raíz de la aplicación. En el desarrollo basado en componentes es un patrón normal que toda la aplicación se construya en un componente principal, del que colgará todo un árbol de componentes especializados en hacer cada una de las cosas necesarias.

Nota: En Angular se desarrolla en base a componentes. Si nunca has conocido este paradigma quizás te parezca extraño, pero poco a poco lo iremos entendiendo todo y apreciando las ventajas. Obviamente, no lo podemos explicar todo en un artículo, así que te pedimos un poco de paciencia.

Otro de los detalles que encontrarás en "src" son varios archivos con extensión ".ts". Son archivos con

código TypeScript. Recuerda que en Angular se programa usando TypeScript y que en el proceso de transpilado de la web, realizado por WebPack, ese código pasará a traducirse a Javascript. No hace falta que te preocupes mucho todavía, pues ya lo estudiaremos con calma. Puedes abrir si quieres el `main.ts`, que es el código de TypeScript principal de la aplicación, el punto de inicio de ejecución, aunque ya te advertimos que este archivo prácticamente no lo tendremos que tocar.

Dentro de "src" encontrarás también una carpeta llamada "app", que contiene el código del componente principal, que está dividido en varios archivos. Si abres el archivo ".html" verás el código de presentación del componente, que es el que se muestra cuando se visualiza la aplicación recién creada.

En la carpeta "src" hay muchos otros archivos, pero no queremos aburrirte con todos los detalles, pues los tendremos que analizar poco a poco.

Carpeta node_modules

Es la carpeta donde npm va colocando todas las dependencias del proyecto, es decir, el código de todas las librerías o componentes que estemos usando para basarnos en el desarrollo de una aplicación. Por ejemplo, el propio Angular es una dependencia.

Carpeta e2e

En esta carpeta se colocan los archivos para la realización de las pruebas "end to end".

Otras carpetas

A medida que trabajes podrás encontrar en el proyecto carpetas como "test" o "dist", para realizar el test o para almacenar los archivos listos para producción. Sigue leyendo las carpetas disponibles en el proyecto Angular 2.0 para más información.

Archivos y carpetas del proyecto con Angular 2.0

Impacta un poco que, recién creado un proyecto para Angular 2 por medio de Angular CLI, veamos en la carpeta de archivos varias subcarpetas, y varias de ellas con el contenido de cientos de ficheros. No obstante, no todo es código de tu aplicación, sino muchas veces son carpetas para crear la infraestructura de todo el tooling NodeJS incluido para gestionar una aplicación Angular 2.

Ahora conoceremos las partes que nos hacen falta para comenzar, aunque sin entrar en demasiados detalles.

Todos los archivos del raíz: Seguro que muchos de los lectores reconocen muchos de los archivos que hay dentro, como `package.json` (descriptor de dependencias npm) o `.gitignore` (archivos y carpetas que git debería ignorar de este proyecto cuando se añada al repositorio). En resumen, todo lo que encontraremos en esta raíz no son más que archivos que definen nuestro proyecto y configuran el entorno para diversas herramientas.

Nota: Observarás que no hay un `index.html`, porque esta no es la carpeta raíz de los archivos que se deben servir por el servidor web.

src: Es la carpeta más interesante para ti como desarrollador, ya que es el lugar donde colocarás el código fuente de tu proyecto. En realidad más en concreto la carpeta "app" que encontrarás dentro de "src" es donde tienes que programar tu aplicación. Observarás que ya viene con diversos contenidos, entre otras cosas el index.html que debe servir como página de inicio. No obstante, no es exactamente el directorio raíz de publicación, porque al desplegar el proyecto los resultados de compilar todos los archivos se llevarán a la carpeta "dist".

En la carpeta src es donde vas a realizar todo tu trabajo como desarrollador. Seguramente otros muchos archivos te resulten familiares, como el favicon.ico.

Verás además varios archivos .ts, que son código fuente TypeScript. Como quizás sepas, los archivos .ts solo existen en la etapa de desarrollo, es decir, en el proyecto que el navegador debe consumir no encontrarás archivos .ts, básicamente porque el navegador no entiende TypeScript. Esos archivos son los que se compilarán para producir el código .js que sí entienda el navegador.

Nota: Si todavía te encuentras reticente al uso de TypeScript no te preocupes, ya que cualquier código Javascript que pongas en ese fichero es código TypeScript válido. Por tanto tú podrías perfectamente escribir cualquier código Javascript dentro de los archivos .ts y todo irá perfectamente. Si además conoces algo de TypeScript y lo quieres usar para facilitarte la vida en tiempo de desarrollo, tanto mejor para ti.

dist: Es la versión de tu aplicación que subirás al servidor web para hacer público el proyecto. En dist aparecerán todos los archivos que el navegador va a necesitar y nunca código fuente en lenguajes no interpretables por él. (Observa que no hay archivos .ts dentro de dist). Ojo, pues muy probablemente tengas que iniciar el servidor web integrado en Angular CLI para que aparezca la carpeta "dist" en el directorio de tu proyecto. Puedes obtener más información sobre cómo lanzar el servidor web en el [artículo de Angular CLI](#).

Public: Es donde colocas los archivos estáticos del proyecto, imágenes y cosas similares que se conocen habitualmente como "assets". Estos archivos también se moverán a "dist" para que estén disponibles en la aplicación una vez subida al servidor web desde donde se va a acceder.

e2e: Es para el desarrollo de las pruebas. Viene de "end to end" testing.

node_modules: Son los archivos de las dependencias que mantenemos vía npm. Por tanto, todas las librerías que se declaren como dependencias en el archivo package.json deben estar descargados en esta carpeta node_modules. Esta carpeta podría haber estado dentro de src, pero está colgando de la raíz porque vale tanto para las pruebas, como para la aplicación cuando la estás desarrollando.

tmp: Es una carpeta que no tocaremos, con archivos temporales que generará Angular CLI cuando esté haciendo cosas.

Typings: Esto son definiciones de tipos usados por las librerías que usa un proyecto en Angular 2. Estos tipos te sirven para que el editor, gracias a TypeScript, pueda informarte con el "intellisense" en el acto de escribir código, sobre las cosas relacionadas con esas librerías.

De momento eso es todo, esperamos que esta vista de pájaro te sirva de utilidad para reconocer la estructura básica de un proyecto a desarrollar con Angular 2. En el siguiente artículo entraremos en detalle ya sobre el código, analizando por dentro algunas de estas carpetas y archivos generados desde Angular CLI y realizando nuestras primeras pruebas.

Este artículo es obra de Alberto Basalo
Fue publicado por primera vez en 20/ 06/ 2016
Disponible online en <http://desarrolloweb.com/articulos/analisis-carpetas-proyecto-angular2.html>

Vista de pájaro al código de una aplicación Angular (4)

Una aproximación a los elementos principales que encontramos en una aplicación Angular, versión 4, entendiendo el código.

En el capítulo anterior ofrecimos una descripción general de la [estructura de carpetas de una aplicación Angular](#), junto con algunos de sus principales archivos. Ahora vamos a introducirnos en el código y tratar de entender cómo está construida, al menos cuál es el flujo de ejecución de los archivos que vamos a encontrar.

La aplicación que vamos a radiografiar es la que conseguimos mediante el comando "ng new" del CLI. Esperamos que ya tengas tu aplicación generada. Si no es así, te recomendamos antes la lectura del artículo de [Angular CLI](#). Vamos a usar la versión 4 de Angular, aunque serviría para cualquier versión del framework a partir de la 2.

Obviamente no podremos cubrir todos los detalles, pero esperamos que de este artículo te ofrezca bastante luz sobre cómo se ejecutan los archivos principales de Angular. Verás que iremos saltando de un archivo a otro, dentro de la aplicación básica generada con "ng new", recorriendo básicamente el flujo de ejecución. Al final tendrás un resumen completo de los pasos de este recorrido que esperamos te sirva para apreciarlo de manera general.



Nota: lo cierto es que este artículo cubre Angular desde que se pasó a Webpack. En las versiones tempranas de Angular 2, usaban SystemJS para la gestión y carga de dependencias. El código de una aplicación usando esa versión antigua ya se explicó en el artículo [Zambullida en el código del proyecto inicial de Angular 2](#).

Archivo Index.html

Es de sobra sabido que las aplicaciones web comienzan por un archivo llamado index.html. Con Angular además, dado que se construyen páginas [SPA \(Single Page Application\)](#) es todavía más importante el index.html, pues en principio es el archivo que sirve para mostrar cualquier ruta de la aplicación.

El index.html en nuestra aplicación Angular, como cualquier otro archivo de los que forman parte de la app, se encuentra en el directorio "src". Si lo abres podrá llamarte la atención al menos un par de cosas:

Ausencia de Javascript:

No hay ningún Javascript de ningún tipo incluido en el código. En realidad el código Javascript que se va a ejecutar para inicializar la aplicación es inyectado por Webpack y colocado en el index.html cuando la aplicación se ejecuta o cuando se lleva a producción.

El BODY prácticamente vacío:

En el cuerpo de la página no aparece ningún HTML, salvo una etiqueta "app-root". Esta es una etiqueta no estándar, que no existe en el HTML. En realidad es un componente que mantiene todo el código de la aplicación.

Nota: Deberíamos hacer un stop al análisis del código para nombrar la "Arquitectura de componentes". En Angular y en la mayoría de librerías y frameworks actuales se ha optado por crear las aplicaciones en base a componentes. Existe un componente global, que es la aplicación entera y a su vez éste se basa en otros componentes para implementar cada una de sus partes. Cada componente tiene una representación y una funcionalidad. En resumen, las aplicaciones se construyen mediante un árbol de componentes, que se apoyan unos en otros para resolver las necesidades. En el index.html encontramos lo que sería el componente raíz de este árbol. Comenzamos a ver más detalle sobre los componentes en el siguiente artículo [Introducción a los componentes en Angular 2](#).

Archivo main.ts

Como hemos visto, no existe un Javascript definido o incluido en el index.html, pero sí se agregará más adelante para que la aplicación funcione. De hecho, si pones la aplicación en marcha con "ng serve -o" se abrirá en tu navegador y, al ver el código fuente ejecutado podrás ver que sí hay código Javascript, colocado antes de cerrar el BODY del index.html.

Ese código está generado por Webpack, e inyectado al index.html una vez la página se entrega al navegador. El script Javascript que el navegador ejecutará para poner en marcha la aplicación comienza por el archivo main.ts, que está en la carpeta "src".

Nota: Podría llamarte la atención que no es un archivo Javascript el que contiene el código de la aplicación, pero ya dijimos en la [Introducción a Angular](#), que este framework está escrito usando el lenguaje [TypeScript](#). En algún momento ese código se traducirá, lógicamente, para que se convierta en Javascript entendible por todos los navegadores. Ese proceso lo realiza Webpack pero realmente no debe preocuparnos mucho ya que es el propio Angular CLI que va a realizar todas las tareas vinculadas

con Webpack y el compilador de Javascript para la traducción del código, de manera transparente para el desarrollador.

Si abres el `main.ts` observarás que comienza realizando una serie de "import", para cargar distintas piezas de código. Esos import son típicos de ES6, aunque en este caso te los ofrece el propio TypeScript, ya que el transpilador de este lenguaje es el que los convertirá a un Javascript entendible por todos los navegadores.

Los import que encuentras los hay de dos tipos.

1.- imports de dependencias a librerías externas

Estos import son código de otros proyectos, dependencias de nuestra aplicación. Están gestionados por npm y han sido declarados como dependencias en el archivo `package.json` que deberías conocer. Tienen la forma:

```
import { enableProdMode } from '@angular/core';
```

En este caso nos fijamos en `'@angular/core'` que es una de las librerías dependientes de nuestra aplicación, instaladas vía npm, cuyo código está en la carpeta `"node_modules"`. En esa carpeta, de código de terceros, nunca vamos a tocar nada.

2.- Imports a código de nuestro propio proyecto

También encontraremos imports a elementos que forman parte del propio código de la aplicación y que por tanto sí podríamos tocar. Los distingues porque tienen esta forma:

```
import { AppModule } from './app/app.module';
```

Aquí nos fijamos en la ruta del módulo que se está importando `'./app/app.module'`, que es relativa al propio `main.ts`.

Nota: en la ruta al archivo que se está importando falta el `".ts"`, ya que el archivo es un TypeScript. Pero realmente no se debe colocar, puesto que ese archivo `.ts` en algún momento se traducirá por un archivo `.js`. Para no liarse, recuerda que los import a archivos TypeScript no se coloca la extensión.

Obviamente, para profundizar habría que entender qué es cada import de los que se van realizando, los cuales tendremos tiempo de analizar llegado el momento.

Además de los imports, que es código que se va requiriendo, hay un detalle que encontramos al final del fichero:


```
platformBrowserDynamic().bootstrapModule(AppModule)
    .catch(err => console.log(err));
```

Esto es el bootstrap, el arranque, de la aplicación, que permite decirle a Angular qué tiene que hacer para comenzar a dar vida a la aplicación. Al invocar al sistema de arranque estamos indicando qué módulo (AppModule) es el principal de la aplicación y el que tiene los componentes necesarios para arrancar.

Al final, el bootstrap provocará que Angular lea el código del módulo y sepa qué componentes existen en el index, para ponerlos en marcha, provocando que la aplicación empiece a funcionar, tal como se haya programado.

Nota: fíjate que para arrancar la aplicación se hace uso de los elementos que hemos importado en las líneas anteriores, entre ellos platformBrowserDynamic y AppModule.

Archivo app.module.ts

De entre todos los imports que se hace en el main.ts hay uno fundamental, que nos sirve para tirar del hilo y ver qué es lo que está pasando por abajo, para hacer posible que todo comience a funcionar. Se trata de app.module.ts, que podemos considerar como el módulo principal de la aplicación.

En este fichero nos encontramos, como viene ya siendo habitual, varios imports de elementos que vienen de otros módulos, pero hay además dos cosas que pueden ser clave para entender el flujo de ejecución.

Import de nuestro componente raíz

En el módulo principal, app.module.ts, encontramos el import al componente raíz de la aplicación (esa etiqueta HTML no-estándar que aparecía en el BODY del index).

```
import { AppComponent } from './app.component';
```

Este componente es importante en este punto porque es el primero que estamos usando y porque es el único que te ofrecen ya construido en la aplicación básica creada con el CLI de Angular.

Al importar el componente lo que estamos obteniendo es una referencia "AppComponent", donde estará la clase creada para implementar el componente.

Decorador @NgModule

Esta es la primera vez que quizás estás conociendo los decoradores, algo que viene directamente otorgado por TypeScript. Los decoradores permiten asignar metadata a funciones, clases u otras cosas. Las funciones decoradoras tienen un nombre y las usamos para asignar esos datos, que podrían modificar el comportamiento de aquello que se está decorando.

El decorador completo en la versión de Angular que estamos usando (4) es este:

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

Como ves, a la función decoradora la alimentamos con un objeto pasado por parámetro, en el que indicamos diversos datos útiles para la clase AppModule. Este decorador se irá editando y agregando nuevas cosas a medida que vayamos desarrollando, por lo que nos resultará bastante familiar a poco que comencemos a desarrollar con Angular.

En el decorador hay una propiedad llamada "bootstrap", que contiene un array con los componentes que Angular tiene que dar vida, para conseguir que las cosas comiencen a funcionar: bootstrap: [AppComponent]. Esto le dice a Angular que hay en el index.html hay un componente implementado con la clase AppComponent, que tiene que ejecutar.

Componente raíz de la aplicación

Para llegar al meollo del código, tenemos que observar, aunque todavía por encima, el componente raíz. Lo habíamos usado en el index.html:

```
<app-root></app-root>
```

Pero echemos un vistazo al código del componente. Su ruta la puedes deducir del import que se ha realizado en el app.module.ts, o sea "src/ app/ app.component.ts".

De momento solo queremos que encuentres tu segundo decorador, con el código siguiente.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

- El selector o etiqueta que implementa este componente: "app-root". Justamente es la etiqueta extraña que había en el index.html.
- El archivo .html su template: " ./ app.component.html".
- Los estilos CSS que se usan en el componente: " ./ app.component.css"

Con lo que acabas de ver, será fácil abrir encontrar el archivo donde está el template. ¿no? Puedes abrirlo y

verás el código HTML que aparecía al poner en marcha la aplicación. (Comando "ng serve -o" del CLI)

Resumen del flujo de ejecución de la aplicación básica Angular (4)

Somos conscientes que hemos aportado muchos datos en este artículo, quizás demasiados para asimilarlos en una primera lectura. Si todo es nuevo para ti debe de ser un poco complicado hacerse un esquema perfecto del funcionamiento de las cosas, así que vamos a parar un momento para repasar de nuevo el flujo de ejecución del código.

1. En el index.html tenemos un componente raíz de la aplicación "app-root".
2. Al servir la aplicación (ng serve), o al llevarla a producción (ng build) la herramienta Webpack genera los paquetes (bundles) de código del proyecto y coloca los correspondientes scripts en el index.html, para que todo funcione.
3. El archivo por el que Webpack comienza a producir los bundles es el main.ts
4. Dentro del main.ts encontramos un import del módulo principal (AppModule) y la llamada al sistema de arranque (bootstrapModule) en la que pasamos por parámetro el módulo principal de la aplicación.
5. En el módulo principal se importa el componente raíz (AppComponent) y en el decorador @NgModule se indica que este componente forma parte del bootstrap.
6. El código del componente raíz tiene el selector "app-root", que es la etiqueta que aparecía en el index.html.
7. El template del componente raíz, contiene el HTML que se visualiza al poner en marcha la aplicación en el navegador.

Conclusión

Por fin hemos llegado a seguir la ejecución de la aplicación, para encontrar su contenido. Seguro que habiendo llegado a este punto sentirás una pequeña satisfacción. Puedes aprender más de los componentes en el artículo [Introducción a los componentes en Angular 2](#).

Somos conscientes que mucho del conocimiento ha quedado en el aire y hay muchas dudas que querrás resolver, pero con lo que hemos visto hemos conseguido el objetivo planteado, disponer de una vista de pájaro del código que encuentras al iniciar tu aplicación Angular.

Puedes continuar la lectura del [Manual de Angular](#) para obtener más información sobre decoradores, componentes, y arranque de las aplicaciones.

Este artículo es obra de Miguel Ángel Álvarez
Fue publicado por primera vez en 05/ 10/ 2017
Disponible online en <http://desarrolloweb.com/articulos/codigo-aplicacion-angular.html>

Zambullida en el código del proyecto inicial de Angular 2

Comenzamos a analizar el código de un proyecto básico con Angular 2, en las primeras versiones, generado con Angular CLI. Prestamos al index de la aplicación y al componente principal, que podremos editar para comprobar el funcionamiento.

En este artículo vamos a analizar el código de la aplicación inicial que se instala al hacer un nuevo proyecto con Angular CLI. Ten en cuenta que en artículos anteriores del [Manual de Angular 2](#) hemos abordado ya en una primera instancia la estructura de carpetas generada por Angular CLI, además de explicar cómo crearla a través de comandos de consola.

Por tanto, nos centraremos en entender cuál es el flujo de ejecución del código, junto con los archivos que se van incluyendo y recorriendo al ejecutar la aplicación. Obviamente, habrá cosas en las que no podamos entrar todavía en muchos detalles, pero no hay que preocuparse porque serán materia de estudio en sucesivos artículos.

NOTA IMPORTANTE: El contenido de este artículo aplica a versiones tempranas de Angular 2. Al Principio el sistema para la gestión de paquetes y dependencias era SystemJS. Sin embargo, actualmente Angular trabaja con Webpack. Ya en la versión 2 de Angular se migró a Webpack y hubo algunos cambios relevantes en la manera de construirse la aplicación básica de Angular con el Angular CLI. Es por ello que el contenido seguramente no sea relevante para ti. Si entender el código de Angular 2 o Angular 4 por favor lee el artículo [Vista de pájaro al código de una aplicación Angular](#). Sin embargo, muchos datos ofrecidos en el siguiente texto siguen teniendo vigencia y pueden aclarar puntos interesantes sobre la programación usando este framework.



Archivo package.json

Vamos a comenzar por analizar este archivo, ya que muchas cosas son declaradas inicialmente en él. Al analizar este archivo encontraremos el origen de muchas de las librerías que usa Angular 2.

Como debes saber, package.json es un archivo en el que se declaran las dependencias de una aplicación, gestionadas vía npm. Su código es un JSON en el que se declaran varias cosas.

Inicialmente hay que ver la propiedad "dependencies". En ella encontrarás la librería Angular separada en varios módulos. Esta es una de las características de la nueva plataforma de desarrollo promovida por Angular 2, la modularización del código. Encontrarás que una aplicación Angular 2 necesita ya de entrada diversos módulos como son "common", "compiler", "core", etc.

Luego hay una serie de librerías de terceros, no creadas directamente por el equipo de Angular, o bien creadas por ellos mismos pero con un enfoque universal (capaz de usarse en otros tipos de proyectos). Son "es6-shim", "rxjs", etc.

Todos estos módulos antes se incluían por medio de scripts (etiqueta SCRIPT) en el HTML de la página. Pero en esta versión y gracias a Angular CLI ya viene incorporada una mejor manera de incluir módulos Javascript, por medio de "SystemJS". Observarás que el propio SystemJS está incluido como dependencias

"systemjs" y con él podríamos incluir todo tipo de código, no solo JS, sino otros ficheros necesarios como CSS.

Nota: Al describir la [estructura de carpetas del proyecto Angular 2](#) ya explicamos que todas las dependencias de package.json te las instala npm en la carpeta "node_modules".

Entendiendo lo básico de SystemJS

Para comenzar por nuestro análisis del código, vamos a abrir el archivo "src/ index.html".

Como dijimos, la carpeta src es donde se encuentran las fuentes de tu proyecto. En ella encontramos un index.html que es la raíz de la aplicación. Todo empieza a ejecutarse a través de este archivo.

Te llamará la atención el código siguiente:

```
{{#each scripts.polyfills}}<script src="{{.}}"></script>{{/each}}
```

Ese código hace un recorrido por una serie de librerías y las va colocando dentro de etiquetas SCRIPT para incluir su código en la aplicación. De momento lo que debes saber sobre este código es que por medio de él se cargan librerías externas que necesita Angular y que hemos visto declaradas en las "dependencies" del archivo package.json. Lo que nos interesa saber es que así se está cargando, entre otras, la librería SystemJS.

Ahora, en el final de index.html encontrarás allí un script. Cuando llegamos a este punto, SystemJS ya está cargado y en este Script se realiza una inicialización de esta librería para cargar los elementos necesarios para comenzar a trabajar:

```
<script>
  System.import('system-config.js').then(function () {
    System.import('main');
  }).catch(console.error.bind(console));
</script>
```

Encontramos el objeto "System", que es una variable global definida por la librería SystemJS. Por medio del método "import" consigue cargar módulos. Apreciarás que se está cargando inicialmente un archivo llamado "system-config.js".

Luego vemos el método then() y catch(). Estos métodos los deberías de reconocer, pues son pertenecientes a un patrón bien conocido por los desarrolladores Javascript, el de promesas. El método then() se ejecutará cuando se termine de cargar el módulo "system-config.js" y el método catch() se ejecutaría en caso que no se haya podido cargar ese archivo. Gracias a then(), después de haber cargado "system-config.js" entonces se cargará "main", que enseguida veremos qué es.

En este punto te preguntarás ¿Dónde está system-config.js?. Quizás no lo encuentres, pero veas en la misma carpeta "src" el archivo system-config.ts. Ese es un archivo TypeScript que contiene el código de system-

config.js antes de transpilar con el TypeScript Compiler.

Nota: TypeScript es un lenguaje para el programador. Lo que usa el navegador es Javascript. El TypeScript compiler se encargará de hacer esa conversión del .ts a un .js.

El archivo system-config.ts generalmente no lo vas a tener que tocar, porque Angular CLI te lo irá actualizando. Si lo abres sin duda irás reconociendo algunas líneas, cosas que necesitará SystemJS. No vamos a entrar ahora en el detalle, de momento quédate con que es código generado.

Por su parte la referencia a "main" que teníamos antes en los import del index.html System.import('main'), es un import. No le ponen ni siquiera la extensión del archivo "main.js" y esto es porque "main" es un alias declarado en el system-config.ts. Fíjate en el código del archivo, en estas líneas:

```
// Apply the CLI SystemJS configuration.
System.config({
  map: {
    '@angular': 'vendor/@angular',
    'rxjs': 'vendor/rxjs',
    'main': 'main.js'
  },
  packages: cliSystemConfigPackages
});
```

El objeto "map" tiene una lista de alias y archivos que se asocian. Siendo que "main" corresponde con "main.js". Nuevamente, no encontrarás un "main.js" entre los archivos del proyecto, en la carpeta "src", porque lo que tendremos es un main.ts que luego se convertirá en el main.js.

Ahora puedes abrir main.js. Verás que su código nuevamente hace uso de SystemJS, realizando diversos imports. Estos imports son como los que conoces en ECMAScript 6 y básicamente lo que te traen son objetos de diversas librerías.

Encontrarás este código en main.js, o algo similar:

```
import { bootstrap } from '@angular/platform-browser-dynamic';
import { enableProdMode } from '@angular/core';
import { TestAngular2AppComponent, environment } from './app/';
```

Por ejemplo, estás diciéndole que importe el objeto "bootstrap" de la librería "@angular/ platform-browser-dynamic". Esa librería está declarada dentro de "system-config.ts"

Luego verás otras líneas, que es el bootstrap, o inicio de la aplicación Angular. No vamos a entrar en detalle, pero es lo equivalente al "ng-app" que colocabas antes en el código HTML de tu index. Esto, o algo parecido ya se podía hacer con Angular 1 y se conocía como el arranque manual de Angular.

Componente principal de una aplicación

Seguimos analizando index.html y encontramos en el código, en el cuerpo (BODY) una etiqueta que llamará la atención porque no es del HTML tradicional. Es el uso de un componente y su código será algo como:

```
<mi-nombre-proyecto-app>Loading...</mi-nombre-proyecto-app>
```

Ese es el componente raíz de nuestra aplicación Angular 2. Hablaremos de componentes con detalle más adelante. De momento para lo que te interesa a ti, que es reconocer el flujo de ejecución básico, hay que decir que su código está en la carpeta "src/ app".

En esa carpeta encontrarás varios archivos del componente que analizaremos con calma más adelante. De momento verás un archivo ".html" que contiene la vista de este componente y un archivo ".css" que contiene el CSS. Si tu componente se llamaba "mi-nombre-proyecto-app", estos archivos se llamarán "mi-nombre-proyecto.component.html" y "mi-nombre-proyecto.component.css".

Para terminar esta primera zambullida al código te recomendamos editar esos archivos. Es código HTML y CSS plano, por lo que no tendrás ningún problema en colocar cualquier cosa, siempre que sea HTML y CSS correcto, claro está.

Para quien use Angular 1 ya reconocerá una estructura como esta:

```
{{title}}
```

Es una expresión que Angular 2 sustituirá por un dato que se declara en el archivo .ts del componente. De momento lo dejamos ahí.

Ejecutar el proyecto

Para comprobar si tus cambios en el HTML del componente han tenido efecto puedes probar el proyecto. La ejecución de esta aplicación ya la vimos en el artículo de [Angular CLI](#), por lo que no necesitarás mayores explicaciones. De todos modos, como resumen, sigue los pasos:

Desde la raíz del proyecto, con el terminal, ejecutamos el comando:

```
ng serve
```

Luego nos dirigimos a la URL que nos indican como resultado de la ejecución del comando, que será algo como:

```
http://localhost:4200/
```

Entonces deberías ver la página funcionando en tu navegador, con el HTML editado tal como lo has dejado en el archivo .html del componente principal.

En futuros artículos profundizaremos sobre muchos de los puntos relatados aquí. De momento creemos

que esta introducción al código debe aclararte muchas cosas, o plantearte muchas otras dudas.

Si es tu caso, no te preocupes por sentirte despistado por tantos archivos y tantas cosas nuevas, pues poco a poco iremos familiarizándonos perfectamente con toda esta infraestructura. Para tu tranquilidad, decir que esta es la parte más compleja y que, a partir de aquí, las cosas serán más agradecidas. Si además vienes de Angular 1, empezarás a reconocer mejor las piezas que antes existían en el framework.

Este artículo es obra de Alberto Basalo
Fue publicado por primera vez en 29/ 06/ 2016
Disponible online en <http://desarrolloweb.com/articulos/codigo-proyecto-inicial-angular2.html>

Introducción a los componentes en Angular

Un primer acercamiento al mundo de los componentes en Angular (2 en adelante), a través del componente inicial que se crea en todo nuevo proyecto. Aprenderás a reconocer sus partes y realizaremos unas modificaciones.

En Angular se desarrolla en base a componentes. Desde la aplicación más básica de Angular (2, 4 o en adelante), el Hola Mundo, todo tiene que comenzar por un componente. Nuestra aplicación "componetizada" se construirá, la verdad, en base a un árbol de componentes de "n" niveles, desde un componente principal a sus hijos, nietos, etc.

Por si no lo sabes, los componentes son como etiquetas HTML nuevas, que podemos inventarnos para realizar las funciones que sean necesarias para nuestro negocio. Pueden ser cosas diversas, desde una sección de navegación a un formulario, o un campo de formulario. Para definir el contenido de esta nueva etiqueta, el componente, usas un poco de HTML con su CSS y por supuesto, un poco de Javascript para definir su funcionalidad.

Básicamente eso es un componente, una de las piezas fundamentales de las aplicaciones en Angular, que nos trae diversos beneficios que mejoran sensiblemente la organización de una aplicación, su mantenimiento, reutilización del código, etc.



Para comenzar a introducirnos en el desarrollo en base a componentes vamos a realizar en este primer artículo un análisis del componente inicial, que contiene de toda aplicación Angular y que podemos encontrar en el proyecto básico creado vía [Angular CLI](#).

Localizar el componente inicial

En el [Manual de Angular 2](#) hemos visto que nuestra aplicación se desarrolla en el directorio "src". Allí encontramos el archivo index.html raíz de la aplicación.. Si lo abres verás que no tiene ningún contenido en sí. Apenas encontrarás el uso de un componente, una etiqueta que no pertenece al HTML. Es fácil localizarlo porque es el único contenido del BODY de la página.

```
<app-root></app-root>
```

Nota: Dependiendo de tu versión de Angular este componente puede tener un nombre diferente. Además, también puede cambiar su nombre dependiendo de la configuración del Angular CLI, que escribes en el archivo ".angular-cli.json". En principio todos los componentes se crean con el prefijo "app", por lo que siempre empiezan con esas letras y un guión, como "app-root". Pero podrías cambiar el prefijo editando el fichero de configuración en la propiedad "prefix" y colocando cualquier otro valor. Ejemplo "prefix": "dw".

Este es el componente donde tu aplicación Angular 2 va a desarrollarse. Todos los demás componentes estarán debajo de éste, unos dentro de otros en un árbol. Todo lo que ocurra en tu aplicación, estará dentro de este componente.

Nota: En versiones tempranas de Angular 2 había un texto como contenido dentro del componente (Loading...) es lo que aparecerá en el navegador mientras no carga la página. Una vez que la aplicación se inicie, Angular lo sustituirá por el contenido definido para el propio componente, cuando arranque la aplicación.

Si al arrancar la aplicación (ng-serve) ves que ese mensaje de "Loading..." tarda en irse es porque estás en modo de desarrollo y antes de iniciarse la app tienen que hacerse varias tareas extra, como transpilado de código, que no se necesitarán hacer cuando esté en producción.

Entendiendo el código del componente

El código de este componente está generado de antemano en la carpeta "src/ app". Allí encontrarás varios ficheros que forman el componente completo, separados por el tipo de código que colocarás en ellos.

- app.component.html: Equivale a lo que conocemos por "vista" en la arquitectura MVC.
- app.component.css: Permite colocar estilos al contenido, siendo que éstos están encapsulados en este componente y no salen afuera.
- app.component.ts: Es el corazón de nuestro componente, un archivo con código TypeScript, que se traducirá a Javascript antes de entregarse al navegador. Por si te sirve la comparación, sería el equivalente al controlador en el MVC, aunque en Angular 2 desapareció el controlador tal como se conocía en AngularJS (1.x).
- app.component.spec.ts: Un archivo TypeScript destinado a tareas de testing de componentes.

En este archivo encontrarás diverso contenido, expresado en código HTML. Como hemos dicho, este archivo es lo que sería la vista y admite toda clase de código HTML, con etiquetas estándar y el uso de otros componentes. Además podemos colocar expresiones, declarar binding entre componentes, eventos, etc.

Nota: Los que no conozcan de todo eso que estamos hablando (expresiones, eventos, binding, etc.) no se preocupen, porque lo veremos más adelante.

Dentro del HTML de la vista, entre otras cosas, encontrarás:

```
{{title}}
```

Eso es una expresión. Angular lo sustituirá por el contenido de una variable "title" antes de mostrarlo al cliente. Esa variable se define en la declaración del componente.

Declaración del componente: `app.component.ts`

Este es el archivo con el script necesario para la creación del componente, creado mediante código TypeScript. Es como el controlador en el patrón MVC, solo que en Angular 2 no se le llama controlador, o "controller". Ahora es una clase normal, de programación orientada a objetos, como las que nos ofrece ES6, sólo que aquí es TypeScript quien nos la facilita.

Si abres el archivo `app.component.ts` encontrarás varias cosas.

- El import de "component" dentro de `@angular/ core`
- Una función decoradora que hace la acción de registrar el componente
- La clase que hace las veces de controlador

La función decoradora observarás que declara diversas cuestiones.

```
@Component({  
  moduleId: module.id,  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})
```

Una de ellas es el "selector" de este componente, o el nombre de la etiqueta que se usará cuando se desee representar. Mediante la propiedad "templateUrl" asociamos un archivo .html que se usará como vista del componente. Por último se define su estilo mediante la propiedad "styleUrls", indicando a un array de todas las hojas de estilo que deseemos.

En la clase del componente, que se debe colocar con un export para que se conozca fuera de este módulo, es la parte que representa el controlador en una arquitectura MVC. En ella colocaremos todas las propiedades y métodos que se deseen usar desde la vista.

```
export class AppComponent {  
  title = 'proyecto-angular2 works!';  
}
```

Esas propiedades representan el modelo de datos y se podrán usar expresiones en las vistas para poder visualizarlas.

Nota: Observa además que el nombre de la clase de este componente tiene una forma especial. Mientras que el nombre de la etiqueta del componente (su "selector") tiene las palabras separadas por guiones, aquí tenemos una notación "Pascal Case" típica de las clases (class de programación orientada a objetos). Esto es una constante. En el HTML que no se reconocen mayúsculas y minúsculas se separan las palabras por guiones por guiones, colocando todo en minúscula. Por su parte, los mismos nombres en Javascript se escriben con "Pascal Case", todo junto y con la primera letra de cada palabra en mayúscula.

Alterando el código de nuestro componente

Para terminar este artículo vamos a hacer unos pequeños cambios en el código del componente para comprobar si la magia de Angular está funcionando.

Algo muy sencillo sería comenzar por crear una nueva propiedad en la clase del componente. Pero vamos además a colocar un método para poder usarlo también desde la vista.

```
export class ProyectoAngular2AppComponent {  
  title = 'Manual de Angular de DesarrolloWeb.com';  
  visible = false;  
  decirAdios() {  
    this.visible = true;  
  }  
}
```

Nota: Esta clase "class" se escribe en un archivo TypeScript, pero realmente lo que vemos es casi todo Javascript válido en ES6 y ES7. TypeScript entiende todo ES6 e incluso algunas cosas de ES7.

Ahora vamos a ver el código HTML que podría tener nuestra vista.

```
<h1>  
  {{title}}  
</h1>  
<p [hidden]="!visible">  
  Adiós  
</p>  
<button (click)="decirAdios()">Decir adiós</button>
```

En este HTML hemos incluido más cosas de las que puedes usar desde Angular. Habíamos mencionado la expresión, entre llaves dobles, que permite volcar el contenido de propiedades del componente. También encuentras el uso de una propiedad de un elemento, como es "hidden", entre corchetes (nuevo en Angular 2). Además de la declaración de un evento "click" que se coloca entre paréntesis.

Nota: En el siguiente artículo explicaremos con detalle [toda esta sintaxis nueva que podemos usar en las vistas](#) para declarar eventos, hacer data-binding, etc.

Otro detalle que puedes observar es la propiedad "visible" del componente, que se usa para asignarla al atributo hidden del elemento "p". El método de la clase, decirAdios() se usa para asociarlo como manejador del evento "click".

Hablaremos más adelante de todas estas cosas que puedes colocar en las vistas y algunas otras, junto con las explicaciones sobre la sintaxis que se debe usar para declararlas.

Nota: Al modificar los archivos del componente, cualquiera de ellos, tanto el html, css o ts, se debería refrescar automáticamente la página donde estás visualizando tu proyecto una vez puesto en marcha con el comando "ng serve", gracias al sistema de "live-reload" que te monta Angular CLI en cualquier proyecto Angular 2.

Otra cosa interesante del entorno de trabajo es que, si usas Visual Studio Code u otro editor con los correspondientes plugin TypeScript, te informarán de posibles errores en los archivos .js. Es una ayuda muy útil que aparece según estás escribiendo.

Con esto acabamos nuestro primer análisis y modificaciones en el componente inicial. Estamos seguros que esta última parte, en la que hemos modificado el código del componente básico, habrá resultado ya algo más entretenida.

Este artículo es obra de Alberto Basalo
Fue publicado por primera vez en 06/ 07/ 2016
Disponible online en <http://desarrolloweb.com/articulos/introduccion-componentes-angular2.html>

Sintaxis para las vistas en Angular

Expresiones, binding, propiedades, eventos. Son muchas cosas las que podemos expresar en las vistas HTML de las aplicaciones Angular 2. Te ofrecemos una introducción general.

En los artículos anteriores del [Manual de Angular](#) hemos analizado la [estructura de una aplicación básica](#). Una de las cosas fundamentales que encontramos es el [componente principal](#), sobre el cual hemos hecho pequeñas modificaciones para comprobar que las cosas están funcionando y comenzar a apreciar el poder

del framework.

Dentro del componente básico encontramos varios archivos y uno de ellos es el código HTML del componente, al que comúnmente nos referiremos con el nombre de "vista", denominación que viene por el patrón de arquitectura MVC. Pues bien, en ese código HTML -la vista-, de manera declarativa, podemos definir y usar muchas de las piezas con las que contamos en una aplicación: propiedades, eventos, bindeo...

La novedad en Angular (2, 4 y en adelante), para los que vienen de la versión anterior del framework, es que ahora podemos ser mucho más precisos sobre cómo queremos que la información fluya entre componentes, entre la vista y el modelo, etc. En este artículo vamos a ofrecer una primera aproximación general a todo lo que se puede declarar dentro de una vista, teniendo en cuenta que muchas de las cosas necesitarán artículos específicos para abordarlas en profundidad.



Nota: En este artículo, así como a lo largo del manual, hacemos uso constantemente de conceptos del MVC. Recordamos que existe un artículo genérico de lo que es el MVC, [introducción al modelo vista controlador](#). De todos modos, en Angular 2 no se aplica una implementación perfectamente clara sobre el MVC. Existe una separación del código por responsabilidades, lo que ya nos aporta los beneficios de la arquitectura por capas, pero la implementación y características de estas capas no queda tan definida como podría darse en un framework backend. En este artículo y los siguientes verás que hacemos referencia a los modelos y realmente no es que exista una clase o algo concreto donde se coloca el código del modelo. Ese modelo realmente se genera desde el controlador y para ello el controlador puede obtener datos de diversas fuentes, como servicios web. Esos datos, tanto propiedades como métodos, se podrán usar desde la vista. Es algo más parecido a un VW (View Model), o sea, un modelo para la vista, que se crea en el controlador. Por otra parte, tampoco existe un controlador, sino una clase que implementa el View Model de cada componente. En resumen, conocer el MVC te ayudará a entender la arquitectura propuesta por Angular 2 y apreciar sus beneficios, pero debemos mantener la mente abierta para no confundirnos con conocimientos que ya podamos tener de otros frameworks.

HTML permitido en los Templates de Angular

Los templates de los componentes, o vistas de los componentes, ya sean colocados inline en el próximo código TypeScript o en un archivo aparte, se escriben con HTML y permiten colocar mediante la sintaxis de Angular expresiones, bindeos, eventos, etc. que vamos a introducir en este artículo.

Casi todo el HTML válido es un código potencialmente usable en los templates de Angular. Sin embargo hay algunas excepciones que conviene conocer.

No está permitido colocar scripts Javascript en el template

La más importante excepción de HTML válido a usar en un template (una vista) es la etiqueta `SCRIPT`, ya que puede ser origen de problemas de seguridad. Bajo el prisma de la separación del código por responsabilidades, no deberíamos colocar una etiqueta `SCRIPT` dentro de un template, pues la parte del desarrollo de la lógica de los componentes se debería colocar en el código `TypeScript`.

Pero, aún cometiendo la imprudencia o mala práctica de colocar una etiqueta `SCRIPT`, Angular hará caso omiso de ella y no ejecutará ese código `Javascript`, evitando posibles problemas e inyecciones de código no deseado.

Nota: esto mismo ocurre si en una cadena volcada en un template por interpolación `{{ }}` o bindeo a propiedad `[]` contiene una etiqueta `SCRIPT`. Angular se dará el trabajo de sanitizar el código a colocar en el template, evitando problemas como la inyección de código (xss).

Más tarde en este artículo explicamos la interpolación (sintaxis `{{ }}` dobles llaves) y el bindeo a propiedades (sintaxis `[]` dobles corchetes), que son dos de las principales piezas declarables en una vista. Aunque en ocasiones la interpolación la nombraremos como "expresiones" y el binding a propiedad lo nombramos simplemente como "propiedad". Nos referimos a lo mismo, esperamos no liarte.

No tiene sentido usar ciertas etiquetas

Hay otras etiquetas que tampoco tiene sentido usarse en un template, como `BODY` o `HEAD`. Un componente es una parte concreta de la página, no la cabecera, ni el cuerpo, por lo que no tendrían ninguna utilidad colocar esas etiquetas en un template.

Otra etiqueta no usable es `BASE`, que sirve para indicar por ejemplo la ruta de base a la que aplicar todas las rutas relativas de los enlaces en todo el documento HTML.

Por lo demás, puedes usar prácticamente cualquier otro HTML disponible en el lenguaje de marcación.

Piezas declarables en una vista

Comenzaremos por describir las cosas que disponemos para su declaración en una vista, de modo que todos podamos usar un único vocabulario.

- **Propiedad:** Cualquier valor que podemos asignar por medio de un atributo del HTML. Ese elemento puede ser simplemente un atributo del HTML estándar, un atributo implementado mediante el propio Angular 2 o un atributo personalizado, creado para un componente en específico.
- **Expresión:** Es un volcado de cualquier información en el texto de la página, como contenido a cualquier etiqueta. La expresión es una declaración que Angular procesará y sustituirá por su valor, pudiendo realizar sencillas operaciones.
- **Binding:** Es un enlace entre el modelo y la vista. Mediante un binding si un dato cambia en el modelo, ese cambio se representa en la vista. Pero además en Angular se introduce el "doble binding", por el cual si un valor se modifica en la vista, también viaja hacia el modelo. Con la novedad en Angular (2 en adelante) que el doble binding es opcional.
- **Evento:** es un suceso que ocurre y para el cual se pueden definir manejadores, que son funciones

que se ejecutarán como respuesta a ese suceso.

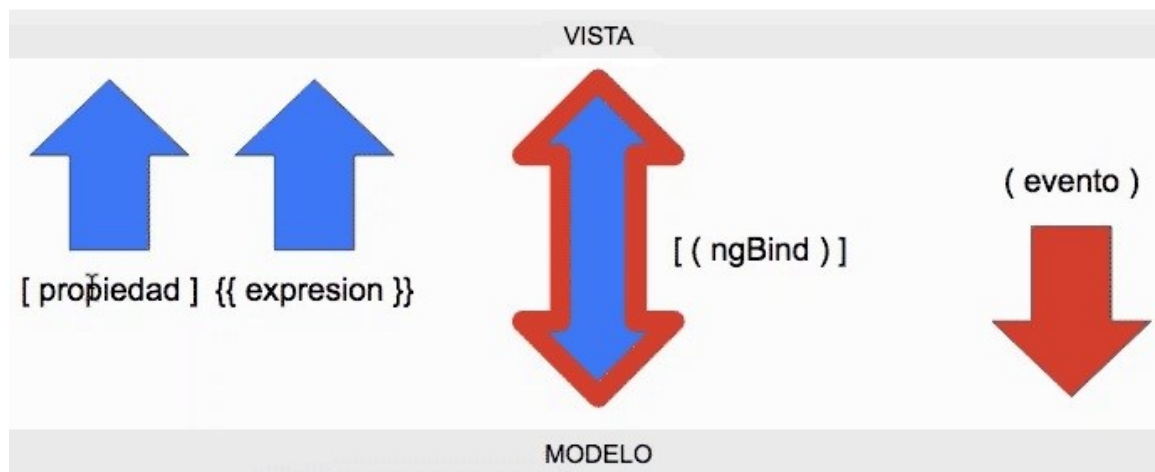
Nota: Generalmente cuando hablemos de "binding" en la mayoría de las ocasiones nos referimos a "doble binding", que es la principal novedad que trajo Angular 1 y que le produjo tanto éxito para este framework. Sin embargo, este mismo doble binding es un arma de doble filo, pues puede disminuir el rendimiento de la aplicación y en ocasiones puede producir un flujo de datos difícil de entender y depurar.

Debido al coste operacional del doble binding (coste en tiempo de procesamiento si la aplicación es muy compleja y se producen muchos enlaces), la velocidad de Angular puede verse afectada. Es el motivo por el que se han producido nuevas sintaxis para poder expresar bindings de varios tipos, de una y de dos direcciones. Dicho de otra manera, ahora se entrega al programador el control del flujo de la información, para que éste pueda optimizar el rendimiento de la aplicación.

Flujo de la información de la vista al modelo y modelo a vista

El programador ahora será capaz de expresar cuándo una información debe ir del modelo hacia la vista y cuándo debe ir desde la vista al modelo. Para ello usamos las anteriores "piezas" o "herramientas" en el HTML, las cuales tienen definida de antemano un sentido para el flujo de los datos.

En el siguiente diagrama puedes ver un resumen del flujo de la información disponible en Angular, junto con las piezas donde podemos encontrarlo y su sintaxis.



1. Las propiedades tienen un flujo desde el modelo a la vista. Una información disponible en el modelo se puede asignar como valor en un elemento del HTML mediante una propiedad, usando la notación corchetes. Por ej: `[propiedad]`
2. Las expresiones también viajan desde el modelo a la vista. La diferencia de las propiedades es que en este caso las usamos como contenido de un elemento y además que se expresan con dobles llaves. Por ej: `{{ expresión }}`
3. El binding (a dos sentidos, o doble binding) lo expresamos entre corchetes y paréntesis. En este caso la información fluye en ambos sentidos, desde el modelo a la vista y desde la vista al modelo. Por ej: `[(ngBind)]`
4. Los eventos no es que necesariamente hagan fluir un dato, pero sí se considera un flujo de

aplicación, en este caso de la vista al modelo, ya que se originan en la vista y generalmente sirven para ejecutar métodos que acabarán modificando cosas del modelo. Por ej: (evento)

Nota: Como ves, ahora existen diversas sintaxis para expresar cosas en las vistas. Quizás nos resulte extraño, pero enseguida nos familiarizaremos. La notación más rara es la que usamos para expresar un binding en dos direcciones `[(ngBing)]`, pero una manera sencilla de acordarnos de ella es con su denominación anglosajona "banana in a box". Los paréntesis parecen una banana, dentro de los corchetes, que parecen una caja.

Ejemplos de sintaxis utilizada en vistas de Angular 2

Realmente ya hemos visto ejemplos de buena parte de las piezas posibles a declarar en una vista. Si te fijas en el artículo anterior dedicado a la [Introducción a los componentes en Angular 2](#). Ahora les podemos dar nombres a cada uno de los elementos encontrados.

Propiedades:

Era el caso de la propiedad "hidden" que usamos para mostrar / ocultar determinados elementos. En este caso, hidden no es una propiedad estándar del HTML, sino que está generada por Angular 2 y disponible para aplicar tanto en etiquetas HTML comunes como en componentes personalizados.

```
<p [hidden]="!visible">Adiós</p>
```

También podríamos aplicar valores a atributos del HTML con datos que están en propiedades del modelo, aunque no soporta todos los atributos del HTML estándar. Por ejemplo, podríamos asignar una clase CSS (class) con lo que tuviésemos en una propiedad del modelo llamada "clase".

```
<div [class]="clase">Una clase marcada por el modelo</div>
```

O el enlace de un enlace podríamos también definirlo desde una variable del modelo con algo como esto:

```
<a [href]="enlace">Pulsa aquí</a>
```

En el código anterior se supone que el componente tendrá una propiedad llamada "enlace", que servirá para volcar su contenido en el href de la etiqueta A.

En general, el uso más corriente que haremos de las propiedades es personalizar el estado o comportamiento del componente, mediante datos que tengamos en el modelo. Veremos este caso más adelante cuando [analicemos con mayor detalle los componentes](#).

Lo que de momento debe quedar claro es que las propiedades van desde el modelo a la vista y, por tanto, si se modifica el valor de una propiedad en el modelo, también se modificará la vista. Pero, si dentro de la vista se modifica una propiedad no viajará al modelo automáticamente, pues el enlace es de una sola

dirección.

Obtendrás más detalles sobre el bindeo a propiedad en el artículo dedicado a [Property binding de Angular](#).

Expresiones:

Es el caso de la propiedad del modelo "title" que se vuelca como contenido de la página en el encabezamiento.

```
<h1>
  {{title}}
</h1>
```

Simplemente existe esa sustitución del valor de la propiedad, colocándose en el texto de la página. El enlace es de una única dirección, desde el modelo a la vista. En la vista tampoco habría posibilidad de modificar nada, porque es un simple texto.

El caso de propiedades del HTML con valores que vienen del modelo también podríamos implementarlo por medio de expresiones, tal como sigue.

```
<a href="{{enlace}}">Clic aqui</a>
```

Nota: Las dos alternativas (usar expresiones con las llaves o los corchetes para propiedades, como hemos visto para el ejemplo de un href de un enlace cuyo valor traes del modelo) funcionan exactamente igual, no obstante para algunos casos será mejor usar la sintaxis de propiedades en vez de la de expresiones, como es el caso del enlace. Algo que entenderemos mejor cuando lleguemos al sistema de rutas.

Explicamos más sobre las expresiones en el artículo sobre [String interpolation de Angular](#).

Eventos:

Esto también lo vimos en el [artículo anterior \(introducción a componentes\)](#), cuando asociamos un comportamiento al botón. Indicamos entre paréntesis el tipo de evento y como valor el código que se debe de ejecutar, o mejor, la función que se va a ejecutar para procesar el evento.

```
<button (click)="decirAdios()">Decir adiós</button>
```

Con respecto a Angular 1.x entenderás que ahora todas las directivas como ng-click desaparecen, dado que ahora los eventos solo los tienes que declarar con los paréntesis. Esto es interesante ya de entrada, porque nos permite definir de una única manera cualquier evento estándar del navegador, pero es más interesante todavía cuando comencemos a usar componentes personalizados, creados por nosotros, que podrán disparar también eventos personalizados. Capturar esos eventos personalizados será tan fácil como capturar

los eventos estándar del HTML.

Doble binding:

Para este último caso no hemos visto todavía una implementación de ejemplo, pero lo vamos a conseguir muy fácilmente. Como se dijo, usamos la notación "banana in a box" para producir este comportamiento de binding en dos direcciones. Los datos viajan de la vista al modelo y del modelo a la vista.

```
<p>
  ¿Cómo te llamas? <input type="text" [(ngModel)]="quien">
</p>
```

En este caso, desde el HTML estaríamos creando una propiedad dentro del modelo. Es decir, aunque no declaremos la propiedad "quien" en el Javascript, por el simple hecho de usarla en la estructura de binding va a producir que se declare automáticamente y se inicialice con lo que haya escrito en el campo de texto. Si la declaramos, o la inicializamos desde la clase que hace las veces de controlador, tanto mejor, pero no será necesario.

ACTUALIZADO: Ten muy en cuenta que para poder usar ngModel necesitas importar un módulo adicional, disponible mediante el propio Angular, pero que no viene cargado por defecto. Esto ocurre desde Angular 4. Ahora, cuando quieras usar la directiva ngModel tendrás que hacer el import de FormsModule. En este momento del manual es un poco pronto para explicar todos los detalles de esta importación. Lo tendrás todo más claro más adelante, cuando [hablemos específicamente de los módulos](#) y sobre todo en el artículo en el que [explicamos todas las posibilidades de la directiva ngModel](#).

Nota: La notación "banana in a box" tiene una explicación y es que usa tanto el flujo de datos desde el modelo a la vista, que conseguimos con los corchetes para las propiedades, como el flujo desde la vista al modelo que conseguimos con los paréntesis para los eventos.

Para quien conozca Angular 1.x, ngModel funciona exactamente igual que la antigua directiva. En resumen, le asignamos el nombre de una propiedad en el modelo, en este caso "quien", con la que se va a conocer ese dato. A partir de entonces lo que haya escrito en el campo de texto viajará de la vista al modelo y si cambia en el modelo también se actualizará la vista, produciendo el binding en las dos direcciones.

Si quisiéramos visualizar ese dato en algún otro de la vista, por ejemplo en un párrafo, usaríamos una expresión. Por ejemplo:

```
<p>
  Hola {{quien}}
</p>
```

Conclusión

Con lo que hemos aprendido hasta aquí tenemos una base inicial con la que comenzar a usar Angular y

realizar pequeños ejemplos sin perdernos demasiado. Podemos decir que hemos dado nuestro primer paso en el aprendizaje. Te recomendamos parar un instante la lectura del [Manual de Angular](#) para experimentar un poco por tu cuenta, cambiando el código en la vista y en el componente raíz de la aplicación, para afianzar este conocimiento

En los próximos artículos del manual vamos a abordar con mayor detalle el modelo de componentes, construyendo nuevos componentes en nuestro proyecto y practicando con todo lo visto hasta ahora. Nuestro siguiente paso es explicar con mayor detalle [cómo es la arquitectura de componentes](#), ya que es un concepto importante en el desarrollo con Angular.

Este artículo es obra de Miguel Ángel Álvarez
Fue publicado por primera vez en 19/ 07/ 2016
Disponible online en <http://desarrolloweb.com/articulos/sintaxisvistasangular2.html>

Los componentes en Angular

Abordamos el desarrollo basado en componentes con todo detalle. Es la pieza más importante de Angular que nos permitirá no solo estructurar una aplicación de una manera ordenada, sino encapsular funcionalidad y facilitar una arquitectura avanzada y de fácil mantenimiento de los proyectos Javascript con la nueva versión de este framework.

El concepto de los componentes en Angular y su arquitectura

Qué es un componente para Angular y cómo usaremos los componentes para realizar la arquitectura de una aplicación.

En este artículo del [Manual de Angular 2](#) queremos abordar el concepto del componente desde un punto de vista teórico, sin entrar a ver cómo se construyen en Angular 2. Esa parte práctica la dejaremos para el próximo artículo, aunque cabe recordar, para los que se impacientan por ver código, que ya dimos una [zambullida en el código de los componentes](#) cuando comenzamos con el "hola mundo".

El objetivo es entender mejor cuál es la arquitectura promovida por Angular 2 para el desarrollo de aplicaciones y qué papel específico desempeñan los componentes. Es importante porque toda aplicación Angular 2 se desarrolla en base a componentes y porque es algo relativamente nuevo en el framework.



Árbol de componentes

Una aplicación Angular 2 se desarrolla a base de crear componentes. Generalmente tendrás un árbol de componentes que forman tu aplicación y cada persona lo podrá organizar de su manera preferida. Siempre existirá un componente padre y a partir de ahí podrán colgar todas las ramas que sean necesarias para crear tu aplicación.

Esto no resultará nada extraño, pues si pensamos en una página web tenemos un mismo árbol de etiquetas, siendo BODY la raíz de la parte del contenido. La diferencia es que las etiquetas generalmente son para mostrar un contenido, mientras que los componentes no solo encapsulan un contenido, sino también una funcionalidad.

En nuestro árbol, como posible organización, podemos tener en un primer nivel los bloques principales de la pantalla de nuestra aplicación.

- Una barra de herramientas, con interfaces para acciones principales (lo que podría ser una barra de navegación, menús, botonera, etc.).
- Una parte principal, donde se desplegarán las diferentes "pantallas" de la aplicación.
- Un área de logueo de usuarios.
- Etc.

Nota: Obviamente, ese primer nivel de componentes principales lo dictará el propio proyecto y podrá cambiar, pero lo anterior nos sirve para hacernos una idea.

Luego, cada uno de los componentes principales se podrá subdividir, si se desea, en nuevos árboles de componentes.

- En la barra de herramientas principal podríamos tener un componente por cada herramienta.
- En el área principal podríamos tener un componente para cada "pantalla" de la aplicación o "vista". A su vez, dentro de cada "vista" o "pantalla" podríamos tener otra serie de componentes que implementen diversas funcionalidades.
- Etc.

Los niveles del árbol serán los que cada aplicación mande, atendiendo a su complejidad, y cada desarrollador estime necesario, en función de su experiencia o preferencias de trabajo. A medida que componetizamos conseguimos dividir el código de la aplicación en piezas menores, con menor complejidad, lo que seguramente sea beneficioso.

Si llegamos a un extremo, y nos pasamos en nuestra ansia de componetizar, quizás obtengamos el efecto contrario. Es decir, acabemos agregando complejidad innecesaria a la aplicación, puesto que existe un coste de tiempo de trabajo y recursos de procesamiento para posibilitar el flujo de comunicación entre componentes.

Componentes Vs directivas

En Angular 2 perdura el concepto de directiva. Pero ahora tenemos componentes y la realidad es que ambos artefactos se podrían aprovechar para usos similares. La clave en este caso es que los componentes son piezas de negocio, mientras que las directivas se suelen usar para presentación y problemas estructurales.

Puedes pensar en un componente como un contenedor donde solucionas una necesidad de tu aplicación. Una interfaz para interacción, un listado de datos, un formulario, etc.

Para ser exactos, en la documentación de Angular 2 nos indican que un componente es un tipo de directiva. Existen tres tipos de directivas:

Componentes: Un componente es una directiva con un template. Habrá muchas en tu aplicación y resuelven necesidades del negocio. Directivas de atributos: Cambian la apariencia o comportamiento de

un `element`. Por ejemplo tenemos `ngClass`, que nos permite colocar una o más clases de CSS (atributo `class`) en un elemento. Directivas estructurales: Son las que realizan cambios en el DOM del documento, añadiendo, manipulando o quitando elementos. Por ejemplo `ngFor`, que nos sirve para hacer una repetición (similar al `ngRepeat` de Angular 1.x), o `ngIf` que añade o remueve elementos del DOM con respecto a una expresión condicional.

Por tanto, a diferencia de otras librerías como Polymer, donde todo se resuelve mediante componentes, hay que tener en cuenta qué casos de uso son los adecuados para resolver con un componente.

Las partes de un componente

Aunque también hemos analizado anteriormente, cuando [repasamos la aplicación básica de Angular 2](#) generada con [Angular CLI](#), cuáles son las partes fundamentales de un componente, vamos a volver a este punto para poder ver sus piezas de manera global.

Un componente está compuesto por tres partes fundamentales:

- Un `template`
- Una `clase`
- Una función decoradora

Las dos primeras partes corresponden con capas de lo que conocemos como MVC. El `template` será lo que se conoce como `vista` y se escribe en HTML y lo que correspondería con el controlador se escribe en Javascript por medio de una `clase` (de programación orientada a objetos).

Por su parte, tenemos el decorador, que es una especie de registro del componente y que hace de "pegamento" entre el Javascript y el HTML.

Todas estas partes son las que vamos a analizar en los próximos artículos con mayor detalle, comenzando por los [decoradores](#), que introduciremos en el próximo artículo.

Este artículo es obra de Alberto Basalo
Fue publicado por primera vez en 01/ 08/ 2016
Disponible online en <http://desarrolloweb.com/articulos/concepto-teorico-componenteangular2.html>

Decorador de componentes en Angular 2

Qué es un decorador de componentes, qué función tiene y cómo se implementa en un componente básico de Angular 2.

Ahora, una de las funciones básicas que vas a tener que realizar en todo desarrollo con Angular es la decoración de componentes. En sí, no es más que una declaración de cómo será un componente y las diversas piezas de las que consiste.

En el artículo de [introducción a los componentes](#) explicamos solo una de las partes que tiene el archivo `.ts` con el código Javascript / TypeScript principal de un componente. Lo que vimos hasta ahora era la `clase`

que, decíamos, hacía las veces de controlador, que se exporta hacia afuera para que el flujo principal de ejecución de una aplicación sea capaz de conocer al componente. Además, contiene lo que se llama una función decoradora que conoceremos a continuación.



Qué es un decorador

Un decorador es una herramienta que tendremos a nuestra disposición en Javascript en un futuro próximo. Es una de las propuestas para formar parte del estándar ECMAScript 2016, conocido también como ES7. Sin embargo, ya están disponibles en TypeScript, por lo que podemos comenzar a usarlos ya en Angular.

Básicamente es una implementación de un patrón de diseño de software que en sí sirve para extender una función mediante otra función, pero sin tocar aquella original, que se está extendiendo. El decorador recibe una función como argumento (aquella que se quiere decorar) y devuelve esa función con alguna funcionalidad adicional.

Las funciones decoradoras comienzan por una "@" y a continuación tienen un nombre. Ese nombre es el de aquello que queramos decorar, que ya tiene que existir previamente. Podríamos decorar una función, una propiedad de una clase, una clase, etc.

Mira la primera línea del código del archivo .ts de tu componente principal.

```
import { Component } from '@angular/core';
```

Ese import nos está trayendo la clase Component. En la siguiente línea se decora a continuación, con el correspondiente "decorator". No es nuestro objetivo hablar sobre el patrón decorator en sí, ni ver las posibilidades de esta construcción que seguramente tendremos en el futuro ES7, así que vamos a centrarnos en lo que conseguimos hacer con Angular 2 mediante estos decoradores.

Nota: Uno de los motivos por los que Angular 2 ha tomado TypeScript como lenguaje es justamente por permitir usar decoradores. Con TypeScript podemos realizar la decoración de código de ES7 ya mismo, lo que facilita la decoración del código.

Qué información se agrega por medio del decorador

Angular 2 usa los decoradores para registrar un componente, añadiendo información para que éste sea reconocido por otras partes de la aplicación. La forma de un decorador es la siguiente:

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

Como apreciarás, en el decorador estamos agregando diversas propiedades específicas del componente. Esa información en este caso concreto se conoce como "anotación" y lo que le entregamos son unos "metadatos" (metadata) que no hace más que describir al componente que se está creando. En este caso son los siguientes:

- **selector:** este es el nombre de la etiqueta nueva que crearemos cuando se procese el componente. Es la etiqueta que usarás cuando quieras colocar el componente en cualquier lugar del HTML.
- **templateUrl:** es el nombre del archivo .html con el contenido del componente, en otras palabras, el que tiene el código de la vista.
- **styleUrls:** es un array con todas las hojas de estilos CSS que deben procesarse como estilo local para este componente. Como ves, podríamos tener una única declaración de estilos, o varias si lo consideramos necesario.

Nota: Ese código de anotación o decoración del componente es generado por Angular CLI. Además, cuando creamos nuevos componentes usaremos el mismo Angular CLI para obtener el scaffolding (esqueleto) del cual partiremos. Por tanto, no hace falta que memorices la sintaxis para la decoración, porque te la darán hecha. En todo caso tendrás que modificarla si quieres cambiar el comportamiento del componente, los nombres de archivos del template (vista), hojas de estilo, etc.

De momento no necesitamos dar mucha más información sobre los decoradores. Es algo que debemos comenzar a usar para desarrollar componentes en Angular 2, pero no nos tiene que preocupar demasiado todavía, porque de momento no necesitaremos tocar mucho sobre ellos.

Con esto creemos que hemos detallado perfectamente el componente inicial de nuestra aplicación, el generado por [Angular CLI](#) al inicializar el proyecto. En el próximo artículo, por fin, podremos [crear nuestro primer componente propio](#).

Este artículo es obra de Miguel Ángel Álvarez
Fue publicado por primera vez en 09/ 08/ 2016
Disponible online en <http://desarrolloweb.com/articulos/decorador-componentes-angular2.html>

Crear un componente nuevo con Angular 2

En este artículo te vamos a enseñar a crear un nuevo componente con Angular CLI y luego a usarlo en tu aplicación Angular 2.

Después de nuestro repaso teórico a la figura de los [componentes en Angular](#) estamos en condiciones de

irnos al terreno de lo práctico y ver cómo se generan nuevos componentes en una aplicación.

Si abres el `index.html` que hay en la carpeta `"src"` verás como en el `body` existe un único componente, pero sin embargo, una aplicación de Angular tendrá decenas o cientos de ellos. Los componentes se pueden organizar de diversas maneras y veremos más adelante cuando lleguemos a los módulos, que los podemos crear en el módulo principal de la aplicación o en módulos específicos de una funcionalidad en concreto. De momento vamos a crear un componente en el módulo principal, pues es lo más sencillo para comenzar.

Nota: Además de módulos, recuerda que unos componentes usarán o se apoyarán en otros para resolver sus necesidades, creando una estructura de árbol. Todos los componentes que desarrollemos en adelante estarán de alguna manera dentro del componente raíz. Esto no es nuevo, puesto que ya se comentó en el [Manual de Angular](#), pero está bien recordarlo para que quede claro. Partiremos de esta situación agregando ahora nuevos componentes con los que podremos expandir la aplicación.

Pero, en realidad, nadie te obliga a tener un componente único como raíz. Podrías crear un componente y usarlo directamente en el `index.html` de tu aplicación. Haciendo esto convenientemente (pues tendrías que agregar ese componente al `bootstrap` del módulo principal para que funcione) no habría ningún problema por ello. Aunque debido a que ese `index.html` es código generado y generalmente no lo querremos tocar, será más recomendable crear los componentes debajo del componente raíz.



Creamos un componente a través de Angular CLI

Ya usamos [Angular CLI](#) para generar el código de inicio de nuestra aplicación Angular. Ahora vamos a usar esta herramienta de línea de comandos para generar el esqueleto de un componente. Desde la raíz del proyecto lanzamos el siguiente comando:

```
ng generate component nombre-del-componente
```

Nota: Existe un alias para la orden `"generate"` que puedes usar para escribir un poco menos. Sería simplemente escribir `"g"`. Sería algo como `ng g component nombre-del-componente`

Además de componentes, la orden `generate` te permite crear el esqueleto de otra serie de artefactos como son directivas, servicios, clases, etc.

Reconociendo los archivos del componente

Si observas ahora la carpeta "src/ app" encontrarás que se ha creado un directorio nuevo con el mismo nombre del componente que acabamos de crear. Dentro encuentras una serie de archivos que ya te deben de sonar porque los hemos analizado ya para el [componente inicial de las aplicaciones Angular](#).

Nos referimos a los archivos donde colocas el código (TypeScript) de registro del componente, el CSS para los estilos y el HTML para la vista, más el archivo spec.ts, que sirve para el testing. Tendrás en definitiva algo como puedes ver en la imagen.

```
└─ src
  └─ app
    └─ nombre-del-componente
      ├── nombre-del-componente.component.css
      ├── nombre-del-componente.component.html
      ├── nombre-del-componente.component.spec.ts
      └── nombre-del-componente.component.ts
```

Nota: En versiones tempranas de Angular (en esta ocasión nos referimos a Angular 2, al menos en sus primeras releases), existía un archivo llamado "index.ts", que hacía solo tiene un export y servía para importar el componente de manera algo más resumida. En index.ts, se exportaba el propio componente, por su nombre. Servía para que, cuando importas un componente desde otro lugar de tu aplicación, no tengas que referirte al archivo "nombre-del-componente.component.ts" con todas sus letras, sino simplemente a la carpeta donde se encuentra. Ahora en Angular (4) ese archivo no existe.

Componente declarado en el módulo principal

Como hemos dicho, este componente que acabamos de crear residirá en el módulo principal. En adelante cuando veamos módulos explicaremos cómo hacer que el componente se cree dentro de un módulo, pero por el momento esta es la situación. En dicho módulo principal, archivo app.module.ts, se tiene que declarar el componente que acabamos de crear.

Realmente, al crear el componente mediante el terminal, con los comandos del [Angular CLI](#), las modificaciones en el módulo principal enfocadas a la declaración de este nuevo componente ya están realizadas. No obstante es bueno que le echemos un vistazo para irnos familiarizando.

Si abres el archivo app.module.ts, tendrías que reconocer la declaración en estos bloques de código:

1.- El import del componente

Este import nos trae el código TypeScript del componente que acabas de crear. Fíjate la clase del componente y la ruta donde está el código importado.


```
import { NombreDelComponenteComponent } from './nombre-del-componente/nombre-del-componente.component';
```

2.- La declaración "declarations"

En el decorador del módulo principal, en el array de `declarations`, encontrarás nombrado el componente que acabas de crear.

```
@NgModule({  
  declarations: [  
    AppComponent,  
    NombreDelComponenteComponent  
  ],  
  
  [...]  
})
```

Nota: Anteriormente, versiones muy tempranas de Angular 2, se usaba `SystemJS` para la declaración del componente. Esto no aplica a las versiones actuales de Angular: En todos los lugares donde, en adelante, deseabas usar ese componente, con `SystemJS` estabas obligado a importarlo para que se conozca su código. Para ello hemos visto que solo se usaba el nombre del componente y no el archivo donde se escribió su clase. Insistimos, eso era antes, cuando para realizar todo lo que es la carga de módulos se utilizaba `SystemJS`. En aquella época existía un archivo llamado `system-config.ts` donde se administran todas las librerías que se importaban con `SystemJS`. En ellas encontrarías la declaración del nuevo componente que acabamos de generar.

Javascript de nuestro componente

El archivo `"nombre-del-componente.component.ts"` contiene el código Javascript del componente.

Nota: Apreciarás que deberíamos decir que contiene el "código TypeScript del componente", dado que en realidad a día de hoy Angular CLI solo tiene la opción de generar código TypeScript. No debe representar un gran problema para ti, porque realmente todo código Javascript es también código [TypeScript](#), al que se le agregan ciertas cosas, sobre todo para el control de tipos.

Debes reconocer ya diversas partes:

- Imports de todo aquello que necesitemos. En principio de la librería `@angular/ core`, pero luego veremos que aquí iremos colocando muchos otros imports a medida que vayamos necesitando código de más lugares.
- Decorador del componente, para su registro.
- Clase que hace las veces del controlador, que tengo que exportar.

En estas partes ya conocidas no entraremos de momento con más detalles, pues ya los hemos abordado

anteriormente en el [Manual de Angular](#). Ahora te pedimos simplemente echar un vistazo a la cabecera de la clase, en concreto a su "implements":

```
export class NombreDelComponenteComponent implements OnInit {
```

Ese implements es una interfaz, que no están disponibles todavía en Javascript, ni tan siquiera en ES6, pero que ya son posibles de usar gracias a TypeScript. Es simplemente como un contrato que dice que dentro de la clase del componente vamos a definir la función `ngOnInit()`. Sobre esa función no hablaremos mucho todavía, pero es un lugar donde podremos colocar código a ejecutar cuando se tenga la certeza que el componente ha sido inicializado ya.

Solo a modo de prueba, vamos a crear una propiedad llamada "dato" dentro de nuestro componente recién creado. El código nos quedará algo como esto:

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-nombre-del-componente',
  templateUrl: 'nombre-del-componente.component.html',
  styleUrls: ['nombre-del-componente.component.css']
})
export class NombreDelComponenteComponent implements OnInit {

  dato = "Creando componentes para DesarrolloWeb.com";

  constructor() {}

  ngOnInit() {
    console.log('componente inicializado...');
  }

}
```

HTML del componente

El componente que acabamos de crear tiene un HTML de prueba, ya escrito en el archivo "nombre-del-componente.component.html".

Podemos agregarle la expresión para que se vea en el componente la propiedad que hemos generado del lado de Javascript, en la clase del componente. Tendrías algo como esto:

```
<p>
  nombre-del-componente works!
</p>
<p>
  {{ dato }}
</p>
```

Puedes abrir ese HTML y colocar cualquier cosa que consideres, simplemente a modo de prueba, para comprobar que consigues ver ese texto ahora cuando usemos el componente.

Nota: No hemos entrado todavía, pero seguro que alguno ya se lo pregunta o quiere saberlo. El componente tiene un archivo CSS (aunque se pueden definir varios en el decorador del componente) y podemos editarlo para colocar cualquier declaración de estilos. Veremos más adelante, o podrás comprobar por ti mismo, que esos estilos CSS se aplican únicamente al componente donde estás trabajando y no a otros componentes de la aplicación.

Con esto hemos terminado de explicar todo lo relativo a la creación de un componente. El componente está ahí y estamos seguros que estarás ansioso por [usarlo en tu proyecto](#). Es algo que veremos ya mismo, en el próximo artículo del Manual de Angular 2.

Este artículo es obra de Alberto Basalo
Fue publicado por primera vez en 26/ 08/ 2016
Disponible online en <http://desarrolloweb.com/articulos/crear-componente-nuevo-angular2.html>

Usar un componente en Angular 2

Cómo se usan componentes creados por nosotros en Angular 2, una tarea que si bien es sencilla requiere de varios pasos.

En el pasado artículo realizamos todos los pasos para [crear un componente en Angular 2](#). Realmente vimos que la mayoría del código lo generas desde [Angular CLI](#), lo que acelera mucho el desarrollo y facilita nuestra labor.

Ahora, para terminar nuestra práctica, vamos a aprender a usar el componente que acabamos de crear. Es una tarea sencilla, pero debido a la arquitectura de Angular y el modo de trabajo que nos marca, es algo que tendremos que realizar en varios pasos:



1. Crear el HTML para usar el componente

En el lugar de la aplicación donde lo vayas a usar el componente, tienes que escribir el HTML necesario para que se muestre. El HTML no es más que la etiqueta del componente, que se ha definido en la función

decoradora, atributo "selector" ([como vimos al explicar los decoradores](#)).

```
<app-nombre-del-componente></app-nombre-del-componente>
```

Dado que estamos comenzando con Angular 2 y el anterior era el primer componente creado por nosotros mismos, solo lo podremos usar dentro del componente principal (aquel generado por Angular CLI al hacer construir el nuevo proyecto). Aunque puede resultar obvio, esa etiqueta la tienes que colocar dentro del template del componente principal.

El HTML (template) de este componente principal lo encontramos en el archivo "app.component.html". En ese archivo, la vista del componente principal, debemos colocar la etiqueta para permitir mostrar el componente recién creado.

Nota: Además, el componente se creó dentro del módulo principal. No habrá problema en usarlo dentro del componente raíz, que también se creó dentro del módulo principal (app.module.ts). Si lo quisieras usar en otros módulos o lo hubieras creado en otros módulos, a esta operativa habría que añadir algún paso extra. Todo eso lo veremos cuando nos pongamos a explicar los módulos.

El problema es que esa etiqueta no es conocida por el navegador. La solución la aporta Angular, y el código del componente desarrollado en el artículo anterior, dedicado a la [creación de tu primer componente](#). Sin embargo, para que este componente se conozca, debes importarlo convenientemente. Es lo que hacemos en los siguientes pasos.

2. Importar el código del componente

Como decíamos, para poder usar un componente se debe de conocer su código. Para ello tenemos que realizar los correspondientes import. La buena noticia es que Angular CLI ha hecho el trabajo de manera automática. A no ser que quieras usar este componente desde otros módulos, tarea que veremos más adelante cuando lleguemos a módulos, no necesitas realizar ningún import adicional.

No obstante, para familiarizarnos con el código, y por si tenemos que editarlo nosotros manualmente a posteriori, vamos a identificar en el módulo principal "app.module.ts" los puntos donde se ha importado el código de este componente recién creado.

```
import { NombreDelComponenteComponent } from './nombre-del-componente/nombre-del-componente.component';
```

Ese import indica que te quieres traer la clase del componente "NombreDelComponenteComponent" y después del "from" está la ruta desde donde te la traes.

Nota: Recuerda que no necesitas decirle la extensión del archivo donde está la clase NombreDelComponenteComponent. Se trata de un archivo .ts y generalmente los archivos TypeScript no requieren que indiques su extensión al importarlos.

3. Declarar que vas a usar este componente

El import permite conocer el código del componente, pero todavía no es suficiente para poder usarlo. Debemos añadirlo al array de "declarations" (tarea que también ha hecho Angular CLI de manera automática).

En el módulo principal, donde vas a usar el componente de momento, encuentras la función decoradora del módulo "@NgModule". En esa función debes declarar todos los componentes que este módulo está declarando, en el array "declarations".

```
declarations: [  
  AppComponent,  
  NombreDelComponenteComponent  
],
```

El decorador completo del módulo principal se vería parecido a esto:

```
@NgModule({  
  declarations: [  
    AppComponent,  
    NombreDelComponenteComponent  
  ],  
  imports: [  
    BrowserModule  
  ],  
  providers: [],  
  bootstrap: [AppComponent]  
})
```

Observa que el array "declarations" me permite declarar que voy a usar varios componentes. Simplemente separo sus nombres por comas. Lo que indico, como podrás apreciar, es el nombre de la clase del componente que pretendo usar. Esa clase es la que has importado con el correspondiente "import" del paso anterior (punto 2).

Y eso es todo! Al guardar los archivos se debería recargar de nuevo la aplicación en el navegador y deberías ver el HTML escrito para el componente que estás usando y que acabamos de crear.

Conclusión a la introducción de los componentes en Angular

Con este artículo hemos completado el ciclo de introducción a los componentes en el framework Javascript Angular. Hemos reconocido sus partes básicas, hemos creado un componente propio y lo hemos usado en nuestra aplicación.

El proceso puede parecer un tanto laborioso, pero afortunadamente Angular CLI te lo facilita bastante. A medida que se vaya repitiendo observaremos que no es tan complicado. La parte más aburrida de escribir de un componente, el esqueleto o scaffolding y los imports, ya te lo dan hecho gracias a Angular CLI. Así que nos queda simplemente hacer la parte del componente que corresponde a las necesidades de la aplicación.

Con los conocimientos que hemos ido proporcionando en los anteriores capítulos del [Manual de Angular 2](#), estamos seguros de que podrás colocar más código, tanto en el HTML como en la clase del componente para poner en práctica lo aprendido.

Este artículo es obra de Alberto Basalo
Fue publicado por primera vez en 31/ 08/ 2016
Disponible online en <http://desarrolloweb.com/articulos/user-componenteangular2.html>

Lo básico de los módulos en Angular

Vamos ahora a abordar las bases de otro de los actores principales de las aplicaciones desarrolladas con Angular: los módulos. Un módulo (module en inglés) es una reunión de componentes y otros artefactos como directivas o pipes. Los módulos nos sirven principalmente para organizar el código de las aplicaciones Angular y por tanto debemos aprender a utilizarlos bien.

Trabajar con módulos en Angular

Cómo crear módulos, agrupaciones de componentes, directivas o pipes, en el framework Javascript Angular. Cómo crear componentes dentro de un módulo y cómo usarlos en otros modules.

Un módulo es uno de los elementos principales con los que podemos organizar el código de las aplicaciones en Angular. No deben ser desconocidos hasta este momento del [Manual de Angular](#), puesto que nuestra aplicación básica ya disponía de uno.

Sin embargo, en lugar de colocar el código de todos los componentes, directivas o pipes en el mismo módulo principal, lo adecuado es desarrollar diferentes módulos y agrupar distintos elementos en unos u otros. El orden se realizará de una manera lógica, atendiendo a nuestras propias preferencias, el modelo de negocio o las preferencias del equipo de desarrollo.

En este artículo aprenderás a trabajar con módulos, realizando operativas básicas como crear módulos y colocar componentes en ellos.

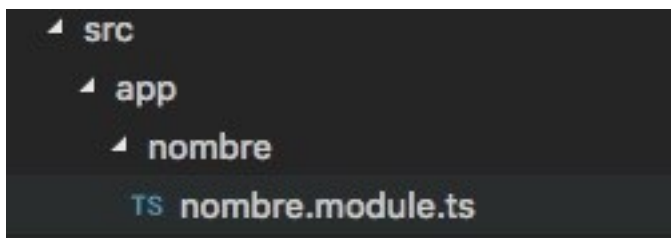


Crear un nuevo módulo

Para facilitar las tareas de creación de módulos nos apoyaremos en el [Angular CLI](#). El comando para generar ese módulo nuevo es "generate" y a continuación tenemos que indicar qué es lo que se quiere generar, en este caso "module", acabando con el nombre del módulo a crear.

```
ng generate module nombre
```

Una vez lanzado este comando en nuestro proyecto, dentro de la carpeta "src/ app" se crea un subdirectorio con el mismo nombre del módulo generado. Dentro encontraremos además el archivo con el código del módulo.



Nota: tanto da que en el comando nombres el módulo como "nombre" o "Nombre" (con la primera en mayúscula). El CLI aplica las convenciones de nombres más adecuadas y como los módulos son clases, internamente les coloca en el código la primera letra siempre en mayúscula. Ya los nombres de los directorios y archivos es otra cosa y no se recomienda usar mayúsculas, por lo que los nombrará con minúscula siempre.

Ahora, si abrimos el código del módulo generado "nombre.module.ts", encontraremos cómo se define un módulo en Angular. La parte más importante es, como ya viene siendo habitual en Angular, un decorador. El decorador de los módulos se llama @NgModule.

```
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
```

Nota: este es el código generado de un módulo con el CLI para Angular 4. En tu caso puede tener algunas diferencias, dependiendo de la versión de Angular con la que estés trabajando.

Como ves en el decorador, tienes de momento un par de arrays definidos:

- imports: con los imports que este módulo necesita
- declarations: con los componentes, u otros artefactos que este module construye.

Generar un componente dentro del módulo

Ahora que tenemos nuestro primer módulo propio, vamos a agregar algo en él. Básicamente comenzaremos por añadirle un componente, usando como siempre el Angular CLI.

Hasta ahora todos los componentes que habíamos creado habían sido generados dentro del módulo principal, pero si queremos podemos especificar otro módulo donde crearlos, mediante el comando:

```
ng generate component nombre/miComponente
```

Esto nos generará una carpeta dentro del módulo indicado, en la que colocará todos los archivos del componente recién creado.

```
└─ nombre
  └─ mi-componente
    # mi-componente.component.css
    <> mi-componente.component.html
    TS mi-componente.component.spec.ts
    TS mi-componente.component.ts
    TS nombre.module.ts
```

Nota: en este caso puedes observar como hemos colocado en el nombre del componente una mayúscula "miComponente", para separar palabras como en "camelCase". Por haberlo hecho así, Angular CLI ha nombrado el archivo separando las palabras por guiones "mi-componente.component.ts". Por su parte, podrás apreciar en el código que la clase del componente, se coloca con PascalCase, como mandan las guías de estilos para clases (class MiComponenteComponent).

Pero además, el comando del CLI también modificará el código del módulo, agregando automáticamente el import del componente y su referencia en el array "declarations". Ahora el código del módulo "nombre-modulo.module.ts" tendrá una forma como esta:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MiComponenteComponent } from './micomponente/micomponente.component';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    MiComponenteComponent
  ]
})
export class NombreModuloModule { }
```

Exportar del módulo hacia afuera

Más adelante, si queremos que este módulo exponga cosas hacia afuera, que se puedan llegar a utilizar desde otros módulos, tendremos que agregar una nueva información al decorador del módulo: el array de exports.

Vamos a suponer que el componente "MiComponenteComponent" queremos que se pueda usar desde otros módulos. Entonces debemos señalar el nombre de la clase del componente en el array de "exports". Con ello el decorador del module quedaría de esta manera.

```
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    MiComponenteComponent
  ],
  exports: [
    MiComponenteComponent
  ]
})
```

Usar el componente en otros módulos

El último punto que nos queda por ver es cómo usar el componente MiComponenteComponent desde otros módulos. Para ello vamos a modificar manualmente el módulo principal de la aplicación, de modo que pueda conocer el componente definido en el módulo nuevo que hemos creado en este artículo.

Para importar el componente realmente lo que vamos a importar es el módulo entero donde se ha colocado, ya que el propio módulo hace la definición de aquello que se quiere exportar en "exports". Requiere varios pasos

Hacer el import del módulo con la sentencia import de Javascript

Para que Javascript (o en este caso TypeScript) conozca el código del módulo, debemos importarlo primero. Esto no es algo de Angular, sino del propio lenguaje en particular.

```
import { NombreModule } from './nombre/nombre.module';
```

Declarar el import en el decorador del module principal

La importación de nuestro módulo se realiza en la declaración "imports" del módulo principal.

Este es el código del decorador del módulo principal.

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    NombreModule
  ],
```

```
providers: [],  
bootstrap: [AppComponent]  
}))
```

Te tienes que fijar en el array imports, que tiene el módulo que hemos creado nosotros mismos en este artículo "NombreModule".

Usar el componente en HTML

Finalmente ya solo nos queda usar el componente. Para ello vamos a colocar en el template del componente raíz el selector declarado en el componente creado.

Nota: recuerda que el selector del componente es el tag o etiqueta que se debe usar para poder usar un componente. Esto ya se detalló anteriormente en este manual. Si no lo recuerdas o quieres más información lee el artículo [Decorador de componentes en Angular 2](#).

Así que simplemente abrimos el archivo "app.component.ts" y colocamos la etiqueta de nuestro nuevo componente generado.

```
<app-mi-componente></app-mi-componente>
```

Eso es todo, si servimos nuestra aplicación deberíamos ver el mensaje del componente funcionando, que de manera predeterminada sería algo como "mi-componente works!".

Si no lo vemos, o no vemos nada, entonces nos tenemos que fijar el error que nos aparece, que podría estar visible en la pantalla del terminal donde hemos hecho el "ng serve", o en la consola de Javascript de tu navegador. Posiblemente allí te diga que tal componente no se conoce, con un mensaje como "parse errors: 'app-mi-componente' is not a known element...". Eso quiere decir que no has hecho el import correctamente en el decorador del módulo principal. O bien que te has olvidado de hacer el exports del componente a usar, en el módulo recién creado.

Este artículo es obra de Miguel Ángel Álvarez
Fue publicado por primera vez en 20/ 10/ 2017
Disponible online en <http://desarrolloweb.com/articulos/trabajar-modulos-angular.html>

Directivas esenciales de Angular

Vamos a conocer algunas directivas esenciales ofrecidas por el propio framework, para el desarrollo de las vistas en Angular.

Directiva ngClass en Angular 2

Estudio de la directiva ngClass de Angular 2, ejemplo práctico de un componente que usa esa directiva.

Después de varios artículos un tanto densos de en el [Manual de Angular 2](#) vamos a estudiar algunas cosas un poco más ligeras, que nos permitirán practicar con el framework sin añadir mucha complejidad a lo que ya conocemos. Para ello vamos a hacer una serie de pequeños artículos acerca de las directivas de Angular 2 más útiles en el día a día del desarrollo de aplicaciones.

Comenzamos con la directiva ngClass, que nos permite alterar las clases CSS que tienen los elementos de la página. Lo veremos en el marco del desarrollo de aplicaciones Angular con un ejemplo básico.

Si vienes de Angular 1.x verás que las directivas han perdido un poco de protagonismo en el desarrollo en Angular 2. Muchas de las antiguas directivas han desaparecido y su uso ha sido sustituido por otras herramientas diversas. Como norma ahora en Angular 2 las directivas se usarán para solucionar necesidades específicas de manipulación del DOM y otros temas estructurales.

Nota: Los componentes también pueden ser considerados como un tipo de directiva, aunque en este caso se usan para resolver problemas de negocio, como ya se introdujo en el artículo sobre las [características básicas de los componentes](#).



No todos los problemas de clases se necesitan resolver con ngClass

Lo primero es decir que la directiva ngClass no es necesaria en todos los casos. Las cosas más simples ni siquiera la necesitan. El atributo "class" de las etiquetas si lo pones entre corchetes funciona como

propiedad a la que le puedes asignar algo que tengas en tu modelo. Esto lo vimos en el [artículo sobre la sintaxis de las vistas](#).

```
<h1 [class]="claseTitular">Titular</h1>
```

En este caso, "claseTitular" es una variable del modelo, algo que me pasa el controlador que tendrás asociado a un componente.

```
export class PruebaComponent implements OnInit {  
  claseTitular: string = "class1";  
  
  cambiaEstado() {  
    this.claseTitular = "class2"  
  }  
  
  ngOnInit() {  
  }  
  
}
```

La class del H1 valdrá lo que haya en la variable claseTitular. Cuando alguien llame al método cambiaEstado() se modificaría el valor de esa variable y por tanto cambiaría la clase en el encabezamiento.

Si esto ya resuelve la mayoría de las necesidades que se nos ocurren ¿para qué sirve entonces ngClass?

Asignar clases CSS con ngClass

La directiva ngClass es necesaria para, de una manera cómoda asignar cualquier clase CSS entre un grupo de posibilidades. Puedes usar varios valores para expresar los grupos de clases aplicables. Es parecido a como funcionaba en Angular 1.x.

A esta directiva le indicamos como valor:

1. Un array con la lista de clases a aplicar. Ese array lo podemos especificar de manera literal en el HTML.

```
<p [ngClass]="['negativo', 'off']">Pueden aplicarse varias clases</p>
```

O por su puesto podría ser el nombre de una variable que tenemos en el modelo con el array de clases creado mediante Javascript.

```
<p [ngClass]="arrayClases">Pueden aplicarse varias clases</p>
```

Ese array lo podrías haber definido del lado de Javascript.

```
clases = ['positivo', 'si'];
```

2. Un objeto con propiedades y valores (lo que sería un literal de objeto Javascript). Cada nombre de propiedad es una posible clase CSS que se podría asignar al elemento y cada valor es una expresión que se evaluará condicionalmente para aplicar o no esa clase.

```
<li [ngClass]="{positivo: cantidad > 0, negativo: cantidad < 0, off: desactivado, on: !desactivado }">Línea</li>
```

Ejemplo de aplicación de ngClass

Vamos a hacer un sencillo ejemplo de un componente llamado "BotonSino" que simplemente muestra un mensaje "SI" o "NO". Al pulsar el botón cambia el estado. Cada estado se representa además con una clase que aplica un aspecto.

Nota: No vamos a explicar las partes del componente porque se vieron con detalle en los artículos anteriores del Manual de Angular 2. Consultar toda la parte de desarrollo de componentes para más información.

Nuestro HTML es el siguiente:

```
<p>
  <button
    [ngClass]="{si: estadoPositivo, no: !estadoPositivo}"
    (click)="cambiaEstado()"
    >{{texto}}</button>
</p>
```

La etiqueta button tiene un par de atributos que son los que hacen la magia. Entre corchetes se aplica la directiva "ngClass", con un valor de objeto. Entre paréntesis se aplica el evento "click", con la invocación de la función encargada de procesar la acción. Además, el texto del botón es algo que nos vendrá de la variable "texto".

Nuestro Javascript será el siguiente:

```
import { Component, OnInit } from '@angular/core';

@Component({
  moduleId: module.id,
  selector: 'app-boton-sino',
  templateUrl: 'boton-sino.component.html',
  styleUrls: ['boton-sino.component.css']
})
export class BotonSinoComponent implements OnInit {

  texto: string = "SI";
  estadoPositivo: boolean = true;
```

```
cambiaEstado() {  
    this.texto = (this.estadoPositivo) ? "NO" : "SI";  
    this.estadoPositivo = !this.estadoPositivo;  
}  
  
ngOnInit() {  
}  
  
}
```

Como sabes, este código TypeScript es la mayoría generado por Angular CLI. Lo que hemos hecho nosotros es lo que está dentro de la clase `BotonSinoComponent`.

En ella creamos las propiedades necesarias en la vista y el método que se encarga de procesar el cambio de estado.

Nuestro CSS:

Lógicamente, para que esto funcione necesitaremos declarar algunos estilos sencillos, al menos los de las clases que se usan en el HTML.

```
button {  
    padding: 15px;  
    font-size: 1.2em;  
    border-radius: 5px;  
    color: white;  
    font-weight: bold;  
    width: 70px;  
    height: 60px;  
}  
  
.si{  
    background-color: #6c5;  
}  
  
.no{  
    background-color: #933;  
}
```

Con esto es todo! Es un ejemplo muy sencillo que quizás para los que vienen de Angular 1.x resulte demasiado básico, pero seguro que lo agradecerán quienes estén comenzando con Angular en estos momentos. Más adelante lo complicaremos algo. Realmente la dificultad mayor puede ser [seguir los pasos para la creación del componente](#) y luego los [pasos para su utilización](#), pero eso ya lo hemos explicado anteriormente.

Directiva ngFor de Angular 2

Explicamos la directiva ngFor, o *ngFor, que nos permite repetir una serie de veces un bloque de HTML en aplicaciones Angular 2.

En este artículo vamos a conocer y practicar con una directiva de las más importantes en Angular 2, que es la directiva ngFor, capaz de hacer una repetición de elementos dentro de la página. Esta repetición nos permite recorrer una estructura de array y para cada uno de sus elementos replicar una cantidad de elementos en el DOM.

Para los que vengan de Angular 1 les sonará, puesto que es lo mismo que ya se conoce de ngRepeat, aunque cambian algunas cosillas sobre la sintaxis para la expresión del bucle, así como algunos mecanismos como la ordenación.

Uso básico de ngFor

Para ver un ejemplo de esta directiva en funcionamiento estamos obligados a crear de antemano un array con datos, que deben ser enviados a la vista, para que ya en el HTML se pueda realizar esa repetición. Como todo en Angular 2 se organiza mediante un componentes, vamos a crear un componente que contiene lo necesario para poder usar esta el ngFor.



Comenzamos con el código de la parte de TypeScript, que es donde tendremos que crear los datos que estarán disponibles en la vista.

```
import { Component, OnInit } from '@angular/core';

@Component({
  moduleId: module.id,
  selector: 'app-listado-preguntas',
  templateUrl: 'listado-preguntas.component.html',
  styleUrls: ['listado-preguntas.component.css']
})
export class ListadoPreguntasComponent implements OnInit {

  preguntas: string[] = [
    "¿España ganará la Euro 2016?",
    "¿Hará sol el día de mi boda?",
    "¿Estás aprendiendo Angular 2 en DesarrolloWeb?",
    "¿Has hecho ya algún curso en EscuelaIT?"
  ];
}
```

```
ngOnInit() {  
    
}
```

Este código nos debe de sonar, pues es el boilerplate de un componente (creado mediante Angular CLI) al que le hemos agregado la declaración de una propiedad de tipo array de strings.

Nota: Hemos asignado el valor de ese array de manera literal, pero lo normal sería que lo obtengas de alguna fuente como un servicio web, API, etc.

Luego, veamos el código HTML de este componente, que es donde colocamos la directiva ngFor.

```
<p *ngFor="let pregunta of preguntas">  
  {{pregunta}}  
</p>
```

Es algo muy sencillo, simplemente tenemos un párrafo que se repetirá un número de veces, una por cada elemento del array de preguntas. En este caso es un párrafo simple, pero si dentro de él tuviéramos más elementos, también se repetirían. Lo que tenemos que analizar con detalle es el uso de la directiva, aunque creemos que se auto-explica perfectamente.

```
*ngFor="let pregunta of preguntas"
```

Lo primero que verás es un símbolo asterisco (*) que quizás parezca un poco extraño. No es más que "azúcar sintáctico" para recordarnos que estas directivas (las comenzadas por el asterisco) afectan al DOM, produciendo la inserción, manipulación o borrado de elementos del mismo.

Nota: En las explicaciones que encontrarás en la documentación de Angular 2 sobre el origen del asterisco en el nombre de la directiva nos mencionan detalles acerca de su implementación a bajo nivel, indicando que para ello se basan en el tag [TEMPLATE](#), uno de las especificaciones nativas de Web Components.

Como valor de la directiva verás que se declara de manera interna para este bucle una variable "pregunta", que tomará como valor cada uno de los valores del array en cada una de sus repeticiones.

Nota: "let" es una forma de declarar variables en Javascript ES6. Quiere decir que aquella variable sólo tendrá validez dentro del bloque donde se declara. En este caso "de manera interna" nos referimos a que "pregunta" solo tendrá validez en la etiqueta que tiene el ngFor y cualquiera de sus los elementos hijo.

Recorrido a arrays con objetos con ngFor

Generalmente ngFor lo usarás para recorrer arrays que seguramente tendrán como valor en cada una de sus casillas un objeto. Esto no cambia mucho con respecto a lo que ya hemos visto en este artículo, pero sí es una bonita oportunidad de aprender algo nuevo con TypeScript.

De momento veamos las cosas sin TypeScript para ir progresivamente. Así sería cómo quedaría la declaración de nuestro array, al que todavía no indicaremos el tipo para no liarnos.

```
preguntasObj = [  
  {  
    pregunta: "¿España ganará la Euro 2016?",  
    si: 22,  
    no: 95  
  },  
  {  
    pregunta: "¿Estás aprendiendo Angular 2 en DesarrolloWeb??",  
    si: 262,  
    no: 3  
  },  
  {  
    pregunta: "¿Has hecho ya algún curso en EscuelaIT??",  
    si: 1026,  
    no: 1  
  }  
]
```

Como ves, lo que antes era un array de strings simples ha pasado a ser un array de objetos. Cada uno de los objetos nos describen tanto la pregunta en sí como las respuestas positivas y negativas que se han recibido hasta el momento.

Ahora, al usarlo en la vista, el HTML, podemos mostrar todos los datos de cada objeto, con un código que podría ser parecido a este:

```
<p *ngFor="let objPregunta of preguntasObj">  
  {{objPregunta.pregunta}}:  
  <br>  
  <span class="si">Si {{objPregunta.si}}</span> /  
  <span class="no">No {{objPregunta.no}}</span>  
</p>
```

Como ves en este código, dentro del párrafo tengo acceso a la pregunta, que al ser un objeto, contiene diversas propiedades que uso para mostrar los datos completos de cada ítem.

Modificación implementando Interfaces TypeScript

Como has visto, no existe mucha diferencia con respecto a lo que teníamos, pero ahora vamos a darle un uso a TypeScript que nos permitirá experimentar algo que nos aporta el lenguaje: interfaces.

En este caso vamos a usar las interfaces simplemente para definir un tipo de datos para las preguntas, un esquema para nuestros objetos pregunta. En este caso solo lo vamos a usar para que, a la hora de escribir código, el editor nos pueda ayudar indicando errores en caso que la interfaz no se cumpla. Así, a la hora de escribir código podremos estar seguros que todas las preguntas con las que trabajemos tengan los datos que son necesarios para la aplicación.

Nota: Las interfaces que se sabe son mecanismos para solventar las carencias de herencia múltiple, en este caso las vamos a usar como una simple definición de tipos.

Algo tan sencillo como esto:

```
interface PreguntasInterface {  
  pregunta: string;  
  si: number;  
  no: number;  
}
```

Permite a TypeScript conocer el esquema de un objeto pregunta. Ahora, apoyándonos en esa interfaz podrás declarar tu array de preguntas de esta manera.

```
preguntasObj: PreguntasInterface[] = [  
  {  
    pregunta: "¿Te gusta usar interfaces?",  
    si: 72,  
    no: 6  
  }  
]
```

Ahora estamos indicando el tipo de los elementos del array, diciéndole que debe concordar con lo definido en la interfaz. ¿Te gusta? Quizás ahora no aprecies mucha diferencia, pero esto se puede usar para varias cosas, significando una ayuda en la etapa de desarrollo, y sin afectar al rendimiento de la aplicación, puesto que las interfaces en TypeScript una vez transpilado el código no generan código alguno en Javascript.

Nota: Lo normal es que coloques el código de la interfaz en un archivo independiente y que hagas el correspondiente "import". Recordando que Angular CLI tiene un comando para generar interfaces que te puede resultar útil. De momento si lo deseas, a modo de prueba lo puedes colocar en el mismo archivo que el código TypeScript del componente.

Quizás para los que no estén acostumbrados a TypeScript sea difícil hacerse una idea exacta sobre cómo te ayudaría el editor de código por el simple hecho de usar esa interfaz. Para ilustrarlo hemos creado este vídeo en el que mostramos las ayudas contextuales cuando estamos desarrollando con Visual Studio Code.

Para ver este vídeo es necesario visitar el artículo original en:
<http://desarrolloweb.com/articulos/directiva-ngfor-angular2.html>

Hablaremos más sobre interfaces en otras ocasiones. Ahora el objetivo era simplemente ver una pequeña muestra de las utilidades que podría aportarnos.

Como habrás visto no es difícil entender esta directiva, pero ten en cuenta que hemos visto lo esencial sobre las repeticiones con ngFor. Hay mucho más que habría que comentar en un futuro, acerca de usos un poco más avanzados como podría ser la ordenación de los elementos del listado.

Este artículo es obra de Miguel Ángel Álvarez
Fue publicado por primera vez en 27/ 09/ 2016
Disponible online en <http://desarrolloweb.com/articulos/directiva-ngfor-angular2.html>

Directiva ngModel

Explicaciones sobre la directiva ngModel de Angular, con ejemplos de uso diversos, con binding de una y dos direcciones. Importar FormsModule para su funcionamiento en Angular 4.

La directiva ngModel es un viejo conocido para las personas que vienen de las versiones antiguas del framework, cuando se llamaba AngularJS. Quizás para ellos no requiera tantas explicaciones de concepto, aunque sí será importante explicar cómo usarla, porque han cambiado bastantes cosas.

De todos modos, tanto para desarrolladores experimentados como para los más novatos, vamos a repasar en este artículo del [Manual de Angular](#) los aspectos básicos de ngModel y ver algunos ejemplos sencillos de funcionamiento en componentes.



Qué es ngModel

Pensando en las personas que son nuevas en Angular, tendremos que comenzar aclarando qué es ngModel. Básicamente se trata de un enlace, entre algo que tienes en la definición del componente con un campo de formulario del template (vista) del componente.

Nota: "model" en ngModel viene de lo que se conoce como el modelo en el MVC. El modelo trabaja con los datos, así que podemos entender que el enlace creado con ngModel permite que desde la vista pueda usar un dato. Sin embargo, con expresiones ya se podía usar un dato, volcando su valor en la vista como {{ dato }}. La diferencia es que al usar ese dato en campos de formulario, debes aplicar el valor mediante la directiva ngModel.

Por ejemplo, tenemos un componente llamado "cuadrado", que declara una propiedad llamada "lado". La class del componente podría quedar así:

```
export class CuadradoComponent {  
  lado = 4;  
}
```

Si queremos que ese valor "lado" se vuelque dentro de un campo INPUT de formulario, tendríamos que usar algo como esto.

```
<input type="number" [ngModel]="lado">
```

Estamos usando la directiva ngModel como si fuera una propiedad del campo INPUT, asignándole el valor que tenemos declarado en el componente.

Dar soporte a la propiedad ngModel en el campo de formulario

Sin embargo, nuestro componente "cuadrado" no funcionaría todavía, porque Angular 4 en principio no reconoce ngModel como propiedad de un campo INPUT de formulario. Por ello, si ejecutas tu aplicación con el código tal como hemos hecho hasta ahora, obtendrás un mensaje de error como este: "Can't bind to 'ngModel' since it isn't a known property of 'input'."

La solución pasa por traernos esa propiedad, que está en el módulo "FormsModule".

Para ello tenemos que hacer la operativa tradicional de importar aquello que necesitamos. En este caso tendremos que importar FormsModule en el módulo donde vamos a crear aquel componente donde se quiera usar la directiva ngModule.

Por ejemplo, si nuestro componente "CuadradoComponent" está en el módulo "FigurasModule", este sería el código necesario para declarar el import de "FormsModule".

```
import { NgModule } from '@angular/core';  
import { CommonModule } from '@angular/common';  
import { CuadradoComponent } from './cuadrado/cuadrado.component';  
import { FormsModule } from '@angular/forms';  
  
@NgModule({  
  imports: [  
    CommonModule,
```

```
    FormsModule
  ],
  declarations: [CuadradoComponent],
  exports: [CuadradoComponent]
})
export class FigurasModule { }
```

Del código del módulo anterior, debes fijarte en dos cosas:

1.- El import de FormsModule, que viene de `@angular/ forms`

```
import { FormsModule } from '@angular/forms';
```

2.- La declaración en el array de imports del módulo en cuestión:

```
imports: [
  [...],
  FormsModule
],
```

Con esta infraestructura ya somos capaces de usar `ngModule` en las vistas de los componentes Angular.

Enlace de una o de dos direcciones con `ngModel`

Tal como hemos dejado el código hasta el momento en la vista, el input estaba asociado al valor de una propiedad del componente, sin embargo, era un enlace de una única dirección.

```
<input type="number" [ngModel]="lado">
```

Con esa sintaxis en la vista, lo que haya en la propiedad "lado" se escribía como valor del input, pero no encontramos el doble binding. Y aunque esto es algo que ya se trató en el artículo de [sintaxis de las vistas en Angular](#), queremos recordar que, para especificar el doble binding, se debe usar la sintaxis "banana in a box".

```
<input type="number" [(ngModel)]="lado">
```

Ahora, lo que se escriba en el campo INPUT también viajará hacia el modelo, actualizando el valor de la propiedad "lado" del componente.

Evento `ngModelChange`

Si lo deseas, también tienes disponible un evento llamado "ngModelChange" que se ejecuta cuando cambia el valor en la propiedad asociada a un `ngModel`, con el que podríamos conseguir un comportamiento idéntico al visto en el punto anterior del doble binding, pero sin usar el binding doble.

Tienes que trabajar con `ngModelChange` en conjunto con la directiva `ngModel`. Mediante `ngModel` asocias la propiedad que quieres asociar y entonces tendrás la posibilidad de asociar un manejador de evento a `ngModelChange`, cada vez que esa propiedad cambia.

Dentro del manejador de evento podemos además usar una variable llamada `$event`, en la que recibimos el nuevo valor escrito, que podríamos volcarla de nuevo a la propiedad por medio de una asignación, para conseguir el mismo efecto del binding en las dos direcciones. El código te quedaría como esto:

```
<input type="number" [ngModel]="lado" (ngModelChange)="lado = $event">
```

Nota: Esta sintaxis no aporta ninguna ventaja en términos de rendimiento, por lo que en principio no se suele usar mucho. Generalmente vamos a preferir usar la sintaxis del binding de dos direcciones `[(ngModel)]`, por ser más concisa. De todos modos, podría ser interesante disponer de `ngModelChange` en el caso que, cuando cambiase el modelo, necesitas realizar otras acciones, adicionales a la simple asignación de un nuevo valor en la propiedad del componente.

Ejemplo realizado en este artículo

Ahora para la referencia, voy a dejar el código realizado para ilustrar el comportamiento y uso de la directiva `ngModel`. Estará compuesto por varios archivos.

Vista del componente:

Comenzamos con la vista del componente, archivo `cuadrado.component.html`, que es lo más importante en este caso, ya que es el lugar donde hemos usado `ngModel`.

He creado varias versiones de enlace al `input`, como puedes ver a continuación.

```
<p>
  Tamaño del lado {{lado}}
</p>
<p>
  1 way binding <input type="number" [ngModel]="lado">
</p>
<p>
  2 way binding: <input type="number" [(ngModel)]="lado">
</p>
<p>
  Evento ngModelChange: <input type="number" [ngModel]="lado" (ngModelChange)="cambiaLado($event)">
</p>
```

Otra cosa que apreciarás en la vista es que el evento `ngModelChange` no tiene escrito el código del manejador en la propia vista, por considerarlo un antipatrón. Es más interesante que el código se coloque dentro de un método del componente. En este caso tendrás que enviarle al método el valor `$event`, para que lo puedas usar allí.

Código TypeScript del componente:

El código del componente estará en el archivo `cuadrado.component.ts` y no tiene ninguna complicación en especial.

```
import { Component } from '@angular/core';

@Component({
  selector: 'dw-cuadrado',
  templateUrl: './cuadrado.component.html',
  styleUrls: ['./cuadrado.component.css']
})
export class CuadradoComponent {
  lado = 1;

  cambiaLado(valor) {
    this.lado = valor;
  }
}
```

Módulo donde se crea el componente:

Por último recuerda que es esencial que importes en el módulo donde estés creando este componente el propio módulo del core de Angular donde se encuentra la directiva `ngModule`. Esto lo hemos descrito en un bloque anterior, pero volvemos a colocar aquí el código fuente:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { CuadradoComponent } from './cuadrado/cuadrado.component';
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [
    CommonModule,
    FormsModule
  ],
  declarations: [CuadradoComponent],
  exports: [CuadradoComponent]
})
export class FigurasModule { }
```

Por supuesto, para poder usar este componente "CuadradoComponent" en otro módulo tendrás que hacer el correspondiente import, pero esto es algo que ya se trató en el artículo de los [módulos en Angular](#).

Nota: Además de la aplicación de `ngModel` que hemos conocido en este artículo, relacionada directamente con el sistema de binding de Angular, `ngModel` también sirve para definir qué campos de formulario deben ser tratados al generarse los objetos `ngForm`. Estos objetos se crean automáticamente y Angular los pone a disposición para poder controlar el formulario de una manera muy precisa. Ese uso

lo veremos más adelante cuando hablemos en detalle de los formularios de Angular.

Este artículo es obra de Miguel Ángel Álvarez

Fue publicado por primera vez en 26/ 10/ 2017

Disponible online en <http://desarrolloweb.com/articulos/directiva-ng-model-angular.html>

Binding en Angular al detalle

En los siguientes artículos vamos a explorar detenidamente el sistema de data-binding creado con Angular, tratando con profundidad algunos temas ya usados dentro de este manual, así como abordando nuevos temas importantes relacionados con el bindeo de datos.

Interpolación `{{}}` en Angular al detalle

Todo lo que tienes que saber sobre el binding por interpolación de strings en Angular, generada con la sintaxis de las dobles llaves `{{}}`.

Con este artículo comenzamos una serie de entregas del [Manual de Angular](#) en las que vamos a abordar distintos aspectos del binding en este framework. Son importantes, porque a pesar de ser cosas básicas muchas veces las dejamos pasar y ese desconocimiento acaba creando confusiones más adelante.

De todos modos, no vamos a explicar todo desde cero, ya que las bases del data-binding ya las hemos conocido en artículos anteriores del manual. Concretamente es interesante que hayas leído el artículo sobre la [sintaxis para las vistas de componentes](#).

En este artículo vamos a tratar muchas cosas sobre la interpolación en Angular que quizás hemos dado por sentadas en artículos anteriores y que merece la pena tratar de manera detallada y clara.



Qué es la interpolación de strings

La interpolación de cadenas, también conocida en inglés como "string interpolation", o simplemente como interpolación, es un mecanismo de Angular de sustitución de una expresión por un valor de cadena en un template.

Cuando Angular ve en un template algo escrito entre dobles llaves `{{}}` lo evalúa y lo trata de convertir en una cadena, para luego volcarlo en el template.

```
<p>Esto es un caso de interpolación de {{algunaCadena}}</p>
```

Eso quiere decir a Angular que debe sustituir una propiedad del componente llamada "algunaCadena" y colocar su valor tal cual en el template.

Si "algunaCadena" es una cadena, simplemente se escribirá su valor en el template, pero si eso no era una cadena tratará de colocarlo de manera que lo fuera. Por ejemplo, si tuviera un valor numérico colocará el número tal cual en el template.

Puedes usar también la interpolación como valor de propiedades de elementos HTML, como es el siguiente caso.

```

```

En este caso se colocará el valor de la propiedad urlImagen como src para el elemento IMG.

La interpolación es dinámica. Quiere decir que, si cambia el valor de la propiedad del componente, Angular se dará el trabajo de cambiar todos los lugares donde se está haciendo uso de esa propiedad, lo que cambiaría el texto que hay escrito del párrafo anterior, o cambiaría la imagen que se está visualizando en el elemento IMG anterior.

Expresiones, entre las dobles llaves

Lo que se coloca entre las dobles llaves son llamadas expresiones. Podemos crear expresiones simples y complejas y Angular se dará el trabajo de evaluarlas antes de volcar el resultado dentro del template.

Lo que es importante es que, aquella evaluación de la expresión, debe de ser convertida en una cadena antes de volcarse en un template. Es decir, cualquier expresión al final de cuentas se convertirá en una cadena y eso es lo que se colocará en la vista del componente.

Una expresión puede ser algo tan simple como una propiedad del componente, como las usadas anteriormente en este artículo.

```
{{algunaCadena}}
```

Esa sencilla expresión se evalúa al valor que tenga la propiedad algunaCadena y se vuelca en el template.

También podemos ver expresiones con operaciones matemáticas.

```
{{ 1+ 1 }}
```

Expresiones con operador lógico de negación:

```
{{ ! valorBoleano }}
```

Incluso expresiones cuyo valor es calculado por un método del componente.

```
{{ metodoComponente() }}
```

Lo que devuelva ese método del componente es lo que se colocará en el template. Ese método se volverá a ejecutar cada vez que el estado del componente cambie, es decir, cada vez que cambie una de sus propiedades, produciendo siempre una salida actualizada.

Nota: aunque vamos a insistir sobre este punto y se entenderá mejor si sigues la lectura del artículo, hay que decir que ese método tiene que limitarse a producir salida y además ser sencillo y rápido de ejecutar. Esto evitará afectar negativamente al rendimiento de las aplicaciones. El motivo es sencillo: si durante la ejecución de la aplicación se modifican las propiedades del componente, Angular volverá a ejecutar ese método, actualizando la salida convenientemente. Eso se hará ante cualquier pequeño cambio en el componente, y no solo ante cambios en las propiedades con las que trabaje el método, incluso se producirá solamente por haber ocurrido un evento susceptible de modificar el estado. Si pones en el método código que tarde en ejecutarse, Al producirse muchas invocaciones repetidas a este método, se multiplicará el coste de tiempo de ejecución, produciendo que la aplicación caiga en rendimiento y afectando negativamente a la experiencia de usuario.

Los efectos laterales están prohibidos en las expresiones

La interpolación en Angular es un enlace (binding) de una única dirección. Cuando cambian los valores en el componente viajan hacia el template, produciendo la actualización de la vista.

La interpolación no debería producir cambios en la otra dirección, es decir, modificar algo en la vista afectando al estado del componente. Dicho de otro modo, las expresiones nunca deben contener sentencias que puedan producir efectos laterales. Es decir, código que pueda afectar a cambios en las propiedades de componentes.

Ten en cuenta que las expresiones tienen como utilidad devolver una cadena para volcarla en el template del componente, por lo que no deberías colocar lógica de negocio en ellas, manipulando variables o propiedades de estado del componente o la aplicación.

Por este detalle, algunos operadores están prohibidos en las expresiones, por ejemplo los de asignación, incrementos, decrementos, etc. Por ejemplo, nunca hagas algo como esto:

```
{{ valorBoleano = false }}
```

Si Angular observa una expresión con operaciones capaces de realizar efectos laterales, te devolverá un error, como el que ves en la siguiente imagen:

```
✖ ▶ Uncaught Error: Template parse errors: compiler.es5.js:1694  
Parser Error: Bindings cannot contain assignments at column  
17 in [  
  
  {{ valorBoleano = false }}]
```

Por este mismo motivo, cuando dentro de una expresión invocas a un método, no deberías modificar en el código de ese método datos del componente capaces de producir efectos laterales, sino simplemente limitarte a producir una salida.

```
{{ metodoProduceEfectosLaterales() }}
```

En el caso que `metodoProduceEfectosLaterales()` tenga código que produzca cambios el el estado del componente Angular no te lo va a advertir y va a ejecutar el método sin producir ningún mensaje de error, pero aun así no lo deberías hacer. No es solo por ceñirse a las buenas prácticas, sino porque Angular no tendrá en cuenta aquel cambio en el estado del componente para desencadenar otras operaciones necesarias, lo que puede producir que tus templates no muestran los cambios en propiedades u otros efectos no deseables.

Otras consideraciones en las expresiones

Otras cosas que está bien saber cuando se usan expresiones son las siguientes:

- Las expresiones tienen como contexto al componente del template que se está desarrollando. Pero no pueden acceder al contexto global, como variables globales u objetos como `document` o `window`. Tampoco a cosas del navegador como `console.log()`.
- No se pueden escribir estructuras de control, como sentencias `if`, `for`, etc.
- La ejecución de una expresión debe ser directa. Por ejemplo no podrás hacer que se consulte un API REST en una expresión.
- Mantén la simplicidad. Cosas como usar una negación en la expresión es correcto, pero si lo que tienes que colocar es cierta lógica compleja, entonces conviene usar un método en el componente.
- Ofrecer siempre el mismo resultado con los mismos valores de entrada. Esto se conoce como expresiones idempotentes, aquellas que ejecutadas varias veces dan siempre el mismo resultado. Obviamente, si la entrada es distinta y los datos que se usan para calcular el valor de la expresión cambian, el resultado también cambiará, pero seguirá siendo idempotente si con el mismo juego de valores de entrada conseguimos siempre la misma salida. Esto es necesario para que los algoritmos de detección de cambios de Angular funcionen correctamente.

Este artículo es obra de Miguel Ángel Álvarez
Fue publicado por primera vez en 30/ 10/ 2017
Disponible online en <http://desarrolloweb.com/articulos/binding-interpacion-angular.html>

Binding a propiedades en Angular al detalle

Qué es el property binding, junto con una serie de detalles importantes sobre el bindeo a propiedades que como desarrollador Angular debes conocer.

Aunque ya hemos hablado del binding de propiedades, vamos a volver sobre este concepto pues hemos pasado por alto bastantes detalles en los que merece la pena detenerse con calma. Se trata de una característica disponible en el [desarrollo de templates de los componentes](#), que hemos visto aplicada a lo

largo varios ejemplos del [Manual de Angular](#), pero esperamos poder ofrecer algunos conocimientos extra que seguramente te vendrán bien.

Comenzaremos repasando de nuevo el concepto, para ofrecer luego diversas explicaciones adicionales sobre cómo funciona el property binding. Además resolveremos algunas dudas comunes de las personas que se inician en Angular.



Qué es el bindeo a propiedades

El binding a propiedades, o "property binding" en inglés, sirve para asignar un valor a una propiedad de un elemento de un template. Esa asignación podrá ser un valor literal, escrito tal cual en el template, pero generalmente se tratará de un valor obtenido a través de una propiedad del componente, de modo que si el estado del componente cambia, también cambie la propiedad del elemento asignada en el template.

Para que no quede lugar a dudas tenemos que definir exactamente qué es una propiedad. De hecho, en el párrafo anterior hemos usado la palabra "propiedad" para referirnos a dos cosas, lo que puede dar mayor pie a confusión.

Propiedades de componentes

Los componentes generados con Angular se implementan mediante una clase, de programación orientada a objetos. Las propiedades de esa clase, los datos que almacena el componente, son lo que llamamos "propiedades del componente".

Generalmente en este artículo cuando nos referimos a "property binding" no nos referimos específicamente a propiedades del componente, sino a propiedades expresadas en un template. Éstas pueden ser propiedades de elementos HTML (propiedades del DOM de esos elementos) o propiedades de componentes en general.

Propiedades expresadas en el template

En el HTML del componente, lo que llamamos el template o la vista, las propiedades son cualquier cosa a las que asignamos valores por medio de binding. Esto lo podemos ver bien con un ejemplo.

A continuación tienes elementos del HTML a los que definimos valores por medio de atributos.

```
  
<button disabled>Estoy desactivado</button>
```


A esos atributos de las etiquetas les asignamos valores, que sirven para inicializar los elementos HTML. Debes de saber de sobra cómo funcionan.

Tal como están, esa etiqueta IMG o ese botón tendrán siempre el mismo valor en sus propiedades "src" y "disabled". Sin embargo, si quisieras asignar un valor dinámico a uno de esos atributos, tomando algo definido mediante una propiedad del componente, tendrías que acudir al property binding. Se define el bindeo a propiedad mediante una sintaxis especial de Angular, asunto de este artículo.

```
<img [src]="rutaImagen">
<button [disabled]="estadoBoton">Estoy activado o desactivado</button>
```

En este caso es donde encontramos las propiedades genéricas a las que nos hacemos referencia, aquellas expresadas en el template con la notación de los corchetes de inicio y de cierre.

Gracias al binding de propiedades tenemos una imagen y un botón igualmente, pero en la imagen su src se calcula en función de la propiedad del componente llamada "rutaImagen". En el caso del botón, el estado activo o desactivado, está definido por la propiedad del componente "estadoBoton".

Lo importante que debemos entender es: cuando trabajas con propiedades del template, no estás asignando valores a atributos de HTML. De hecho, al ponerle los corchetes dejan de ser atributos del HTML para pasar a ser propiedades del template de Angular.

Variantes del property binding

Mediante los corchetes puedes asignar valores dinámicos a propiedades de un componente, propiedades de directivas, o en el caso de los elementos HTML nativos, estás asignando valores directamente al DOM.

Vamos a analizar más de cerca estos casos con varios ejemplos.

```
<img [src]="rutaImagen">
```

En este ejemplo estamos colocando un valor a una propiedad de un elemento nativo del HTML, que nos acepta un string.

Sin embargo no todas las propiedades aceptan cadenas, algunas aceptan booleanos, como era el caso del botón.

```
<button [disabled]="estadoBoton">Estoy activado o desactivado</button>
```

Este es un caso interesante, porque el elemento nativo del HTML funciona de modo diferente. Si tiene el atributo "disabled" está desactivado. Si ese atributo no aparece, entonces estará activado. Ya para definir el valor de la propiedad de Angular aparecerá siempre el atributo, pero en algunas ocasiones estará desactivado y en otras activado, dependiendo del valor de la variable booleana "estadoBoton".

También podemos encontrar binding a propiedades en una directiva, como se puede ver en el siguiente código.

```
<div [ngStyle]="objEstilos">DIV con estilos definidos por Angular</div>
```

En este caso, mediante la propiedad "objEstilos" se está definiendo dinámicamente el style de esta división. Si cambia el objeto cambiará la aplicación de los estilos al elemento.

Por último, tenemos asignación a propiedades personalizadas de nuestros propios componentes.

```
<mi-componente [propiedadPersonalizada]="valorAsignado"></mi-componente>
```

Este es un caso especial, que explicaremos con detalle cuando hablemos de las propiedades @Input de componentes. Básicamente nos permiten crear cualquier tipo de propiedad en componentes, que somos capaces de definir en el padre. En ese caso concreto, mediante la propiedad del componente padre "valorAsignado" estamos aplicando el valor del componente hijo en la propiedad "propiedadPersonalizada".

Binding de una dirección

Como se ha mencionado, la asignación de una propiedad es dinámica. Es decir, si cambia con el tiempo, también cambiará la asignación al elemento al que estamos bindeando.

Volvemos de nuevo al ejemplo:

```
<img [src]="rutaImagen">
```

Al crearse la imagen se tomará como valor de su "src" lo que haya en la propiedad del componente "rutaImagen". Si con el tiempo "rutaImagen" cambia de valor, el valor viajará también a la imagen, alterando el archivo gráfico que dicha imagen muestra.

Property binding es siempre de una dirección, de padre a hijo. El padre define un valor y lo asigna al hijo. Por tanto, aunque el hijo modifique el valor nunca viajará al padre.

Obviamente, asignando un valor a un [src] de una imagen dicha imagen no va a cambiar el valor, pero en el caso de un componente creado por nosotros sí que podría ocurrir.

```
<app-cliente [nombre]="nombreCliente"></app-cliente>
```

En este caso, el padre ha definido que el nombre del cliente sea lo que éste tiene en su propiedad "nombreCliente". Podría ocurrir que el componente app-cliente cambiase el valor de su propia propiedad "nombre". Si eso ocurre, el componente padre no notará nada por darse un binding de una única dirección (padres a hijos).

Nota: En breve explicaremos cómo es posible crear componentes que acepten valores de entrada en sus propiedades. Esto adelantamos que se hace con el decorador `@Input` al definir la propiedad en la clase del componente.

Valores posibles para una propiedad

Entre las comillas asignadas como valor a una propiedad podemos colocar varias cosas. Lo común es que bindeemos mediante una propiedad del componente, tal como se ha visto en ejemplos anteriores, pero podríamos colocar otro código TypeScript.

Así, dentro del valor de una propiedad, podemos usar un pequeño conjunto de operadores como el de negación `!"`, igual que teníamos en las expresiones en el binding por interpolación de cadenas.

Además vamos a ver un ejemplo interesante porque nos dará pie a explicar otras cosas: podríamos bindear a un literal.

```
<img [src]='ruta.jpg'>
```

En este caso `"ruta.jpg"` está escrito entre comillas simples, luego es un literal de string. Quiere decir que Angular lo evaluará como una cadena, lo asignará a la propiedad y se olvidará de él. Esto no tiene mucho sentido, ya que lo podríamos haber conseguido exactamente el mismo efecto con el propio atributo HTML, sin colocar los corchetes.

```
<img src='ruta.jpg'>
```

Esta posibilidad (la asignación de un literal) tendrá más sentido al usarlo en componentes personalizados.

```
<app-cliente nombre="DesarrolloWeb.com"></app-cliente>
```

Al no colocar la propiedad entre corchetes, es como hacer un binding a un literal de string. Por tanto, el componente `app-cliente` recibirá el valor `"DesarrolloWeb.com"` en su propiedad `"nombre"` al inicializarse, pero no se establecerá ningún binding durante su existencia.

También podemos bindear a un método, provocando que ese método se ejecute para aplicar un valor a la propiedad bindeada.

```
<p [ngClass]="obtenClases()">Esto tiene class o clases dependiendo de la ejecución de un método</p>
```

En este caso, cada vez que cambie cualquier cosa del estado del componente, se ejecutará de nuevo `obtenClases()` para recibir el nuevo valor de clases CSS que se deben aplicar al elemento.

Nota: El código de tu método `obtenClases()` debería ser muy conciso y rápido de ejecutar, puesto que su invocación se realizará muy repetidas veces. En definitiva, debes usarlo con cuidado para no afectar al rendimiento de la aplicación.

Evitar efectos laterales

Del mismo modo que ocurría con las expresiones en la Interpolación de strings, se debe evitar que al evaluar el valor de una propiedad se cambie el estado del componente.

Por ejemplo, si en la expresión a evaluar para definir el valor de una propiedad colocas una asignación, verás un error de Angular advirtiéndote este asunto.

```
<img [src]="ruta = 'ruta.jpg'">
```

Esto último no se puede hacer. Angular se quejará con un mensaje como "template parse errors: Parser Error: Bindings cannot contain assignments".

Ten en cuenta que, al bindear una propiedad mediante el valor devuelto por un método, Angular no mostrará errores frente a efectos laterales producidos por modificar el estado del componente. Por ejemplo en:

```
<img [src]="calculaSrc()">
```

Si en `calculaSrc()` cometemos la imprudencia de modificar el estado, Angular no será capaz de detectarlo y podremos encontrar inconsistencias entre las propiedades del componente y lo que se está mostrando en el template.

De momento es todo lo que tienes que saber sobre el bindeo de propiedades. En el siguiente artículo revisaremos otra de las preguntas frecuentes de las personas que comienzan con Angular, ¿Cuándo usar interpolación o bindeo a propiedades?.

Este artículo es obra de Miguel Ángel Álvarez
Fue publicado por primera vez en 09/ 11/ 2017
Disponible online en <http://desarrolloweb.com/articulos/bindeo-propiedades-angular.html>

Binding a propiedad vs interpolación de strings, en Angular

Cuándo usar el binding de propiedad o la interpolación de strings, dos alternativas con objetivos distintos en Angular, que a veces pueden confundir.

En este artículo vamos a analizar en contraposición dos alternativas de binding en Angular, que a veces pueden producir dudas, básicamente porque podrían ser usadas en contextos similares.

El objetivo es ayudar al lector a identificar los casos donde se recomienda usar el bindeo de propiedades o la interpolación de strings, de modo que sea capaz de escoger la mejor opción en cada caso.

Recuerda que estos conceptos ya se han explicado, de modo que deberías conocer al menos qué diferencias tienen. Si no es así te recomendamos leer el artículo sobre la [interpolación de strings](#) y el [binding de propiedades](#).



Dos mecanismos distintos, que pueden usarse en situaciones parecidas

Sabiendo que la interpolación de strings y el binding de propiedades son cosas bien distintas, queremos mostrar que en ocasiones pueden ser usadas en los mismos lugares, produciendo idéntico resultado.

Observemos el siguiente código.

```
<p>
  
</p>
<p>
  <img [src]="urlImagen">
</p>
```

En el primer caso se está realizando interpolación, colocando como valor de una propiedad algo que viene de una variable del componente. En el segundo caso se está usando binding a la propiedad src, colocando el valor de la misma variable.

Ambos casos resuelven el HTML del componente con el mismo resultado. ¿Cuál es el correcto?

Tenemos más ejemplos. Observa ahora este otro caso, en el que básicamente colocamos una clase obtenida por medio de una propiedad del componente. El resultado será el mismo.

```
<ul>
  <li [class]="valorClass">item 1</li>
  <li class="{{valorClass}}">item 2</li>
</ul>
```

Por último veamos un caso donde también obtenemos el mismo resultado, pero usando dos aproximaciones mucho más diferentes.

```
<p>{{texto}}</p>

<p [innerHTML]="texto"></p>
```

En el primer párrafo estamos colocando como contenido un texto por interpolación. Mientras que en el segundo párrafo se coloca el mismo texto, a través de la propiedad del componente `innerHTML`.

Si estás confuso con ambas posibilidades sería más que normal. Intentaremos arrojar algo de luz sobre cuál de ellas te conviene usar en cada caso.

Uso de la interpolación con cadenas

En todos los casos anteriores estamos en definitiva volcando cadenas en diversos puntos del template. Aunque esas cadenas a veces se coloquen en propiedades del componente, no dejan de ser cadenas.

En todos los lugares donde se trate de colocar cadenas en una vista, puede que tenga más sentido usar interpolación, debido en mi opinión a la claridad del código de la vista.

Pero ojo, porque esto es una opinión personal, que no atiende a ningún motivo relacionado con el propio framework. De hecho, lo cierto es que no existe una diferencia real entre una u otra posibilidad, por lo que se pueden usar indistintamente.

Lo importante sería establecer una norma a seguir por todos los desarrolladores en cuanto a cuál usar en los casos en los que no exista diferencia, de modo que todo el mundo tome las mismas decisiones y obtengamos un código más homogéneo.

Uso de property binding con valores distintos de cadenas

El caso de valores que no sean cadenas de caracteres es ya distinto. Realmente la interpolación de strings siempre debe devolver una cadena, por lo que no será adecuada si aquello que se tiene que usar en una propiedad no es directamente una cadena.

Por ejemplo, la propiedad `"disabled"` de un botón debe igualarse por property binding a un valor booleano, luego no será adecuado usar interpolación. Por tanto, lo correcto sería esto:

```
<button [disabled]="activo">Clic aquí</button>
```

En este caso concreto, no deberíamos usar interpolación, ya que el resultado no será el deseado. Por ejemplo, esto no sería correcto.

```
<button disabled="{{activo}}">Clic aquí</button>
```

El motivo es que `activo` se interpolaría por la palabra `"false"`. En este caso `disabled="false"` en HTML sería lo mismo que colocar el atributo `"disabled"` sin acompañarlo de ningún valor, lo que equivale al final a que el botón esté desactivado.

En definitiva, cada vez que tengamos que usar un dato que no sea una cadena, es preferible colocar el

binding de propiedad. Por ejemplo, la directiva `ngStyle` espera un objeto como valor, por lo que estaríamos obligados a usar `property binding`.

```
<div [ngStyle]="estilo">Test de estilo</div>
```

Sanitización de cadenas en templates de Angular

En el caso que tanto `property binding` como `string interpolation` se usen para volcar cadenas en el template, hay una diferencia destacable en la manera como las cadenas se van a sanitizar antes de volcarse al template.

En vista de esta propiedad en el componente:

```
cadenaXSS = 'Esto es un <script>alert("test")</script> con XSS';
```

La propiedad `cadenaXSS` presenta un problema de seguridad, por una inyección de código de un script, lo que se conoce por XSS.

Podríamos volcarla en un template de la siguiente manera:

```
<div>{{ cadenaXSS }}</div>  
<div [innerHTML]="cadenaXSS"></div>
```

Angular en ambos casos hace un saneamiento de la cadena, desactivando la etiqueta `SCRIPT`, para evitar problemas de seguridad por inyección de código XSS. Pero el resultado de ese saneamiento es diferente en cada caso.

En este caso concreto, la salida que obtendremos es la siguiente:

```
Esto es un <script>alert("test")</script> con XSS Esto es un alert("test") con XSS
```

Tienes que observar que, en el caso de la interpolación de strings, la etiqueta `script` te aparece como texto en la página. Es decir, será un simple texto y no una apertura y cierre de una etiqueta para colocar un script Javascript.

Esperamos que con estas indicaciones hayas resuelto algunas dudas típicas que pueden surgir cuando empiezas a trabajar en el desarrollo con Angular. Ahora estás en condiciones de escoger la mejor opción en cada caso, entre `string interpolation` y `Property binding`.

Servicios en Angular

Comenzamos una serie de artículos que nos explicarán qué son los servicios, una de las piezas principales de las aplicaciones en Angular. Estudiaremos cómo usar servicios dentro de aplicaciones, cómo crearlos, cómo agregarlos a módulos o componentes, cómo instanciar servicios, etc. Veremos también qué papel juegan los servicios dentro del marco de una aplicación, con ejemplos de uso que puedan ilustrar las buenas prácticas a los programadores.

Servicios en Angular

Qué son los servicios en el framework Javascript Angular, cómo crear nuestros primeros services, cómo usarlos desde los componentes.

Hasta ahora en el [Manual de Angular](#) hemos hablado mucho de componentes, y también de módulos. Pero existen otros tipos de artefactos que podemos usar para organizar el código de nuestras aplicaciones de una manera más lógica y fácil de mantener.

En importancia a componentes y módulos le siguen los servicios. Con este artículo iniciamos su descripción. Los servicios, o "services" si lo prefieres en inglés, son una de las piezas fundamentales en el desarrollo de aplicaciones Angular. Veremos qué es un servicio y cómo dar nuestros primeros pasos en su creación y utilización en un proyecto.



Concepto de servicio en Angular

Si eres ya viejo conocido del desarrollo con Angular sabrás lo que es un servicio, pues es una parte importante dentro de la arquitectura de las aplicaciones con este framework. Pero si eres nuevo seguro que te vendrá bien tener una pequeña descripción de lo que es un servicio.

Básicamente hemos dicho anteriormente que el protagonista en las aplicaciones de Angular es el componente, que las aplicaciones se desarrollan en base a un árbol de componentes. Sin embargo, a medida que nuestros objetivos sean más y más complejos, lo normal es que el código de los componentes también vaya aumentando, implementando mucha lógica del modelo de negocio.

En principio esto no sería tan problemático, pero sabemos que la organización del código, manteniendo piezas pequeñas de responsabilidad reducida, es siempre muy positiva. Además, siempre llegará el momento en el que dos o más componentes tengan que acceder a los mismos datos y hacer operaciones similares con ellos, que podrían obligarnos a repetir código. Para solucionar estas situaciones tenemos a los servicios.

Básicamente un servicio es un proveedor de datos, que mantiene lógica de acceso a ellos y operativa relacionada con el negocio y las cosas que se hacen con los datos dentro de una aplicación. Los servicios serán consumidos por los componentes, que delegarán en ellos la responsabilidad de acceder a la información y la realización de operaciones con los datos.

Cómo crear un servicio

Tal como viene siendo costumbre en el desarrollo con Angular, nos apoyaremos en [Angular CLI](#) para la creación del esqueleto, o scaffolding, de un servicio.

Para crear un servicio usamos el comando "generate service", indicando a continuación el nombre del servicio que queremos generar.

```
ng generate service clientes
```

Esto nos generaría el servicio llamado "ClientesService". La coletilla "Service", al final del nombre, te la agrega Angular CLI, así como también nombra al archivo generado con la finalización "-service", para dejar bien claro que es un servicio.

Es habitual que quieras colocar el servicio dentro de un módulo en concreto, para lo que puedes indicar el nombre del módulo, una barra "/" y el nombre del servicio. Pero atención: ahora lo detallaremos mejor, pero queremos advertir ya que esto no agregará el servicio al código de un módulo concreto, sino que colocará el archivo en el directorio de ese módulo. Enseguida veremos qué tienes que hacer para que este servicio se asigne realmente al módulo que desees.

```
ng generate service facturacion/clientes
```

Nota: como los servicios son algo que se suele usar desde varios componentes, muchos desarrolladores optan por crearlos dentro de un módulo compartido, que puede llamarse "común", "shared" o algo parecido.

Ahora, si quieres, puedes echarle un vistazo al código básico de un service, generado por el CLI. Enseguida lo examinamos. No obstante, antes queremos explicar otro paso importante para que el servicio se pueda usar dentro de la aplicación.

Agregar la declaración del servicio a un módulo

Para poder usar este servicio es necesario que lo agregues a un módulo. Inmediatamente lo podrás usar en

cualquiera de los componentes que pertenecen a este módulo. Este paso es importante que lo hagas, puesto que, al contrario de lo que ocurría al crear un componente con el CLI, la creación de un servicio no incluye la modificación del módulo donde lo has creado.

Así pues, vamos a tener que declarar el servicio manualmente en el módulo. Lo haremos gracias al decorador del módulo (`@NgModule`), en el array de "providers". El decorador de un módulo con el array de providers podría quedar más o menos así.

```
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [ListadoClientesComponent],
  providers: [ClientesService]
})
```

Obviamente, tendrás que hacer el correspondiente import al módulo, para que se conozca la clase `ClientesService`.

```
import { ClientesService } from './clientes.service';
```

Como decimos, ahora este módulo ha declarado el servicio como "provider", por lo que sus componentes podrán usar este servicio.

Nota: El provider que acabamos de declarar en el módulo es un array que también podremos declarar en un componente, con lo que podríamos asignar un servicio a un componente en concreto y no a un módulo completo. Más adelante explicaremos dónde puedes declarar el provider, dependiendo de cómo quieres que tu servicio se gestione a nivel de aplicación.

Código básico de un service en Angular

Ahora podemos examinar el código generado para nuestro servicio y aprender nuevas cosas de Angular. Este sería nuestro recién creado servicio "ClientesService".

```
import { Injectable } from '@angular/core';

@Injectable()
export class ClientesService {

  constructor() { }

}
```

Como verás, el servicio no tiene nada todavía, solo su declaración, pero hay cosas interesantes que tenemos

que explicar, principalmente un nuevo decorador que no habíamos conocido hasta el momento: "@injectable".

El decorador @injectable indica a Angular que la clase que se decora, en este caso la clase ClientesService, puede necesitar dependencias que puedan ser entregadas por inyección de dependencias. De momento puedes quedarte que los servicios necesitan de este decorador, aunque realmente disponer de él no es condición indispensable.

Nota: La inyección de dependencias es un patrón de desarrollo bastante habitual en el mundo de los frameworks. Ya es familiar para los antiguos desarrolladores de Angular, pues estaba ya implementada en las versiones de AngularJS (1.x). Sin embargo, no es algo propio o específico de Angular, sino de la programación en general. Tenemos un artículo que explica de manera teórica más cosas importantes sobre este patrón de diseño de software: [Inyección de dependencias](#).

El import, arriba del todo, { Injectable } from '@angular/core', lo que hace es que nuestra clase conozca y sea capaz de usar el decorador @injectable.

Por lo demás, el servicio está vacío, pero le podemos poner ya algo de código para que nos sirva de algo. De momento vamos a exponer mediante el servicio una simple propiedad con un dato, que luego vamos a poder consumir desde algún componente. Para ello usamos las propiedades de la clase, tal como nos permite TypeScript.

```
export class ClientesService {  
  accesoFacturacion = 'https://login.example.com';  
  constructor() { }  
}
```

En nuestra clase ClientesService hemos creado una propiedad llamada "accesoFacturacion", en la que hemos asignado un valor que sería común para todos los clientes. El dato es lo de menos, lo interesante es ver que la declaración no dista de otras declaraciones en clases que hayamos visto ya.

Cómo inyectar dependencias de servicios

Ahora nos toca ver la magia de Angular y su inyección de dependencias, que vamos a usar para poder disponer del servicio en un componente.

Como en otros frameworks, en Angular la inyección de dependencias se realiza por medio del constructor. En el constructor de un componente, que hasta ahora habíamos dejado siempre vacío, podemos declarar cualquiera de los servicios que vamos a usar y el framework se encargará de proporcionarlo, sin que tengamos que realizar nosotros ningún trabajo adicional.

Esto es tan sencillo como declarar como parámetro la dependencia en el constructor del componente.

```
constructor(private clientesService: ClientesService) { }
```

De esta manera estamos indicando a TypeScript y Angular que vamos a usar un objeto "clientesService" que es de la clase "ClientesService". A partir de entonces, dentro del componente existirá ese objeto, proporcionando todos los datos y funcionalidad definida en el servicio.

Nota: Es muy importante que al declarar la inyección de dependencias en el constructor no te olvides de la declaración de visibilidad (ya sea public o private), pues si no la colocas no se generará la propiedad en el objeto construido. Si defines la propiedad de la clase como pública, afectará a su visibilidad: constructor(public clientesService: ClientesService) esto es algo que te proporciona TypeScript. Público o privado el efecto será el mismo, se creará la propiedad "clientesService", inyectando como valor un objeto de la clase ClientesService.

Explicando con detalle la declaración de propiedades implícita en el constructor

Aquí tenemos que detenernos para explicar algo de la magia, o azúcar sintáctico, que te ofrece TypeScript. Porque el hecho que se declare una propiedad en un objeto solo porque se reciba un parámetro en el constructor no es algo usual en Javascript.

Cuando TypeScript detecta el modificador de visibilidad "public" o "private" en el parámetro enviado al constructor, inmediatamente declara una propiedad en la clase y le asigna el valor recibido en el constructor. Por tanto, esta declaración:

```
export class ListadoClientesComponent {  
  constructor(public clientesService: ClientesService) { }  
}
```

Sería equivalente a escribir todo el código siguiente:

```
export class ListadoClientesComponent {  
  clientesService: ClientesService;  
  constructor(clientesService: ClientesService) {  
    this.clientesService = clientesService;  
  }  
}
```

En resumen, TypeScript entiende que, si defines la visibilidad de un parámetro en el constructor, o que quieres hacer en realidad es crear una propiedad en el objeto recién construido, con el valor recibido por parámetro.

Usando el servicio en el componente

Ahora, para acabar esta introducción a los servicios en Angular, tenemos que ver cómo usaríamos este servicio en el componente. No nos vamos a detener demasiado en hacer ejemplos elaborados, que podemos abordar más adelante, solo veremos un par de muestras sobre cómo usar la propiedad declarada en el servicio.

1.- Usando el servicio dentro de la clase del componente

Dentro de la clase de nuestro componente, tendremos el servicio a partir de la propiedad usada en su declaración. En el constructor dijimos que el servicio se llamaba "clientesService", con la primera en minúscula por ser un objeto.

Pues como cualquier otra propiedad, accederemos a ella mediante la variable "this".

```
export class ListadoClientesComponent implements OnInit {  
  
  constructor(public clientesService: ClientesService) { }  
  
  ngOnInit() {  
    console.log(this.clientesService);  
  }  
  
}
```

El ejemplo no vale para mucho, solo para mostrar que, desde que esté creado el objeto podemos acceder al servicio con "this.clientesService".

También nos sirve para recordar que, el primer sitio donde podríamos usar los servicios declarados es en el método `ngOnInit()`. Dicho de otro modo, si necesitamos de estos servicios para inicializar propiedades en el componente, el lugar donde ponerlos en marcha sería el `ngOnInit()`.

2.- Usando el servicio en el template de un componente

Por su parte, en el template de un componente, podrás también acceder al servicio, para mostrar sus propiedades o incluso invocar sus métodos como respuesta a un evento, por ejemplo.

Como el servicio está en una propiedad del componente, podremos acceder a él mediante ese nombre de propiedad.

```
<p>  
  URL De acceso: {{clientesService.accesoFacturacion}}  
</p>
```

Conclusión a la introducción a los servicios en Angular

Eso es todo lo que tienes que saber para comenzar a experimentar con los servicios en Angular. Es momento que pongas las manos en el código y comiences a experimentar por tu cuenta, creando tus propios servicios y usándolos desde tus componentes.

En artículos posteriores veremos servicios un poco más completos, que hagan más cosas que el que hemos visto en la presente entrega.

Este artículo es obra de Miguel Ángel Álvarez
Fue publicado por primera vez en 04/ 12/ 2017
Disponible online en <http://desarrolloweb.com/articulos/serviciosangular.html>

Usar clases e Interfaces en los servicios Angular

Al desarrollar servicios con Angular es una buena idea usar clases e Interfaces para definir las estructuras de datos. Veremos cómo y por qué.

Si algo caracteriza a TypeScript, el lenguaje con el que se desarrolla en Angular, es el uso de tipos. Podemos usar tipos primitivos en variables, lo que nos ayudará a recibir ayudas en el momento en el que desarrollamos el código, pero también podemos definir tipos más complejos por medio de clases e interfaces.

A lo largo del [Manual de Angular](#) hemos declarado variables indicando sus tipos en muchas ocasiones, pero hasta ahora no hemos usado casi nunca las clases para tipar y mucho menos las interfaces. En los servicios es un buen lugar para empezar a acostumbrarnos a ello, lo que nos reportará muchos beneficios a la hora de programar.

Este artículo no avanzará tanto en lo que es ofrecer nuevos conocimientos de Angular, sino más bien en el uso de costumbres que nos ayudarán en el día a día y que por tanto son muy comunes a la hora de desarrollar aplicaciones profesionales. Para entenderlo es bueno que conozcas los [fundamentos de TypeScript](#) y también específicamente de las [interfaces TypeScript](#). Obviamente, necesitarás también saber lo [que es un servicio y cómo crearlos y usarlos en Angular](#).



Ayudas en tiempo de desarrollo, no en tiempo de ejecución

Aunque puede resultar de cajón para muchas personas, queremos comenzar por señalar que las ayudas que te ofrece TypeScript serán siempre en tiempo de desarrollo. Para poder aprovecharte de ellas lo ideal es disponer de un editor que entienda TypeScript, mostrando en el momento que desarrollas los problemas detectados en el código.

A la hora de compilar la aplicación, para su ejecución en el navegador, también te ayudará TypeScript, especificando los errores que ha encontrado en el código, derivados por un uso incorrecto de las variables y sus tipos.

Sin embargo, una vez que el navegador ejecuta el código debes entender que todo lo que interpreta es Javascript, por lo que los tipos no interferirán en nada. No será más pesado para el navegador, ni te alertará de problemas que se puedan producir, ya que éste solo ejecutará código Javascript. Sin embargo, lo cierto es que si tipaste correctamente todo lo que estuvo en tu mano y el compilador no te alertó de ningún error, difícilmente se producirá un error por un tipo mal usado en tiempo de ejecución.

Si ya tienes cierta experiencia con TypeScript habrás observado la cantidad de errores que se detectan prematuramente en el código y lo mucho que el compilador te ayuda a medida que estás escribiendo. Esto facilita ahorrar mucho tiempo y disfrutar de una experiencia de programación más agradable. Por todo ello, usar los tipos siempre que puedas es una idea estupenda y en este artículo queremos explicarte cómo ir un poco más allá, en el marco de los servicios de Angular.

Crear una clase para definir el tipo de tus objetos

La manera más común de definir tipos, para las personas acostumbradas a lenguajes de programación orientados a objetos, son las clases.

Si tienes en tu aplicación que trabajar con cualquier tipo de entidad, es una buena idea que crees una clase que especifique qué datos contiene esa estructura.

Si nuestra aplicación usa clientes, lo más normal es que definas la clase cliente, más o menos como esto:

```
class Cliente {  
  nombre: String;  
  cif: String;  
  direccion: String;  
  creado: Date;  
}
```

Luego, cuando quieras crear un cliente podrás declararlo usando el tipo de la clase Cliente.

```
let cliente: Cliente;
```

A partir de ahora, en el caso que uses un código que no respete esa declaración se alertará convenientemente.

```
class Cliente {
  nombre: String;
  cif: String;
  direccion: String;
  creado: Date;
}

let cliente: Cliente;

cliente = { name: 'test' }
```

[ts]
Type '{ name: string; }' is not assignable to type 'Cliente'.
Object literal may only specify known properties, and 'name' does not exist in type 'Cliente'.
(property) name: string

Crear una interfaz para definir el tipo de tus objetos

Más o menos lo mismo podemos conseguir con una interfaz de TypeScript. Es algo que ya hemos explicado en el artículo de Interfaces TypeScript. Pero en resumen, puedes definir una interfaz de clientes de esta manera.

```
interface Cliente {
  nombre: String;
  cif: String;
  direccion: String;
  creado: Date;
}
```

Luego podemos crear una variable asignando la interface como si fuera un tipo.

```
let cliente: Cliente;
```

A partir de aquí el editor nos avisará cuando el valor asignado no cumpla la interfaz.

```
interface Cliente {
  nombre: String;
  cif: String;
  direccion: String;
  creado: Date;
}

let cliente: Cliente;

cliente = { name: 'test' }
```

[ts]
Type '{ name: string; }' is not assignable to type 'Cliente'.
Object literal may only specify known properties, and 'name' does not exist in type 'Cliente'.
(property) name: string

¿Clases o interfaces?

Si ambas construcciones te sirven para más o menos lo mismo ¿cuándo usar clases y cuándo usar interfaces? La respuesta depende un poco sobre cómo vayas a generar los datos.

Es muy habitual usar simplemente interfaces, desprovistas de inicialización y funcionalidad, ya que esas partes es común que sean delegadas en los servicios. Pero en el caso de usar clases, tus nuevos objetos serán creados con la palabra "new".

```
let cliente1 = new Cliente();
cliente.nombre = 'EscuelaIT S.L.';
cliente.cif = 'B123';
cliente.direccion = 'C/ Del Desarrollo Web .com';
cliente.creado = new Date(Date.now());
```

Si los objetos que debes crear presentan tareas de inicialización pesadas, usando clases, podrás apoyarte en los constructores para resumir los pasos para su creación. En este caso podemos conseguir un código más compacto:

```
let cliente1 = new Cliente('EscuelaIT S.L.', 'B123', 'C/ Del Desarrollo Web .com');
```

Nota: obviamente, para que esto funcione tienes que haber creado el correspondiente constructor en la implementación de la clase Cliente. Observa que en caso de tener un constructor podríamos ahorrarnos pasarle el objeto de la clase Date, para asignar en la propiedad "creado", puesto que podríamos delegar en el constructor la tarea de instanciar un objeto Date con la fecha con el instante actual.

Sin embargo, los objetos que contienen datos para representar en tu aplicación no siempre se generan mediante tu propio código frontend, ya que es habitual que esos datos provengan de algún web service (API REST o similar) que te los proporcione por medio de llamadas "HTTP" (Ajax). En estos casos no harás "new", sino que usarás Angular para solicitar al servidor un dato, que será devuelto en forma de un objeto JSON habitualmente. En estos casos es especialmente idóneo definir una interfaz y declarar la el objeto que te devuelva el servidor con el tipo definido por esa interfaz. Más adelante veremos ejemplos de este caso en concreto.

Uses o no servicios web para traerte los datos, con interfaces también puedes crear objetos que correspondan con el tipo de datos definido en la interfaz. Simplemente los crearías en tus servicios por medio de literales de objeto.

```
let cliente: Cliente;
cliente = {
  nombre: 'EscuelaIT',
  cif: 'ESB123',
  direccion: 'C/ de arriba, 1',
  creado: new Date(Date.now())
};
```

Como puedes ver, la decisión sobre usar clases o interfaces puede depender de las necesidades de tu aplicación, o incluso de tus preferencias o costumbres de codificación.

Incluso, nada te impide tener ambas cosas, una interfaz y una clase implementando esa interfaz, para disponer también de la posibilidad de instanciar objetos ayudados por un constructor.

```
export interface ClienteInterface {  
  nombre: String;  
  cif: String;  
  direccion: String;  
  creado: Date;  
}  
  
export class Cliente implements ClienteInterface {  
  creado: Date;  
  constructor(public nombre: string, public cif: String, public direccion: String) {  
    this.creado = new Date(Date.now());  
  }  
}
```

Definir el modelo de datos en un archivo externo

Para ordenar el código de nuestra aplicación es también habitual que el modelo de datos se defina en un archivo TypeScript aparte. En ese archivo puedes guardar la declaración del tipo y luego importarlo en cualquier lugar donde quieras usar ese tipo de datos.

Archivo "clientes.modelo.ts"

Por ejemplo, tendríamos el archivo "clientes.modelo.ts" y ese archivo tendría, por ejemplo la declaración de la interfaz:

```
export interface Cliente {  
  nombre: String;  
  cif: String;  
  direccion: String;  
  creado: Date;  
}
```

Nota: no te olvides de colocar la palabra "export" delante del nombre de la interfaz, para que esa declaración se pueda importar desde otros archivos.

Archivo "clientes.service.ts"

Por su parte, en el servicio tendríamos que hacer el import de este tipo, definido en `clientes.modelo.ts`, y usarlo al declarar objetos.

```
import { Cliente } from './cliente.modelo';
```

Por supuesto, una vez definido este tipo de datos, gracias a la correspondiente interfaz importada, lo debes de usar en tu servicio, en el mayor número de lugares donde puedas, lo que te proporcionará un código más robusto, capaz de advertirte de cualquier tipo de problemas en tiempo de desarrollo y ahorrarte muchas complicaciones.

Este sería el código de nuestro servicio, donde hacemos uso del tipo `Cliente` importado.

```
import { Injectable } from '@angular/core';
import { Cliente } from './cliente.modelo';

@Injectable()
export class ClientesService {

    clientes: Cliente[] = [];

    constructor() { }

    anadirCliente(cliente: Cliente) {
        this.clientes.push(cliente);
    }

    clienteNuevo(): Cliente {
        return {
            nombre: 'DesarrolloWeb.com',
            cif: 'B123',
            direccion: 'Oficinas de EscuelaIT, C/ Formación online nº 1',
            creado: new Date(Date.now())
        };
    }
}
```

En el código anterior debes fijarte en varias cosas sobre TypeScript:

- La declaración del array "clientes" indica que sus elementos son de tipo `Cliente`.
- En el parámetro del método "anadirCliente" hemos declarado el tipo de datos que recibimos, de tipo `Cliente`.
- El valor de retorno del método `clienteNuevo()` se ha declarado que será de tipo `Cliente`.

Obviamente, si durante la escritura de tu código, no solo en este servicio sino también en cualquier otro lugar donde se use, no envías objetos de los tipos esperados, TypeScript se quejará y te lo hará saber.

Fíjate en la siguiente imagen. Es el código de un componente llamado "AltaClienteComponent". Hemos cometido la imprudencia de declarar un cliente como de tipo `String` (flecha roja). Por ello, todos los métodos en los que trabajamos con ese cliente, apoyándonos en el servicio `ClientesService`, están marcados

como si fueran un error (flechas amarillas).



```
import { ClientesService } from '../clientes.service';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'dw-alta-cliente',
  templateUrl: './alta-cliente.component.html',
  styleUrls: ['./alta-cliente.component.css']
})
export class AltaClienteComponent implements OnInit {

  cliente: string;
  constructor(private clientesService: ClientesService) { }

  ngOnInit() {
    this.cliente = this.clientesService.clienteNuevo();
  }

  agregarCliente() {
    this.clientesService.anadirCliente(this.cliente);
    this.cliente = this.clientesService.clienteNuevo();
  }
}
```

Nota: La anterior imagen pertenece al editor Visual Studio Code, que ya incorpora de casa todo lo necesario para ayudarte al programar con TypeScript, mostrando los correspondientes errores encontrados en tiempo de desarrollo.

Conclusión

Con esto hemos llegado a un ejemplo de un servicio un poco más complejo del que vimos en el artículo anterior, que también se comportará de un modo más robusto, alertando de posibles problemas a la hora de escribir el código, y cuando el compilador de TypeScript analice el código para convertirlo a Javascript.

También nos ha dado pie a usar algunas de las cosas que nos aporta TypeScript, para seguir aprendiendo este superset de Javascript. Espero que a partir de ahora puedas esforzarte por sacar mayor partido a este lenguaje.

Fue publicado por primera vez en 14/ 12/ 2017

Disponible online en <http://desarrolloweb.com/articulos/das-interface-servicios-angular.html>

Práctica Angular con Módulos, Componentes y Servicios

Vamos a crear un ejemplo práctico con lo visto hasta el momento en el Manual de Angular en el que pondremos en uso los conocimientos adquiridos sobre módulos, componentes y servicios.

Somos conscientes de que lo que hemos visto hasta este punto del [Manual de Angular 2](#) puede resultar complicado de asimilar si no realizamos un ejemplo completo, que nos ayude a ver de manera global el flujo de desarrollo con este framework. Por ello, vamos a pararnos aquí para practicar un poco.

Veremos cómo trabajar en el desarrollo de una aplicación Angular en la que participan todos los integrantes que hemos explicado hasta ahora, es decir: [módulos](#), [componentes](#) y [servicios](#).

En el presente ejercicio vamos a construir un sistema de alta de clientes y un listado de clientes que irá incrementando ítems, a medida que los demos de alta. Será interesante para los lectores, pero no deja de ser un sencillo demo de cómo trabajar en Angular en estos primeros pasos, pues hay mucho más en el framework que no hemos tenido tiempo de aprender todavía.


Así que vamos a poner manos a la obra. Procuraré ser muy específico, yendo paso a paso. Confío no olvidarme de mencionar ninguna parte del proceso. Ten en cuenta además que solamente resumiré algunos detalles del código, pues en artículos pasados del presente manual ha quedado ya explicado todo lo que vamos a ver.

Iremos presentando listados de cada una de las partes del código que se irán realizando y al final de este artículo encontrarás también el enlace al repositorio con el código completo.

```
constructor(private clientesService: ClientesService) { }

ngOnInit() {
  this.cliente = this.clientesService.nuevoCliente();
  this.grupos = this.clientesService.getGrupos();
}

nuevoCliente(): void {
  this.clientesService.agregarCliente(this.cliente);
}
```



Creamos nuestra aplicación

El primer paso es, usando el CLI, crear nuestra aplicación nueva. Lo haces entrando en la carpeta de tus proyectos y ejecutando el comando.

```
ng new clientes-app
```

Luego puedes entrar en la carpeta de la aplicación y lanzar el servidor web de desarrollo, para ver lo que se ha construido hasta el momento.

```
cd clientes app  
ng serve -o
```

Creamos nuestro módulo de clientes

La aplicación recién generada ya contiene un módulo principal, sin embargo, yo prefiero dejar ese módulo con pocas o ninguna cosa más de las que nos entregan por defecto al generar la aplicación básica. Por ello crearemos como primer paso un módulo nuevo, llamado "ClientesModule".

Encargamos a Angular CLI la creación del esqueleto de nuestro módulo con el siguiente comando:

```
ng generate module clientes
```

Definir el modelo de datos

Vamos a comenzar nuestro código por definir los tipos de datos que vamos a usar en esta aplicación. Vamos a trabajar con clientes y grupos.

Crearemos el modelo de datos dentro de la carpeta del módulo "clientes". No existe en Angular un generador de este tipo de modelos, por lo que crearé el archivo a mano con el editor. Lo voy a nombrar "cliente.model.ts".

En este archivo coloco las interfaces de TypeScript que definen los datos de mi aplicación.

```
export interface Cliente {  
  id: number;  
  nombre: string;  
  cif: string;  
  direccion: string;  
  grupo: number;  
}  
  
export interface Grupo {  
  id: number;  
  nombre: string;  
}
```

Como ves, he creado el tipo de datos Cliente y, por complicarlo un poquito más, el tipo de datos Grupo. Así, cada cliente generado pertenecerá a un grupo.

Nota: Esta parte de la creación de interfaces es perfectamente opcional. Solo la hacemos para usar esos tipos en la declaración de variables. El compilador de TypeScript nos avisará si en algún momento no respetamos estos tipos de datos, ayudando en tiempo de desarrollo y ahorrando algún que otro error derivado por despistes.

Crear un servicio para los clientes

Lo ideal es crear un servicio (service de Angular) donde concentremos las tareas de trabajo con los datos de los clientes, descargando de código a los componentes de la aplicación y centralizando en un solo archivo la lógica de la aplicación.

El servicio lo vamos a crear dentro de la carpeta del módulo clientes, por lo que especificamos la ruta completa.

```
ng generate service clientes/clientes
```

En el servicio tengo que hacer el import del modelo de datos, interfaces de Cliente y Grupo (creadas en el paso anterior).

```
import { Cliente, Grupo } from './cliente.model';
```

Nuestro servicio no tiene nada del otro mundo. Vamos a ver su código y luego explicaremos algún que otro punto destacable.

```
import { Injectable } from '@angular/core';
import { Cliente, Grupo } from './cliente.model';

@Injectable()
export class ClientesService {

  private clientes: Cliente[];
  private grupos: Grupo[];

  constructor() {
    this.grupos = [
      {
        id: 0,
        nombre: 'Sin definir'
      },
      {
        id: 1,
        nombre: 'Activos'
      },
      {
        id: 2,
        nombre: 'Inactivos'
      },
      {
        id: 3,
        nombre: 'Deudores'
      },
    ],
    this.clientes = [];
  }
}
```

```
getGrupos() {  
    return this.grupos;  
}  
  
getClientes() {  
    return this.clientes;  
}  
  
agregarCliente(cliente: Cliente) {  
    this.clientes.push(cliente);  
}  
  
nuevoCliente(): Cliente {  
    return {  
        id: this.clientes.length,  
        nombre: '',  
        cif: '',  
        direccion: '',  
        grupo: 0  
    };  
}
```

1. Las dos propiedades del servicio contienen los datos que va a mantener. Sin embargo, las hemos definido como privadas, de modo que no se puedan tocar directamente y tengamos que usar los métodos del servicio creados para su acceso.
2. Los grupos los construyes con un literal en el constructor. Generalmente los traerías de algún servicio REST o algo parecido, pero de momento está bien para empezar.
3. Agregar un cliente es un simple "push" al array de clientes, de un cliente recibido por parámetro.
4. Crear un nuevo cliente es simplemente devolver un nuevo objeto, que tiene que respetar la interfaz, ya que en la función nuevoCliente() se está especificando que el valor de devolución será un objeto del tipo Cliente.
5. Fíjate que en general está todo tipado, tarea opcional pero siempre útil.

Declarar el servicio para poder usarlo en los componentes

Una tarea fundamental para poder usar los servicios es declararlos en el "module" donde se vayan a usar.

Para añadir el servicio en el module "clientes.module.ts", el primer paso es importarlo.

```
import { ClientesService } from '../clientes.service';
```

Luego hay que declararlo en el array "providers".

```
providers: [  
    ClientesService  
]
```

Crear componente que da de alta clientes

Vamos a continuar nuestra práctica creando un primer componente. Es el que se encargará de dar de alta los clientes.

Generamos el esqueleto usando el Angular CLI.

```
ng generate component clientes/altaCliente
```

Comenzaremos editando el archivo del componente y luego iremos a trabajar con el template. Por tanto, vamos a abrir el fichero "alta-cliente.component.ts".

Agregar el servicio al componente

Muy importante. Para poder usar el servicio anterior, tengo que agregarlo al componente recién creado, realizando el correspondiente import.

```
import { ClientesService } from '../clientes.service';
```

Y posteriormente ya podré inyectar el servicio en el constructor del componente.

```
constructor(private clientesService: ClientesService) { }
```

Agregar el modelo de datos

Para poder seguir usando los tipos de datos de mi modelo, vamos a agregar el archivo donde se generaron las interfaces.

```
import { Cliente, Grupo } from '../cliente.model';
```

Código TypeScript completo del componente

El código completo de "alta-cliente.component.ts", para la definición de mi componente, quedaría más o menos así

```
import { Cliente, Grupo } from '../cliente.model';
import { ClientesService } from '../clientes.service';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-alta-cliente',
  templateUrl: './alta-cliente.component.html',
  styleUrls: ['./alta-cliente.component.css']
})
```

```
})  
  
export class AltaClienteComponent implements OnInit {  
  
  cliente: Cliente;  
  grupos: Grupo[];  
  
  constructor(private clientesService: ClientesService) { }  
  
  ngOnInit() {  
    this.cliente = this.clientesService.nuevoCliente();  
    this.grupos = this.clientesService.getGrupos();  
  }  
  
  nuevoCliente(): void {  
    this.clientesService.agregarCliente(this.cliente);  
    this.cliente = this.clientesService.nuevoCliente();  
  }  
}
```

Es importante mencionar estos puntos.

1. El componente declara un par de propiedades, el cliente y el array de grupos.
2. En el constructor, que se ejecuta lo primero, conseguimos una instancia del servicio de clientes, mediante la inyección de dependencias.
3. Posteriormente se ejecuta `ngOnInit()`. En este punto ya se ha recibido el servicio de clientes, por lo que lo puedo usar para generar los valores que necesito en las propiedades del componente.
4. El método `nuevoCliente()` es el que se ejecutará cuando, desde el formulario de alta, se produzca el envío de datos. En este código usamos el servicio `clientesService`, para agregar el cliente y generar un cliente nuevo, para que el usuario pueda seguir dando de alta clientes sin machacar los clientes anteriormente creados.

Template del componente, con el formulario de alta de cliente

Vamos a ver ahora cuál es el HTML del componente de alta de clientes, que básicamente contiene un formulario.

Pero antes de ponernos con el HTML, vamos a hacer una importante tarea. Consiste en declarar en el módulo de clientes que se va a usar la directiva `"ngModel"`. Para ello tenemos que hacer dos pasos:

En el archivo `"clientes.module.ts"` comenzamos por importar `"FormsModule"`.

```
import { FormsModule } from '@angular/forms';
```

En el decorador, indicamos el imports del `FormsModule`.

```
imports: [  
  CommonModule,  
  FormsModule
```

```
],
```

Ahora veamos el código del template, en el que reconocerás el uso de la propiedad "cliente" declarada en el constructor, así como el array de grupos.

```
<h2>Alta cliente</h2>
<p>
  <span>Nombre:</span>
  <input type="text" [(ngModel)]="cliente.nombre">
</p>
<p>
  <span>CIF:</span>
  <input type="text" [(ngModel)]="cliente.cif">
</p>
<p>
  <span>Dirección:</span>
  <input type="text" [(ngModel)]="cliente.direccion">
</p>
<p>
  <span>Grupo:</span>
  <select [(ngModel)]="cliente.grupo">
    <option *ngFor="let grupo of grupos" value="{{grupo.id}}">{{grupo.nombre}}</option>
  </select>
</p>
<p>
  <button (click)="this.nuevoCliente()">Guardar</button>
</p>
```

Usar el componente Alta cliente

Este componente, de alta de clientes, lo quiero usar desde el componente raíz de mi aplicación. Como el componente raíz está declarado en otro módulo, necesito hacer que conozca al `AltaClienteComponent`. Esto lo consigo en dos pasos:

1.- Agregar al exports `AltaClienteComponent`

En el módulo de clientes "`clientes.module.ts`" agrego al exports el componente que quiero usar desde otros módulos.

```
exports: [
  AltaClienteComponent
]
```

2.- Importar en el módulo raíz

Ahora, en el módulo raíz, "`app.module.ts`", debes declarar que vas a usar componentes que vienen de `clientes.module.ts`. Para ello tienes que hacer el correspondiente import:


```
import { ClientesModule } from '../clientes/clientes.module';
```

Y luego declaras el módulo en el array de imports:

```
imports: [  
  BrowserModule,  
  ClientesModule  
],
```

Hechos los dos pasos anteriores, ya puedes usar el componente en el template. Para ello simplemente tenemos que escribir su tag, en el archivo "app.component.html".

```
<app-alta-cliente></app-alta-cliente>
```

Llegado a este punto, si todo ha ido bien, deberías ver ya el componente de alta de clientes funcionando en tu página.

Nota: Si se produce cualquier error, entonces te tocará revisar los pasos anteriores o hacer una búsqueda con el texto del error que te devuelva la consola del navegador o el terminal, para ver dónde te has equivocado.

Crear el componente listado-cliente

Para acabar nuestra práctica vamos a crear un segundo componente de aplicación. Será el componente que nos muestre un listado de los clientes que se van generando.

Como siempre, comenzamos con un comando del CLI.

```
ng generate component clientes/listadoClientes
```

Ahora el flujo de trabajo es similar al realizado para el componente anterior. Vamos detallando por pasos.

Creas los import del servicio y de los tipos de datos del modelo.

```
import { Cliente, Grupo } from '../cliente.model';  
import { ClientesService } from '../clientes.service';
```

Injectas el servicio en el constructor.

```
constructor(private clientesService: ClientesService) { }
```

En este componente tendremos como propiedad el array de clientes que el servicio vaya creando. Así pues, declaras dicho array de clientes:

```
clientes: Cliente[];
```

Cuando se inicialice el componente tienes que solicitar los clientes al servicio. Esto lo hacemos en el método `ngOnInit()`.

```
ngOnInit() {  
    this.clientes = this.clientesService.getClientes();  
}
```

Código completo del componente `ListadoClientesComponent`

Puedes ver a continuación el código TypeScript completo de cómo nos quedaría este segundo componente.

```
import { Cliente, Grupo } from '../cliente.model';  
import { ClientesService } from '../clientes.service';  
import { Component, OnInit } from '@angular/core';  
  
@Component({  
    selector: 'app-listado-clientes',  
    templateUrl: './listado-clientes.component.html',  
    styleUrls: ['./listado-clientes.component.css']  
})  
export class ListadoClientesComponent implements OnInit {  
  
    clientes: Cliente[];  
    constructor(private clientesService: ClientesService) { }  
  
    ngOnInit() {  
        this.clientes = this.clientesService.getClientes();  
    }  
  
}
```

Código de la vista

Ahora podemos ver cómo sería la vista, código HTML, del listado de componentes.

```
<h2>  
    Listado clientes  
</h2>  
  
<div *ngIf="! clientes.length">No hay clientes por el momento</div>  
  
<div>
```

```
<article *ngFor="let cliente of clientes">
  <span>{{cliente.nombre}}</span>
  <span>{{cliente.cif}}</span>
  <span>{{cliente.direccion}}</span>
  <span>{{cliente.grupo}}</span>
</article>
</div>
```

No lo hemos comentado anteriormente, pero puedes darle un poco de estilo a los componentes editando el archivo de CSS. Por ejemplo, este sería un poco de CSS que podrías colocar en el fichero "listado-clientes.component.css".

```
article {
  display: flex;
  border-bottom: 1px solid #ddd;
  padding: 10px;
  font-size: 0.9em;
}
span {
  display: inline-block;
  width: 22%;
  margin-right: 2%;
}
```

Usar el componente del listado

Para usar este componente de listado de clientes, ya que lo queremos invocar desde el módulo raíz, tienes que ampliar el exports del module "clientes.module.ts".

```
exports: [
  AltaClienteComponent,
  ListadoClientesComponent
]
```

Como para el anterior componente, de alta de clientes, ya habíamos importado el módulo de clientes, no necesitas hacer nada más. Ahora ya puedes usar el usar el componente directamente en el template del componente raíz "app.component.html".

```
<app-listado-clientes></app-listado-clientes>
```

Es hora de ver de nuevo la aplicación construida y disfrutar del buen trabajo realizado. El aspecto de la aplicación que hemos realizado debería ser más o menos el siguiente:

Alta cliente

Nombre:

CIF:

Dirección:

Grupo:

Listado clientes

Cliente 1	B 123	C/ la la la	1
Cliente 2	A 334	Av. lo lo lo	2

Conclusión

Llegado a este punto, hemos terminado la práctica. Tenemos un sistema de alta y visualización de clientes dinámico, generado en una aplicación Angular con diferentes piezas del framework.

El código completo lo puedes ver en este repositorio en GitHub, en [este enlace que te lleva a un commit concreto](#).

Si has entendido el proceso y por tanto has podido seguir los pasos, estás en el buen camino. Ya conoces las principales piezas para el desarrollo en Angular. Aún queda mucho por aprender, pero los siguientes pasos serán más sencillos.

Si te ha faltado información para entender lo que hemos hecho en este artículo, te sugiero que leas con calma los correspondientes artículos del [Manual de Angular](#), en los que detallamos cada una de las piezas usadas en esta aplicación de demo.

Es normal también que te salgan errores sobre la marcha. Para resolverlos el compilador de TypeScript y el propio Angular ayudan bastante. Puedes preguntar o "googlear" para encontrar respuestas a tus problemas.

Este artículo es obra de Miguel Ángel Álvarez
Fue publicado por primera vez en 05/ 01/ 2018
Disponible online en <http://desarrolloweb.com/articulos/practica-angular-modulos-componentes-servicios.html>

Observables en Angular

En los próximos artículos introduciremos el concepto de observable, una herramienta para la programación reactiva dentro de aplicaciones de Angular, capaz de aumentar sensiblemente el rendimiento de las aplicaciones.

Introducción teórica a los observables en Angular

Aclaremos conceptos sobre los observables, por qué son importantes en Angular., qué es la programación reactiva y qué es RxJS.

En este artículo vamos a comenzar una nueva etapa en nuestro conocimiento de Angular, dedicando tiempo para una primera aproximación a los "observables", que son una de las principales novedades del framework a partir de Angular 2. Los observables representan también una de las mejores formas de optimizar una aplicación, aumentando su rendimiento.

En Angular se usan mucho los observables, dada su utilidad y versatilidad, aunque hemos de admitir que no es una tarea sencilla de aprender inicialmente. Intentaremos acercártelos de una manera sencilla, para que los puedas ir digiriendo poco a poco y suavizar su curva de aprendizaje. De momento, en este artículo del [Manual de Angular](#), nuestro objetivo es ofrecer una introducción general, para que comiences a conocerlos, así como el concepto de programación reactiva.



El "por qué" de los observables

Hemos dicho que los observables son una de las principales herramientas para programar aplicaciones de mayor rendimiento. Obviamente ese es el motivo por el cual se usan en Angular. Pero, ¿dónde reside esa mejora de rendimiento?

Uno de los motivos por los que Angular (y en especial su predecesor AngularJS) se convirtió en un framework tan usado es su capacidad de proporcionar una actualización automática de las fuentes de información. Es decir, en Angular somos capaces de usar un almacén de datos y, cuando se modifica ese almacén, recibir automáticamente sus cambios, sin que tengamos que programar a mano ese tránsito de la información.

Incluso, aunque un componente sea el encargado de actualizar ese almacén de datos, hemos visto que, usando servicios, podemos conseguir que otros componentes reciban automáticamente las actualizaciones. Si no lo recuerdas, consulta el artículo de [práctica con Angular con componentes, módulos y servicios](#).

Sin embargo, aunque Angular nos ahorra escribir mucho código, éste tiene un coste en términos de rendimiento. Quizás una aplicación pequeña no se verá tan afectada por el trabajo que Angular hace por debajo, para proporcionarnos automáticamente los cambios, pero sí se dejará notar en aplicaciones medianas. Ya las más grandes acusarán sensiblemente una mayor falta de rendimiento.

Nota: Para ser más concreto, Angular por debajo hace una serie de operaciones de manera repetitiva, en las que consulta los cambios en la fuente de datos, para saber cuándo se actualizan y entonces realizar las acciones oportunas para refrescar los datos en los lugares donde se están usando. Esa no era la mejor estrategia posible y por este motivo, otras [librerías como ReactJS](#), que supieron implementar un patrón de comportamiento más acertado, capaz de ofrecer mayor rendimiento, comenzaron a ganar su espacio ante la hegemonía de Angular.

La solución aplicada en Angular 2 (y que mantienen las siguientes versiones, 4, 5...) fué usar un patrón llamado "Observable", que básicamente nos ahorra tener que hacer consultas repetitivas de acceso a la fuente de información, aumentando el rendimiento de las aplicaciones.

Programación reactiva

Hacemos aquí una pausa para introducir otro concepto relacionado con los observables, como es la "programación reactiva". Aunque para hablar de programación reactiva existen libros enteros, vamos a explicar muy por encima sobre lo que se trata.

Programación tradicional

Primero establezcamos una base sobre un conocimiento de la programación tradicional que nos parece obvio, pero que es la base sobre la programación reactiva, que tiene que ver con el flujo de ejecución de las instrucciones.

En programación tradicional las instrucciones se ejecutan una detrás de otra. Por tanto, si realizamos un cálculo con dos variables y obtenemos un resultado, aunque las variables usadas para hacer el cálculo cambien en el futuro, el cálculo ya se realizó y por tanto el resultado no cambiará.

```
let a = 1;
let b = 3;
let resultado = a + b; // resultado vale 4
// Más tarde en las instrucciones...
a = 7; // Asignamos otro valor a la variable a
// Aunque se cambie el valor de "a", resultado sigue valiendo 4,
```

El anterior código ilustra el modo de trabajo de la programación tradicional y la principal diferencia con respecto a la programación reactiva. Aunque pueda parecer magia, en programación reactiva la variable resultado habría actualizado su valor al alterarse las variables con las que se realizó el cálculo.

Programación reactiva y los flujos de datos

Para facilitar el cambio de comportamiento entre la programación tradicional y la programación reactiva, en ésta última se usan intensivamente los flujos de datos. La programación reactiva es la programación con flujos de datos asíncronos.

En programación reactiva se pueden crear flujos (streams) a partir de cualquier cosa, como podría ser los valores que una variable tome a lo largo del tiempo. Todo puede ser un flujo de datos, como los clics sobre un botón, cambios en una estructura de datos, una consulta para traer un JSON del servidor, un feed RSS, el listado de tuits de las personas a las que sigues, etc.

En la programación reactiva se tienen muy en cuenta esos flujos de datos, creando sistemas que son capaces de consumirlos de distintos modos, fijándose en lo que realmente les importa de estos streams y desechando lo que no. Para ello se dispone de diversas herramientas que permiten filtrar los streams, combinarlos, crear unos streams a partir de otros, etc.

Como objetivo final, reactive programming se ocupa de lanzar diversos tipos de eventos sobre los flujos:

- La aparición de algo interesante dentro de ese flujo
- La aparición de un error en el stream
- La finalización del stream

Como programadores, mediante código, podemos especificar qué es lo que debe ocurrir cuando cualquiera de esos eventos se produzca.

Si quieres leer más sobre programación reactiva, una introducción mucho más detallada la encuentras en el artículo [The introduction to Reactive Programming you've been missing](#).

Observables y programación reactiva

El patrón observable no es más que un modo de implementación de la programación reactiva, que básicamente pone en funcionamiento diversos actores para producir los efectos deseados, que es reaccionar ante el flujo de los distintos eventos producidos. Mejor dicho, producir dichos eventos y consumirlos de diversos modos.

Los componentes principales de este patrón son:

- **Observable:** Es aquello que queremos observar, que será implementado mediante una colección de eventos o valores futuros. Un observable puede ser creado a partir de eventos de usuario derivados del uso de un formulario, una llamada HTTP, un almacén de datos, etc. Mediante el observable nos podemos suscribir a eventos que nos permiten hacer cosas cuando cambia lo que se esté observando.
- **Observer:** Es el actor que se dedica a observar. Básicamente se implementa mediante una colección de funciones callback que nos permiten escuchar los eventos o valores emitidos por un observable. Las callbacks permitirán especificar código a ejecutar frente a un dato en el flujo, un error o el final del flujo.
- **Subject:** es el emisor de eventos, que es capaz de crear el flujo de eventos cuando el observable sufre cambios. Esos eventos serán los que se consuman en los observers.

Estas son las bases del patrón. Sabemos que hemos puesto varios conceptos que sólo quedarán más claros cuando los veamos en código. Será dentro de poco. Aunque vistas muy por encima, son conceptos con los que merece la pena comenzar a familiarizarse.

Existen diversas librerías para implementar programación reactiva que hacen uso del patrón observable. Una de ellas es RxJS, que es la que se usa en Angular.

Qué es RxJS

Reactive Extensions (Rx) es una librería hecha por Microsoft para implementar la programación reactiva, creando aplicaciones que son capaces de usar el patrón observable para gestionar operaciones asíncronas. Por su parte RxJS es la implementación en Javascript de ReactiveExtensions, una más de las adaptaciones existentes en muchos otros lenguajes de programación.

RxJS nos ofrece una base de código Javascript muy interesante para programación reactiva, no solo para producir y consumir streams, sino también para manipularlos. Como es Javascript la puedes usar en cualquier proyecto en este lenguaje, tanto del lado del cliente como del lado del servidor.

La librería RxJS de por sí es materia de estudio para un curso o un manual, pero tenemos que introducirla aquí porque la usa Angular para implementar sus observables. Es decir, en vez de reinventar la rueda, Angular se apoya en RxJS para implementar la programación reactiva, capaz de mejorar sensiblemente el desempeño de las aplicaciones que realicemos con este framework.

Como habrás entendido, podemos usar RxJS en diversos contextos y uno de ellos son las aplicaciones Angular. En los próximos artículos comenzaremos a ver código de Angular para la creación de observables y de algún modo estaremos aprendiendo la propia librería RxJS.

Conclusión

Con lo que hemos tratado en este artículo tienes una base de conocimiento esencial, necesaria para dar el paso de enfrentarte a los observables en Angular.

No hemos visto nada de código pero no te preocupes, porque en el próximo artículo vamos a realizar una práctica de uso de observables en Angular con la que podrás practicar con este modelo de trabajo para la comunicación de cambios. Lo importante por ahora es aclarar conceptos y establecer las bases de conocimiento necesarias para que, a la hora de ver el código, tengas una mayor facilidad de entender cómo funciona esto de los observables y la programación reactiva.

Este artículo es obra de Miguel Ángel Álvarez
Fue publicado por primera vez en 16/ 01/ 2018
Disponible online en <http://desarrolloweb6.com/articulos/introduccion-teorica-observables-angular.html>