

RAÚL EXPÓSITO

Cómo Aprender AngularJS - Tu Sherpa de AngularJS



Estás leyendo la traducción al castellano que he hecho del artículo '*Howto Learn AngularJS - Your AngularJS Sherpa*', publicado en ng-newsletter.com. Puedes acceder al original pulsando en [este enlace](#).

Aprender AngularJS puede ser complicado. Hay una inmensa cantidad de recursos disponibles en la web. Las entradas de los blogs pueden ser contradictorias y confusas y una simple búsqueda en google puede devolverte 5 artículos describiendo la misma situación de una forma totalmente distinta.

Esta guía ha sido escrita para ser tu hoja de ruta; tu guía personal sobre cómo aprender AngularJS. Te mostraremos los conceptos de AngularJS en un orden lógico y conceptual. La intención de esta entrada es que sea la primera entrada que leas cuando empieces a aprender AngularJS. **Usa esta entrada como guía y serás un maestro de Angular en muy poco tiempo.** ¡Empecemos!

Puede que te estés preguntando, “¿qué es AngularJS?” AngularJS es una librería javascript que pone a tu disposición herramientas con las que puedes escribir aplicaciones web ricas. Aunque aparentemente existe una cantidad infinita de artículos que tratan sobre los distintos componentes de AngularJS, la esencia de Angular pueden ser descompuesta en 5 componentes principales:

- Directivas
- Controladores
- Ámbitos
- Servicios
- Inyección de Dependencias

Una vez hayas entendido estos cinco conceptos básicos aprender Angular no tendrá ningún misterio. Explicaremos cada uno de estos términos más adelante.

Pero antes de comenzar a explicar los detalles de Angular es útil preguntarse por qué deberíamos usar AngularJS. ¿Por qué no utilizar JavaScript directamente o una librería como [jQuery](#)?

Usando Javascript Directamente

Vamos a utilizar un ejemplo sencillo y a mostrar las diferencias. En este ejemplo vamos a tener un formulario de entrada que nos pida el nombre de usuario. Vamos a ir mostrando ese nombre en el HTML a medida que el usuario va escribiendo.

Usando únicamente javascript, le indicaremos a nuestro campo de entrada incluido en el formulario (`input`) que invoque a la función encargada de actualizar la salida (`updateOutput`) cada vez que se puse una tecla (lo que se detecta gracias al evento `onkeyup`). El elemento se pasa a sí mismo (`this`) como argumento en la función.



The screenshot shows the JS Bin online editor interface. It has a top navigation bar with tabs for HTML, CSS, JavaScript, Console, Output, and Help. The HTML tab is active, showing the following code:

```
<!DOCTYPE html>
<html>
<head>
<meta name="description"
content="Basic example
of binding data to an
input in jQuery" />
<script
src="http://code.jquery.
com/jquery-
2.0.3.min.js"></script>
<meta charset="utf-8">
<title>JS Bin</title>
</head>
<body>
```

The JavaScript tab is also active, showing the following code:

```
function
updateOutput(elem) {
  var name = elem.value;
  var outputElement =
document.getElementById(
"output");

  outputElement.innerHTML
= "Hello " + name;
}
```

The Output panel on the right shows the rendered HTML, which includes a text input field and the text "Hello". The input field contains the text "Enter your name:". Below the input field, the text "Hello" is displayed. At the bottom right of the Output panel, there is a checkbox for "Auto-run JS" which is checked, and a button labeled "Run with JS".

En `updateOutput` buscamos el elemento `#output` y cambiamos el `innerHTML` del elemento de salida para que tenga el valor (`value`) de nuestro campo de entrada `elem` .

Cuando diseñamos software, es muy importante tener en cuenta la forma en la que unas partes del sistema dependen de otras. En este ejemplo, `input` “sabe” a qué función tiene que invocar para que siga la ejecución y la función `updateOutput` “sabe” a qué elemento DOM tiene que enviar los datos de vuelta. Esta forma de programar hará que nuestro ejemplo deje de funcionar si cambiamos los nombres de los selectores y, lo que es peor, hará que sea realmente difícil reutilizar el código.

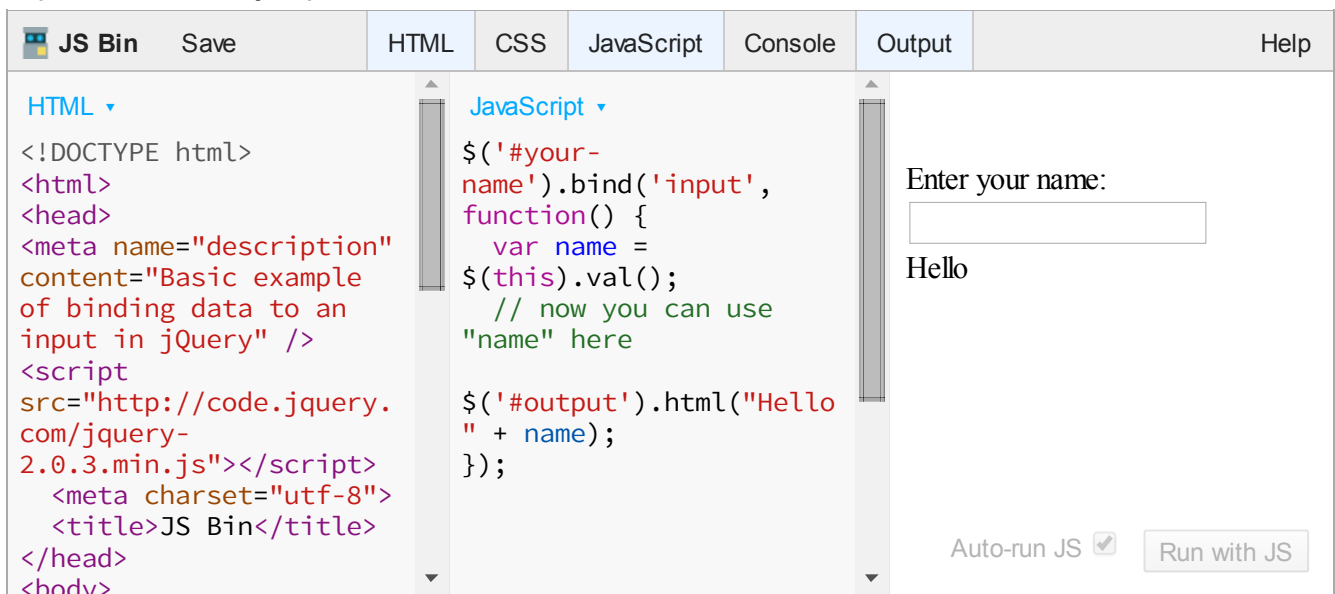
Como el código y la presentación son tan dependientes el uno del otro, decimos que tenemos un "alto acoplamiento" entre la presentación y el código, lo que hace difícil actualizar y mantener los componentes de nuestra aplicación.

UJS y jQuery

Los desarrolladores se han dado cuenta de que seguir este estilo de programación hace que los programas se vuelvan inmanejables. Es por ello que se creó el concepto de "javascript no intrusivo" (UJS por sus siglas en inglés). Una de las ideas que persigue el javascript no intrusivo consiste en lograr que el marcado de las páginas describa únicamente la estructura del documento, y no su funcionalidad.

Anecdóticamente, durante muchos años todos los que utilizaron UJS emplearon jQuery o alguna librería similar para asociar el marcado a los comportamientos. Generalmente, cuando usamos jQuery y queremos hacer un cambio en una página, usamos un "selector" para buscar un elemento en la página. Tras ello modificamos directamente el elemento y la página se actualiza.

Aquí tenemos un ejemplo:



The screenshot shows the JS Bin online editor interface. It has a top bar with tabs for HTML, CSS, JavaScript, Console, Output, and Help. The HTML tab is active, showing the following code:

```
<!DOCTYPE html>
<html>
<head>
<meta name="description"
content="Basic example
of binding data to an
input in jQuery" />
<script
src="http://code.jquery.
com/jquery-
2.0.3.min.js"></script>
<meta charset="utf-8">
<title>JS Bin</title>
</head>
<body>
```

The JavaScript tab is also active, showing the following code:

```
$('#your-
name').bind('input',
function() {
    var name =
$(this).val();
    // now you can use
    "name" here

$('#output').html("Hello
" + name);
});
```

The Output panel on the right shows the rendered HTML, which includes a text input field and the text "Hello". Below the output, there are checkboxes for "Auto-run JS" (checked) and a "Run with JS" button.

En este ejemplo buscamos a través del árbol DOM y encontramos el elemento cuyo identificador es `#your-name` y, tras ello, le asociamos un comportamiento al evento `input`. Si queremos mostrar el nombre necesitaremos un lugar donde ubicarlo, así que buscamos en el árbol DOM de nuevo y encontramos el elemento con el identificador `#output`. Tras ello cambiamos su contenido para que sea `"Hello " + name`.

Esta forma de manipular directamente los elementos del árbol DOM se conoce como estilo "imperativo".

La idea de que el marcado deba ser utilizado para describir la estructura del documento y no su funcionalidad es interesante. Convierte el problema del acoplamiento bidireccional entre la presentación y el código en un acoplamiento unidireccional en aquellos lugares donde hayas

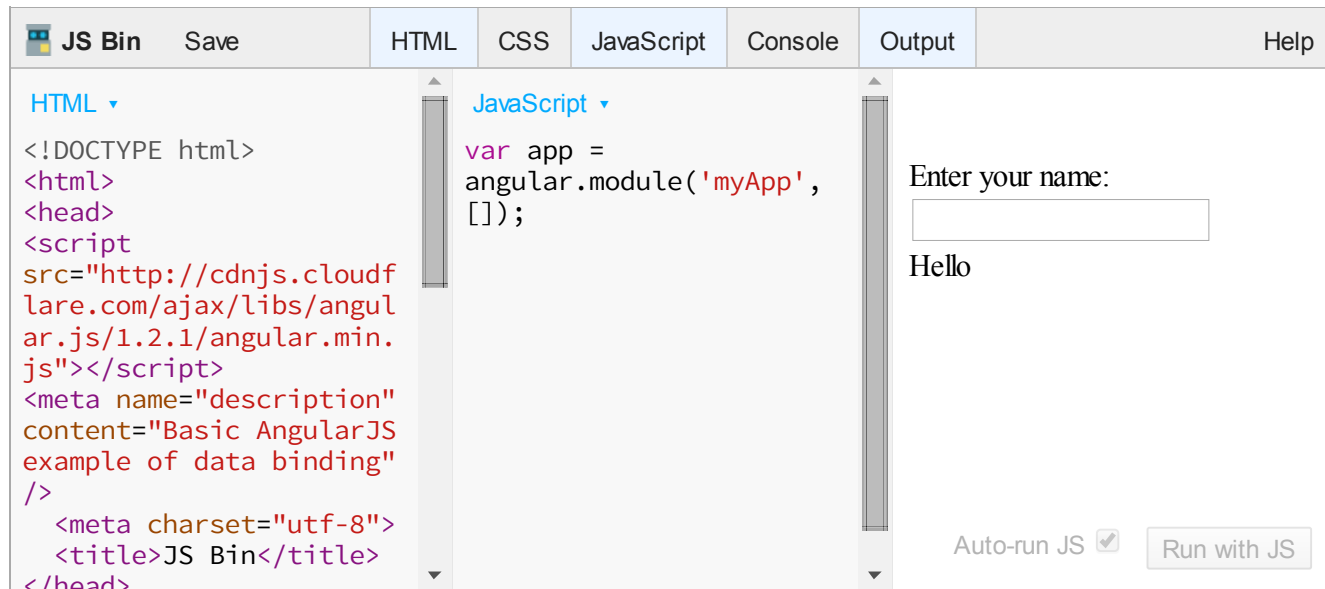
especificado programáticamente los identificadores o las clases CSS.

Además, hay muchos casos en los cuales es útil usar marcado para definir comportamientos. Por ejemplo, cuando se utilizan plantillas es común indicar “muestra esta variable aquí” o “repite esta lista de elementos aquí”. Definiendo *algunos* comportamientos mediante el marcado podemos dejar claro con facilidad cómo se utilizará la presentación. Este beneficio es positivo siempre y cuando no introduzcamos un acoplamiento innecesario.

Uno de los mayores problemas cuando se construyen aplicaciones web con jQuery es que pasamos mucho tiempo lidiando con el marcado. El marcado simplemente define cómo ver nuestros datos así que, ¿no sería estupendo si nuestro código sólo se encargase de los datos y la presentación se actualizase ella sola?

Ejemplo con AngularJS

Esta es la clase de problemas que AngularJS trata de resolver. Vamos a implementar nuestro ejemplo usando AngularJS.



The screenshot shows the JS Bin online editor interface. It has tabs for HTML, CSS, JavaScript, Console, Output, and Help. The HTML tab is active, showing the following code:

```
<!DOCTYPE html>
<html>
<head>
<script
src="http://cdnjs.cloudflare.com/ajax/libs/angular.js/1.2.1/angular.min.js"></script>
<meta name="description"
content="Basic AngularJS example of data binding"
/>
<meta charset="utf-8">
<title>JS Bin</title>
</head>
```

The JavaScript tab is also active, showing the following code:

```
var app =
angular.module('myApp',
[]);
```

The Output tab shows the rendered HTML, which includes a text input field and the text "Hello". The text input field is labeled "Enter your name:" and contains the text "Hello".

At the bottom right of the editor, there is a checkbox labeled "Auto-run JS" which is checked, and a button labeled "Run with JS".

En este ejemplo le estamos indicando a Angular que queremos crear una aplicación llamada `myApp`. En la etiqueta `input` enlazamos el valor de `input` a la variable `name` mediante el atributo `ng-model`. Para mostrar el valor de `name` en la presentación, ponemos `{{name}}` en el HTML (también conocido como hacer *binding* del valor de `name` en la presentación). Angular procesa las etiquetas “mustache” (`{{}}`) como si fueran una plantilla y se encarga de mantener el valor actualizado si la etiqueta `input` cambia.

Es importante observar que en la versión de Angular no estamos buscando elementos por un selector para manipular directamente el HTML. En su lugar, estamos *declarando* dónde deberían incluirse las variables para que Angular se encargue de las labores de fontanería por nosotros.

La forma de angular de desarrollar aplicaciones dice que nosotros nos encarguemos de los datos que Angular ya se encarga de manipular el árbol DOM.

Esta forma de cambiar elementos en el árbol DOM sigue lo que se conoce como programación "declarativa".

¿Qué son las directivas de AngularJS?

Para poder trabajar con Angular necesitamos entender el concepto de *directivas*.

Para poder entender bien las directivas podemos usar una analogía. Piensa por un momento en cómo se crean las páginas web: escribimos etiquetas HTML y cada etiqueta define un comportamiento diferente. Cuando usamos las etiquetas `` y `` le estamos indicando al navegador que queremos utilizar una lista sin ordenar. Cuando usamos las etiquetas `<video>` o `<audio>`, le estamos diciendo al navegador que queremos que muestre un video o reproduzca una pista de audio.

Pero la cantidad de etiquetas HTML que el navegador sabe interpretar es finita. ¿Qué ocurre si queremos una etiqueta `<weather>` que nos muestre el clima?, ¿Qué hay de un tag `<login>` que muestre a nuestro usuario el panel de autenticación? *Las directivas nos dan el poder de crear nuestros propios elementos HTML.* Esto es, las directivas nos permiten crear elementos DOM (o atributos) personalizados y especificar su comportamiento.

Una *directiva* es simplemente una función que devuelve un elemento DOM para darle funcionalidad extra. En Angular usaremos directivas *por todas partes*.

Es importante darse cuenta de un detalle: cuando miramos el marcado de una aplicación desarrollada con Angular, todas las etiquetas `ng-` son directivas facilitadas por el propio Angular.

Por ejemplo, podemos usar la directiva `ngClick` para indicarle a un elemento que debe ejecutar una función cuando se haga click sobre él. Por ejemplo:

```
1 <button ng-click="runWhenButtonClicked()">Haz click sobre mí</button>
2 <!-- En otro elemento -->
3 <div ng-click="runWhenDivClicked()">Mejor haz click sobre mí</div>
```

En el ejemplo anterior, el atributo HTML `ng-click` es una *directiva* que añade a un *escuchador de eventos* el evento de hacer click sobre el `<button>` y el `<div>`. Hay un **montón** de *directivas* *incuidas en Angular*.

Cuando hablamos de directivas, tenemos que usar el nombrado "lowerCamelCase" (p.e. `ngClick`) para poder referenciar a la directiva. Esto es así porque en el código fuente de Angular hay una función definida llamada `ngClick`. Sin embargo, cuando usamos una directiva en el HTML usamos el nombrado "kabob" (p.e. `ng-click`). Básicamente: `ngClick` y `ng-click` hacen referencia a lo mismo, AngularJS automáticamente hace la conversión de uno al otro. Puede ser un poco confuso al principio, pero la idea es que permite que cada código luzca mejor en su contexto.

Las directivas son lo que hacen de Angular tan poderoso y son una parte de lo que hace que Angular sea *declarativo*. Angular viene de serie con un montón de directivas útiles que nos permitirán mostrar u ocultar elementos, enviar y validar formularios, actuar sobre eventos de tipo click, etc.

Por ejemplo, podemos mostrar u ocultar elementos utilizando las directivas `ngShow` y `ngHide`:

```
1 <a ng-click="shouldShowDiv = !shouldShowDiv">Show content below</a>
2 <div ng-show="shouldShowDiv">Este div sólo se mostrará si <em>shouldShowDiv</em> es true</div>
3 <div ng-hide="shouldShowDiv">Este div se ocultará <em>shouldShowDiv</em> no es true</div>
```

La comunidad de Angular *ha desarrollado varias directivas* para muchos tipos de funcionalidades diferentes.

Es sencillo escribir nuestras propias directivas. Hemos escrito *una entrada detallada en la que explicamos cómo crear nuestras propias directivas*.

Para resumir: **Las directivas hacen posible extender HTML con funcionalidades propias.**

Aprende más sobre directivas

- [Nuestra detallada entrada sobre cómo hacer directivas propias.](#)
- [La documentación oficial sobre directivas.](#)
- [Un directorio de directivas desarrolladas por la comunidad.](#)

Expresiones

Cuando usamos directivas en nuestro marcado pasamos, además, argumentos a nuestras directivas. Como vimos en el ejemplo anterior la utilidad en `ng-click="shouldShowDiv = !shouldShowDiv"` de la directiva `ngClick` es establecer el valor de una variable. Angular recupera el argumento de la directiva y lo evalúa casi como si fuera JavaScript.

Estas expresiones nos dan mucha flexibilidad y capacidad con nuestras directivas. Podemos establecer variables, invocar funciones, leer variables, hacer operaciones matemáticas, etc. De hecho, podemos usar las expresiones en cualquier lugar. También podemos usarlas en las propias plantillas:

```
1 <div ng-init="count = 0">
2   <a ng-click="count = count + 1">Add one</a>
3   <h3>Double the count is: {{ 2 * count }}</h3>
4 </div>
```

Cómo arranca Angular

Es importante tener un modelo mental claro sobre qué hace angular cuando se cargan nuestras páginas. Cuando incluimos el javascript de Angular dentro de nuestras páginas HTML, este enlazará un método al evento `DOMContentLoaded` (esto significa que Angular se ejecutará cuando toda la página se haya terminado de cargar). Cuando la página se ha cargado, Angular visitará cada uno de los elementos de tu documento HTML (es decir, recorrerá cada elemento del árbol DOM). Si Angular visita un elemento que contiene una directiva, le adjuntará un comportamiento

La primera directiva que Angular necesita encontrar es la directiva `ngApp` . Angular asociará nuestra aplicación con el elemento donde encuentre la directiva `ngApp` .

Por ejemplo, si nuestra página HTML es la siguiente:

```
1  <!doctype html>
2  <html>
3    <head>
4      <title>Bare HTML page</title>
5    </head>
6    <body>
7      <div id="app" ng-app>
8        <h1>Hello from Angular</h1>
9      </div>
10     <footer>
11       &copy; Fullstack.io, LLC.
12     </footer>
13   </body>
14 </html>
```

Angular encontrará la directiva `ngApp` ubicada en el elemento `<div>` . La aplicación Angular se cargará sobre el elemento `<div>` .

¿Por qué es importante “sobre qué” elemento esté nuestra aplicación? Angular funciona adjuntándose a sí mismo sobre elementos específicos y **sólo dará funcionalidad a sus elementos hijos**.

Esto significa que una aplicación de Angular no puede establecer el `<title>` de la página. **No puede** modificar un elemento que sea padre suyo o hermano. En el ejemplo anterior, esto significa que nuestra aplicación hecha con angular no puede cambiar el `<footer>` , ni el `<head>` ni ningún otro elemento que esté fuera del elemento `<div id="app" ng-app>` .

Esto significa que *podemos* usar otros frameworks aparte de Angular. Esta característica hacer que sea fácil introducir Angular en subcomponentes de nuestras páginas.

Con frecuencia queremos definir un nombre a nuestra aplicación angular en lugar de utilizar el que viene por defecto con `ngApp` . Podemos declarar que queremos usar un módulo personalizado pasándole un nombre como argumento a la directiva `ngApp` .

Esto es, podemos cambiar el `div` anterior de

```
1 <div id="app" ng-app>
```

por

```
1 <div id="app" ng-app='myApp'>
```

Lo cual nos permitirá referenciar el módulo `myApp` en nuestro código javascript.

En lugar de tener un `div` vacío con una directiva `ngApp` como teníamos antes, podemos tener esto:

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>Bare HTML page</title>
5   </head>
6   <body>
7     <div id="app" ng-app='myApp'>
8       <h1>Hello from Angular</h1>
9     </div>
10    <footer>
11      &copy; Fullstack.io, LLC.
12    </footer>
13  </body>
14 </html>
```

Ahora, Angular buscará la aplicación de Angular llamada `myApp` y lanzará un error para hacernos saber que no puede encontrarla.

Aprende más sobre `ngApp`

- [Documentación de Angular sobre ngApp.](#)

- [La ubicación de la directiva ng-app - html vs body.](#)

Módulos

Una vez que tenemos nuestro código escrito en javascript, ¿cómo le decimos a Angular dónde está nuestra aplicación escrita en angular?. Necesitamos indicarlo usando el API

`angular.module()` . Podemos usar dos operaciones principales cuando trabajamos con un módulo:

1. *la creación de un módulo nuevo* (es decir, el “setter”) u
2. *obtener la referencia a un módulo existente* (es decir, el “getter”)

Cuando queramos crear un módulo usaremos estos argumentos en el setter:

```
1 angular.module('myApp', []);
```

Cuando queramos recuperar un módulo ya existente usaremos estos argumentos en el getter (es decir, no usaremos argumentos):

```
1 angular.module('myApp');
```

La única diferencia entre ambos es que el setter recibe un array como segundo parámetro. Hablaremos sobre el segundo parámetro más adelante.

Una vez que hemos creado el módulo `myApp` podremos ubicar nuestra aplicación en un lugar en concreto de nuestro código usando la directiva `ng-app` . Por ejemplo, si tenemos el siguiente HTML:

```
1 <html ng-app="myApp">
2 <head>
3 <!-- etc. -->
4 </head>
5 <body>
```

```
6 <!-- etc. -->
7 </body>
8 </html>
```

`myApp` es básicamente el módulo “principal” de la página y la aplicación arrancará automáticamente por nosotros cuando la página se haya mostrado.

Un módulo, en la terminología de Angular, no es más que un conjunto de objetos que definen una funcionalidad concreta.

Podemos pensar que un coche es una colección de módulos. Desde el punto de vista del conductor el coche es un vehículo que funciona girando el volante y pisando pedales. El coche está formado por módulos independientes como cinturones de seguridad, motor, ruedas, etc. que trabajan de forma conjunta realizando tareas individuales.

Debido a que la funcionalidad de cada componente es limitada, es fácil probarlos de forma independiente y que cada empresa pueda fabricar sus propios componentes. Por ejemplo, el fabricante de las ruedas no necesita saber cómo funcionan los frenos. Puede centrarse simplemente en fabricar neumáticos que sean seguros. La rueda puede ser probada de forma independiente e "incluida" como un componente más del coche.

Los módulos de Angular funcionan de una forma similar. Podemos incluir distintos componentes de una aplicación de Angular para hacer una aplicación con Angular. Podemos escribir pequeños componentes que sean fácilmente probables y mantenibles para conseguir que el objetivo que quieren alcanzar sea obvio.

Aprende más sobre módulos:

- [Documentación de Angular sobre módulos](#)
- [directorio con módulos 'ngModules'](#)

Ámbitos

El siguiente concepto que necesitamos entender para aprender Angular es el concepto de

ámbitos (scope). AngularJS utiliza ámbitos para comunicarse entre componentes - particularmente entre nuestro javascript y nuestro HTML. Los ámbitos son el *pegamento* que une nuestro código y lo que muestra el navegador.

Por ejemplo, imagina que queremos mostrarle un mensaje de bienvenida al usuario. Podemos usar la variable `$scope` en nuestro código JavaScript para hacer que la variable `user` sea accesible desde nuestra vista, el HTML:

```
1 angular.module('myApp') // <-- This is the getter function for a module previously defined
2 .run(function($rootScope) {
3   $rootScope.user = {
4     email: 'ari@fullstack.io'
5   }
6 });
```

Ahora, el HTML de nuestra aplicación angular puede acceder a la variable `user` desde la vista y *enlazarla* en nuestra presentación.

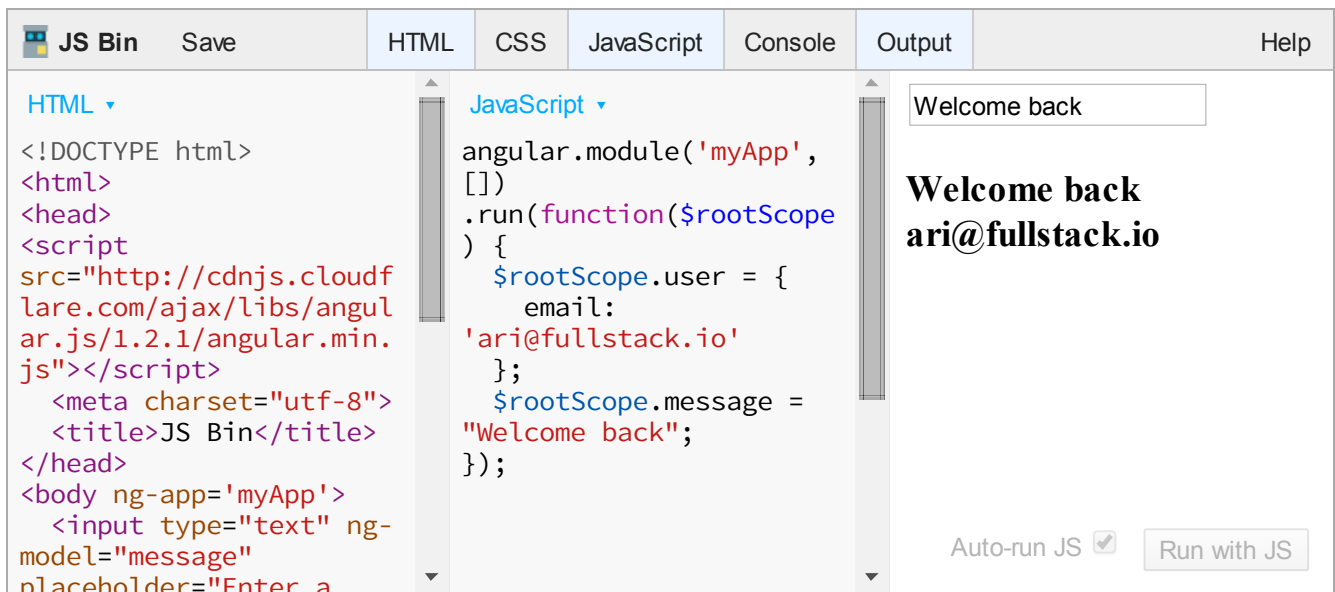
```
1 <span>Welcome back {{ user.email }}</span>
```

*Cuando decimos **enlazarla** en Angular, nos referimos a cualquier valor que se esté mostrando en la vista. Esto ocurre principalmente con datos que se muestran en la vista usando la sintaxis `{{ }}`. En el caso de que te lo estés preguntando, esta sintaxis funciona utilizando una directiva (la directiva `ng-bind`)*

Cuando la vista se carga, nuestra aplicación Angular buscará la variable `user` en el objeto `$scope` y mostrará su atributo `email`. Este es un concepto muy potente: creamos una variable `user` en el `$rootScope` y angular hace que esta variable sea accesible automáticamente desde nuestra vista.

Un detalle importante a tener en cuenta es que este enlace **funciona en ambos sentidos**.

Nuestra vista puede cambiar el valor de una variable. Por ejemplo, podemos escribir un mensaje en una caja de texto (usando la directiva `ngModel`) y esto cambiará el mensaje de nuestro JavaScript. Trata de escribir en el `input` de este ejemplo:



The screenshot shows the JS Bin application interface. It has a top bar with tabs for HTML, CSS, JavaScript, Console, Output, and Help. The HTML tab is active, showing the following code:

```
<!DOCTYPE html>
<html>
<head>
<script
src="http://cdnjs.cloudflare.com/ajax/libs/angular.js/1.2.1/angular.min.js"></script>
<meta charset="utf-8">
<title>JS Bin</title>
</head>
<body ng-app='myApp'>
<input type="text" ng-model="message"
placeholder="Enter a
```

The JavaScript tab is also active, showing the following code:

```
angular.module('myApp',
[])
.run(function($rootScope) {
  $rootScope.user = {
    email:
'ari@fullstack.io'
  };
  $rootScope.message =
"Welcome back";
});
```

The Output panel on the right shows the rendered HTML, which includes a text input field with the value "Welcome back". Below the input field, the text "Welcome back ari@fullstack.io" is displayed. At the bottom right of the Output panel, there are two buttons: "Auto-run JS" (checked) and "Run with JS".

En el ejemplo anterior, estamos guardando el objeto `user` dentro del objeto `$rootScope`. Cuando cambiamos el valor de la caja de texto `input`, el valor de `user.email` cambia en nuestro `$rootScope`. Como `$rootScope` está asociado con nuestra vista, Angular actualiza la vista automáticamente.

Angular tiene muchos ámbitos

Una característica poderosa (aunque también confusa) de Angular es que puedes tener **muchos ámbitos** en tu aplicación. El ámbito `$rootScope` es el ámbito **de mayor nivel** en el contexto de nuestra aplicación.

Esto significa que desde cualquier lugar de la vista (es decir, en todos los elementos hijos que hay debajo del elemento DOM con la directiva `ngApp`) se pueden referenciar variables que se encuentren en el objeto `$rootScope`.

Sin embargo, si abusamos del objeto `$rootScope`, podemos pecar de dejar **demasiada** información en un único ámbito. Dejar todas las variables en un único ámbito hace que actúen como variables globales; en una aplicación grande y compleja acabaremos teniendo problemas para gestionar los datos y el código se hará confuso. Para evitar esto, Angular tiene la capacidad de organizar ámbitos mediante relaciones padres/hijos.

¿Qué significa el signo del `$dólar`? AngularJS utiliza el signo del dólar `$` como prefijo de muchas funciones y objetos que vienen con Angular. El `$` se utiliza para identificar

fácilmente palabras reservadas de Angular. No utilices un prefijo `$` cuando nombres tus servicios porque puede que entres en conflicto con la propia librería.

Al igual que los elementos DOM se incluyen unos dentro de otros, **los ámbitos también pueden incluirse unos dentro de otros**. Del mismo modo que las etiquetas HTML pueden tener una etiqueta padre, los ámbitos también pueden tener padre. Cuando Angular busca el valor de una variable mirará tanto en el ámbito actual como en los ámbitos ascendentes.

Angular crea ámbitos en muchas situaciones. Por ejemplo, se crea un ámbito hijo para cada *controlador* de nuestra aplicación.

Todavía no hemos hablado sobre los controladores, pero lo haremos en la próxima sección.

Pero antes de comenzar con los controladores es necesario señalar algo que quizá no sea obvio: un ámbito es un objeto plano javascript (también conocido como POJO). Aunque un ámbito no tenga una funcionalidad que lo haga realmente útil, no sale de la nada. Los ámbitos son objetos javascript como el resto de cosas de nuestro programa.

Para resumir: **los ámbitos son una forma de organizar pares atributo-valor sin contaminar un espacio de nombres global**.

Aprende más sobre ámbitos:

- [Nuestro artículo sobre ámbitos y controladores.](#)
- [Documentación de Angular sobre `\$scope`.](#)
- [La explicación de Angular de `\$scope`.](#)
- [Construye tu propio Angular con ámbitos.](#)

Controladores en Angular

Mientras que las directivas se usan generalmente en un único elemento del árbol DOM, Angular utiliza el concepto de *controladores* como un elemento con el que organizar *grupos* de elementos del árbol DOM.

Un *controlador* es un trozo de código que define la funcionalidad de una parte de la página.

Echemos un vistazo a este ejemplo:

```
1 angular.module('myApp')
2   .controller('HomeController', function($scope) {
3     // Tenemos acceso a este nuevo
4     // objeto $scope, donde podemos colocar
5     // datos y funciones para poder interactuar con él
6   });
```

Ahora que tenemos el controlador definido, podemos ubicarlo en la página usando la directiva `ngController` de este modo:

```
1 <div ng-controller='HomeController'>
2   <!--
3   Aquí tenemos acceso al objeto
4   $scope definido en el HomeController
5   -->
6 </div>
```

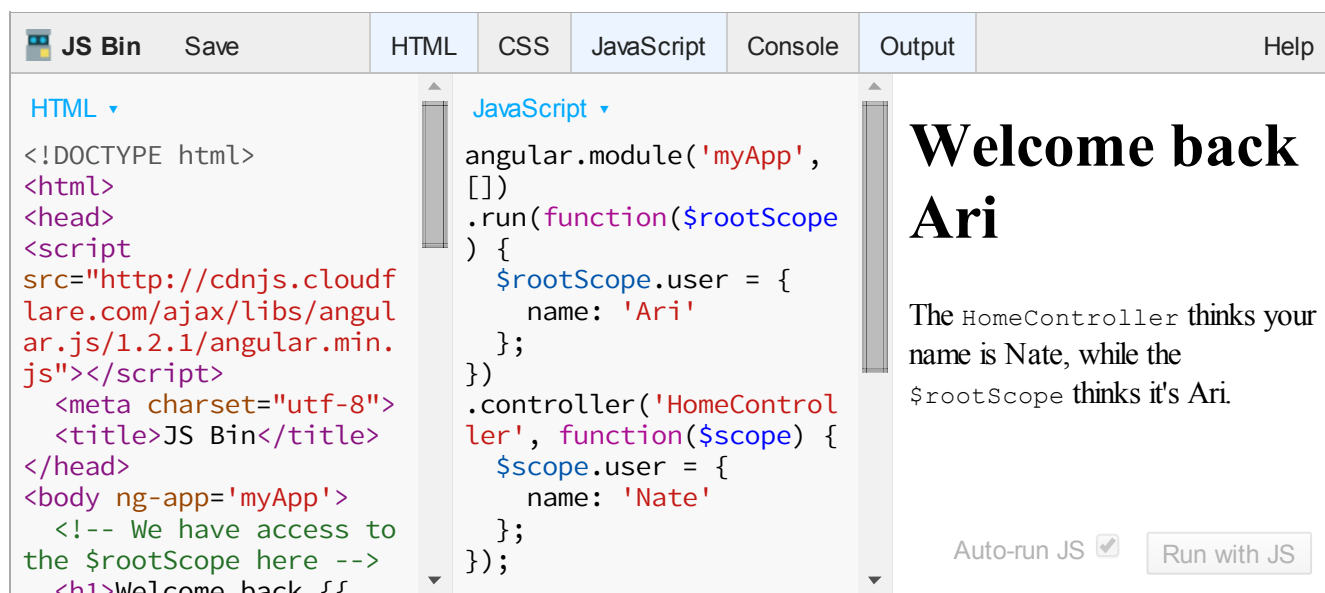
En lugar de colocar toda nuestra funcionalidad en el objeto `$rootScope`, podemos colocarla en el objeto `$scope` de `HomeController` y mantener nuestro `$rootScope` limpio.

El beneficio de crear un ámbito nuevo es que nos permite ser capaces de mantener las variables y los datos ubicados en una parte específica de la página. De este modo podemos definir una variable nueva sin contaminar o entrar en conflicto con otras partes de la página.

Técnicamente, cada vez que creamos un nuevo controlador, Angular crea un nuevo objeto `$scope` por debajo del ámbito que se encuentre encima suyo. (En este caso `$rootScope`).

Utilizamos el término “por debajo” porque los ámbitos están jerarquizados. Cuando Angular trata de buscar el valor de una variable, si no la encuentra en un ámbito la buscará en el ámbito superior en la jerarquía (hasta que alcance `$rootScope`) con la única excepción de los ámbitos aislados (no cubrimos *ámbitos aislados*, pero para aprender más, echa un vistazo a nuestras entradas sobre directivas donde lo explicamos con mucho detalle).

Por ejemplo:



The screenshot shows the JS Bin editor interface. The top bar includes tabs for HTML, CSS, JavaScript, Console, Output, and Help. The HTML panel contains the following code:

```
<!DOCTYPE html>
<html>
<head>
<script
src="http://cdnjs.cloudflare.com/ajax/libs/angular.js/1.2.1/angular.min.js"></script>
<meta charset="utf-8">
<title>JS Bin</title>
</head>
<body ng-app='myApp'>
  <!-- We have access to the $rootScope here -->
  <h1>Welcome back {{
```

The JavaScript panel contains the following code:

```
angular.module('myApp',
[])
.run(function($rootScope) {
  $rootScope.user = {
    name: 'Ari'
  };
})
.controller('HomeController', function($scope) {
  $scope.user = {
    name: 'Nate'
  };
});
```

The Output panel displays the rendered HTML:

Welcome back Ari

The HomeController thinks your name is Nate, while the \$rootScope thinks it's Ari.

At the bottom of the Output panel, there are buttons for "Auto-run JS" (checked) and "Run with JS".

Resumen: **Un controlador es una porción de código que une modelos y vistas usando ámbitos.**

Aprende más sobre controladores

- [Artículo en LosTechnies sobre controladores.](#)

Patrón Modelo-Vista-Controlador

Angular funciona con el patrón de diseño Modelo-Vista-Controlador (también conocido como MVC).

Si no estás familiarizado con MVC, la idea principal es que el *modelo* contiene los datos que forman el núcleo de nuestra aplicación (por ejemplo, un coche). La *vista* es la interfaz del usuario que se muestra ante el usuario (por ejemplo, la página que muestra el coche), y el *controlador* es el código que mantiene ambas partes juntas.

En nuestra aplicación Angular, el controlador es el responsable de actualizar el modelo. El controlador puede "hablar" con la vista, lo que en Angular se consigue a través del objeto `$scope`.

La vista es la interfaz que el usuario ve y con la cual interactúa. En Angular **la vista es el HTML**. Como usuarios sólo interactuamos con la vista y Angular se encarga de modificar el objeto `$scope`.

Recursos

- [Artículo de la wikipedia sobre MVC](#).
- [Coding Horror: Entendiendo MVC](#).
- [La guía de Martin Fowler sobre patrones de diseño que interactúan con el usuario, la cual incluye MVC](#).

\$http, XHR, and Promesas

Una vez que hemos entendido cómo interactúan los datos dentro de nuestra aplicación, el siguiente paso lógico es obtener datos del mundo real y utilizarlos en nuestra aplicación.

¿Cómo podemos interactuar con APIs de backend con Angular?

Angular viene con un recubrimiento sobre `XMLHttpRequest` (también conocido como `XHR`, que es lo que permite tener `AJAX`) invocando a `$http`. El objeto `$http` es una librería que nos ayuda a hacer peticiones HTTP y procesar la respuesta. A menudo usaremos `$http`.

```
1  $http({
2    method: 'GET',
3    url: 'http://foo.com/v1/api',
4    params: {
5      api_key: 'abc'
6    }
7  });
```

Cuando se ejecuta este método se hace una petición GET contra el backend en el servidor ubicado en `http://foo.com/v1/api` con el parámetro `api_key=abc`.

Las peticiones XHR son *asíncronas*. Esto significa que nuestra petición no tiene que parar la

ejecución de nuestra aplicación y esperar una respuesta del servidor. La ventaja de ello es que el usuario puede seguir utilizando la aplicación mientras se hace la petición HTTP, pero introduce problemas ya que hemos de manejar la respuesta una vez que los datos vuelvan del servidor. Debido a esto, el flujo del control asíncrono puede volverse complicado si no contamos con las herramientas adecuadas.

Por lo general, tenemos dos opciones

- Pasar una función que actúe como *callback*. Esta función se invocará cuando finalice la petición HTTP.
- Utilizar una *promesa*. Esta es la aproximación que sigue Angular.

Promesas

Las promesas son objetos que nos ayudan a trabajar con código asíncrono sintiendo que estamos escribiendo código síncrono. Angular utiliza ampliamente las promesas, así que es importante que te familiarices con cómo utilizarlas.

Generalmente usamos sólo tres métodos cuando utilizamos promesas:

```
1  promise
2  .then(function(data) {
3    // Invocado cuando no hay errores con los datos
4  })
5  .catch(function(err) {
6    // Invocado cuando surge un error
7  })
8  .finally(function(data) {
9    // Invocado siempre, independientemente del resultado obtenido como respuesta
10 })
```

Cuando tenemos una promesa, el método `.then()` se invocará cuando tengamos una respuesta sin fallos, el método `catch()` se invocará cuando se produzca un error, y el método `finally()` se invocará independientemente del resultado de la función.

El objeto `$http` devuelve una promesa cuando se completa una petición XHR. Una vez hecho esto, para interactuar con nuestra petición, simplemente usaremos la función `.then()` para cargar los datos en nuestro objeto `$scope` :

```
1  var promise = $http({
2    method: 'GET',
3    url: '/v1/api',
4    params: {
5      api_key: 'abc'
6    }
7  });
8  promise.then(function(obj) {
9    // obj es la petición en bruto generada por Angular
10     // y contiene códigos de estado, los datos en bruto, encabezados,
11     // y la función de configuración utilizada para hacer la petición
12     $scope.data = obj.data;
13  });
```

Usando \$http

Ahora que sabemos cómo usar el objeto `$http`, podemos interactuar con nuestro controlador del modo siguiente:

```
1  angular.module('myApp', [])
2  .controller('HomeController', function($scope, $http) {
3    $http({
4      method: 'GET',
5      url: '/v1/api',
6      params: {
7        api_key: 'abc'
8      }
9    }).then(function(obj) {
10      $scope.data = obj.data;
11    });
12  });
```

Este ejemplo es sólo a modo de demostración. No uses el objeto `$http` dentro de un controlador, es mejor utilizarlo en un servicio. Veremos los servicios en breve.

Pero espera... ¿Cómo accedemos al objeto `$http` en el controlador?

```
1 angular.module('myApp', [])
2 .controller('HomeController', function($scope, $http) {
3     // Tenemos el objeto $scope y el objeto $http
4     // disponibles en este punto
5 });
```

Aprende más sobre Promesas y XHR

- Regístrate en ng-book.com para obtener un capítulo de ejemplo del libro ng-book que trata sobre promesas.
- [Artículo en ng-newsletter's sobre XHR.](#)

Inyección de Dependencias

La *Inyección de Dependencias* (DI) es un término relacionado a cómo se hace referencia a las dependencias desde nuestro código. La inyección de dependencias, como el uso de `require` en Node.js, `require` en Ruby, o `import` en Java hace referencia a cómo los objetos obtienen las dependencias que necesitan para ejecutarse correctamente. Por ejemplo, no necesitamos escribir la función `printf()` en C porque podemos usar la librería `libc` que ya la implementa. Esta librería `libc` se considera una *dependencia*.

La pregunta es, ¿cómo obtenemos esas dependencias que necesitamos para poder ejecutar nuestro código?

A medida que un proyecto se va volviendo más complejo vamos necesitando organizar nuestro código en módulos. De algún modo necesitaremos incluir todos esos módulos en nuestro código final.

La situación es más complicada en el código en el lado del cliente porque, a menudo, cargamos código de distintos archivos. En el navegador, cuando cargamos varios archivos de distintos orígenes, éstos se recuperan en un orden impredecible. Sin las herramientas adecuadas, gestionar este proceso puede ser doloroso.

Por suerte AngularJS ha tenido esto en cuenta - la inyección de dependencias es la solución a estos problemas.

“La inyección de dependencias” es el cómo le decimos Angular qué dependencias necesitamos utilizar para que Angular nos las facilite cuando las necesitemos.

Por ejemplo, para el controlador siguiente, necesitamos acceder tanto al objeto `$scope` como al servicio `$q`. Estamos diciendo (es decir, *anotando*) qué necesitamos para hacer funcionar nuestro controlador. De este modo, en tiempo de ejecución Angular gestionará (*inyectar*á) las dependencias por nosotros.

```
1 angular.module('myApp', [])
2   .controller('HomeController', function($scope, $q) {
3     // Tenemos Los objetos $scope y $q
4     // disponibles en este punto
5   });
```

Le hemos pedido a Angular que inyecte el objeto `$scope` con anterioridad en este artículo, pero nunca habíamos explicado qué significaba. Cuando especificamos que `$scope` es un argumento de la función `controller`, le estamos diciendo a Angular que haga que el objeto `$scope` esté disponible para que lo podamos usar. Especificar las dependencias usando los argumentos de una función es un patrón habitual en Angular.

Aquí tienes algunas cosas que necesitas saber sobre la inyección de dependencias:

- Los nombres deben coincidir con objetos existentes.

Angular inspecciona los argumentos de la función e infiere que la variable “`$scope`” coincide con el servicio `$scope`. Si tratamos de llamarlo `$myScope`, no funcionará.

- El módulo que quieres utilizar tiene que ser *solicitado* desde nuestro módulo.

Si quieres utilizar un módulo personalizado desde otro módulo personalizado debes usar la **sintaxis del setter en la función `module` y solicitarlo en el segundo argumento**. He aquí un ejemplo:

```
1
2 // imagina que tenemos algunos servicios...
3 angular.module('fullstack.services', [])
4   // algo de código que define los servicios aquí
5   .service('WeatherService', function() {
6     // implementa el servicio aquí
7   });
8
9
10 // y queremos usar estos servicios en nuestros controladores
11 angular.module('fullstack.controllers', ['fullstack.services'])
12   // aquí tenemos el servicio WeatherService disponible
13   );
```

El motivo por el cual podemos utilizar `$scope` y `$q` sin indicar que son dependencias se debe a que Angular las trata de forma distinta.

- **El orden no importa**

Podemos utilizar

```
1 .controller('HomeController', function($scope, $q) {
```

o

```
1 .controller('HomeController', function($q, $scope) {
```

que ambas funcionarán.

- **No puedes tener dependencias cíclicas**

Si lo intentas obtendrás un error.

- **Hay una sintaxis alternativa.**

Este es un truco algo avanzado, pero lo queremos compartir de todos modos para que puedas reconocerlo en caso de que te lo encuentres.

Cuando llega el momento de poner la aplicación en producción, “mininizar” el javascript es una práctica común. Este proceso de minimizar cambia el nombre de las variables y puede estropear la habilidad que tiene Angular de inferir nombres de variables.

Hay una sintaxis alternativa que permite lidiar con este problema, la cual se conoce con el nombre de “notación inline”. Este es el aspecto que tiene:

```
1      .controller('HomeController', ['$scope', '$q', function($scope, $q) {  
2          // ...  
3      }]);
```

Observando el ejemplo se puede ver que, en lugar de pasar una función al `controlador`, se le pasa un array. Los argumentos de este array son todos los nombres de los módulos que se le quieren inyectar. El último argumento es la función que define el controlador. El controlador verá cómo se le inyectan los módulos **en ese mismo orden**.

Esta sintaxis es un poco complicada así que, en la práctica, mucha gente usa `ng-min` como un paso de precompilación. `ng-min` convierte la sintaxis “que infiere” en la “notación inline” para preparar nuestro código para ser minimizado.

Aprende más sobre Inyección de Dependencias

- [La guía de angular sobre inyección de dependencias.](#)
- [Wiki de Angular sobre inyección de dependencias.](#)
- [Las bases de la inyección de dependencias de thinktecture.](#)
- [Entrada de Joel Hooks sobre inyección de dependencias.](#)

Servicios

Los *servicios* con otro concepto principal de AngularJS. Cuando usamos `$http` estamos usando un servicio de Angular. Los servicios son objetos `singleton` que realizan tareas comunes a varias partes del sistema.

- Usemos el servicio `$http` a modo de ejemplo: hacer peticiones HTTP no es algo propio de un controlador específico. Necesitamos hacer peticiones HTTP en muchos sitios de nuestro código.
- El colocar cookies en el navegador de usuario tampoco es una tarea propia de un controlador en concreto. Para crear cookies usaremos el servicio de angular `$cookies`.
- Imagina que estamos escribiendo una aplicación que nos informe del clima. Podemos desarrollar nuestro `WeatherService` que contenga el código en común que podemos necesitar en cualquier lugar de nuestra aplicación.

```
1 angular.module('myApp.services', [])
2 .service('WeatherService',
3   function($http) {
4     this.weatherFor = function(zip) {
5       // hacer algo con $http para obtener el clima aquí
6     };
7   });
```

Tras ello podemos usar nuestro servicio del clima en nuestro controlador:

```
1 angular.module('myApp.controllers')
2 .controller('WeatherController', function($scope, WeatherService) {
3   $scope.weather = WeatherService.weatherFor(90210);
4 });
```

Insistimos: los servicios sólo se crean *una vez*. Hay una única instancia de un servicio en concreto

en tu aplicación.

La idea es la de **mantener tus controladores tan finos como sea posible**. En el ejemplo anterior podríamos haber desarrollado `weatherFor` directamente en el controlador, pero dejándolo en el servicio aislamos responsabilidades (es decir, cada trozo de código hace una única cosa). Aparte, dejar `weatherFor` en un servicio permite que esta funcionalidad esté disponible en otros controladores.

Cuando veamos que nuestros controladores empiezan a estar hinchados es que ha llegado el momento de mover código de nuestro controlador a un servicio. No sólo es una buena práctica de programación, es que es más sencillo probar servicios cuando su código no se encuentra entremezclado con el código del resto de nuestros controladores.

¿Cuál es la diferencia entre un `servicio`, una `factoría`, y un `proveedor`? A medida que veas más código escrito con Angular podrás ver que estos tres términos son, a menudo, intercambiables. Eso es porque son lo mismo. Los `servicios` y las `factorías` están implementadas por `proveedores`. La diferencia está en el nivel de configuración que tienes cuando creas cada una de ellas. Puedes ver un ejemplo concreto de la diferencia [mirando aquí](#) o [aquí](#). Para una explicación más detallada, [mira aquí](#). De momento vamos a pensar que son lo mismo.

Aprende más sobre servicios de Angular

- [Entrada en ng-newsletter sobre servicios.](#)
- [Entrada en ng-newsletter sobre los distintos tipos de servicios.](#)
- [Entendiendo los servicios de Angular](#)
- [Angular.js: ¿servicio vs proveedor vs factoría?](#)

Testing

Una de las mayores ventajas de utilizar Angular es que está diseñado para que puede ser probado desde todos los ángulos. Recomienda mucho que haya una separación funcional entre componentes y facilita las pruebas de integración. Puedes usar la inyección de dependencias de

Angular y el soporte de pruebas de Angular con el potente y popular framework Jasmine.

A pesar de que no vamos a explicar cómo realizar pruebas en este artículo (hay un montón de enlaces fantásticos al final de esta sección), es una buena idea entender la terminología para saber por dónde comenzar.

Cuando creamos aplicaciones con angular hay dos tipos de pruebas que podemos realizar:

Pruebas unitarias

Las pruebas unitarias cubren pequeñas unidades atómicas de funcionalidad tomando como punto de referencia el código fuente. Con los tests unitarios crearemos nuestros objetos de angular por nuestra cuenta y haremos afirmaciones de cómo estos objetos deben responder e interactuar con otros componentes.

En las pruebas unitarias nos centramos menos en la funcionalidad de la aplicación como un conjunto y nos centramos más en cómo los objetos interactúan con otros a nivel individual. Por ejemplo, no nos preocupa que el usuario pulse sobre el botón de login y que la función de login se ejecute, sólo comprobaremos que la función de login se comporte como esperamos.

Un ejemplo de cómo realizar un test unitario sobre un controlador de Angular que tiene la única función de mostrar la página de login (por ejemplo) puede ser este:

```
1      'use strict';
2
3      describe('Controller: HomeController', function () {
4
5          // carga el módulo del controlador
6          beforeEach(module('myApp'));
7
8          var HomeCtrl, scope;
9
10         // Inicializa el controlador y un ámbito falso
11         beforeEach(inject(function($controller, $rootScope) {
12             scope = $rootScope.$new();
13             HomeCtrl = $controller('HomeController', {
14                 $scope: scope
15             }));
```

```
16     }));
17
18     it('should have a showLogin function', function () {
19         expect(typeof(scope.showLogin)).toBe("function");
20     });
21 });
```

Testing End-to-End (también conocido como testing E2E)

Por otro lado, al testing End-to-End no le importa *cómo* funcione la función de login, simplemente espera que se comporte como quiere el usuario de la aplicación. El testing End-to-end nos hace suponer que somos los usuarios de la aplicación y nos da la habilidad de definir cómo interactúa el usuario con la página de forma programática. El testing E2E nos libera de la necesidad de tener que hacer click por toda la página cada vez que hacemos un cambio para asegurarnos de que todo *todavía funciona*.

En el testing E2E controlamos directamente un navegador, así que ahí se reflejarán nuestras pruebas. Un ejemplo de test end-to-end (usando protractor) tiene el siguiente aspecto:

```
1 describe('page navigation', function() {
2     var link;
3     beforeEach(function() {
4         link = element(by.css('.header ul li:nth-child(2)'));
5         link.click();
6     });
7
8     it('should navigate to the /about page when clicking', function() {
9         expect(browser.getCurrentUrl()).toMatch(/\/about/);
10    });
11
12    it('should add the active class when at /about', function() {
13        expect(link.getAttribute('class')).toMatch(/active/);
14    });
15
16 });
```

Herramientas de testing

Definimos testing como el proceso de establecer expectativas que la funcionalidad de nuestro código debe satisfacer bajo ciertas condiciones. **Cuando hacemos pruebas, nos estamos centrando en confirmar que el código que se está ejecutando se comporta como debe bajo las condiciones que le hayamos definido.**

JasmineJS

Cuando realizamos pruebas sobre nuestras aplicaciones en angular usamos el framework de testing JasmineJS para definir nuestras pruebas y expectativas. Jasmine es un framework con el que escribir pruebas de comportamiento e incluye un conjunto de funcionalidades que nos permiten definir cómo se debería ejecutar el código. Nos da la función `describe()` con la que definir las pruebas y las expectativas del test que queramos realizar.

Karma

Karma es el entorno de pruebas que usamos para establecer un entorno de testing mientras desarrollamos aplicaciones con angular. Es un ejecutor de pruebas que ejecutará nuestros tests, dándonos un flujo de ejecución completo (incluyendo la ejecución de los tests al detectarse que los ficheros han cambiado) y permitiéndonos escribir tests con distintos frameworks. Trabaja muy bien con frameworks de integración continua y nos permite depurar el código en el navegador.

Protractor

El miembro más nuevo en el núcleo de herramientas de testing. Protractor es un framework de pruebas desarrollado sobre la librería WebDriverJS, la cual funciona sobre Selenium.

Recursos

- [JasmineJS](#).
- [Protractor](#).
- [Selenium](#).
- [Artículo muy detallado sobre testing en YearOfMoo](#).

Copyright © 2014 Raúl Expósito

