



ANGULAR<sub>(2)</sub>

**Formador:** Vladimir Bataller

**Requisitos:**

- HTML y CSS
- Javascript.

## Características

Angular 2 es una completa redefinición de AngularJS.

Mientras que la web de AngularJS (1.x) era <https://angularjs.org>, la url de Angular (2) es:

<https://angular.io/>

Angular JS (1.x) seguirá vigente y creando nuevas versiones para dar soporte a aplicaciones existentes. En un futuro se crearán herramientas para actualizar aplicaciones de la versión 1 a la 2 o incluso hacer aplicaciones en las que convivan ambas versiones.

Las aplicaciones Angular favorecen la claridad del código y el desarrollo rápido.

Angular está desarrollado en Typescript y para crear aplicaciones Angular también se recomienda su uso.

Angular 2 introduce la inyección de dependencias *Lazy*, no hace falta que todos los módulos estén cargados.

## Historia

- El empleado de Google Miško Hevery empezó el diseño de AngularJS como un proyecto personal en 2009.
- La version 1.0 de AngularJS se libero en 2012, mientras que Angular (2) se liberó en septiembre de 2016.
- El Proyecto tuvo mucha aceptación y actualmente es un proyecto oficial Google.

¿Por qué?

A continuación se describen algunas limitaciones que presentaba AngularJS 1.x para el desarrollo de aplicaciones empresariales de gran escala.

**Tipado dinámico:** Angular 1 empleaba EcmaScript 5 y por lo tanto empleaba tipado dinámico que podía producir errores más complicados de detectar.

**Sobrecarga:** el navegador se podía ver muy sobrecargado por la carga de muchas bibliotecas y vistas de la aplicación.

**SEO:** conseguir que las aplicaciones AngularJS mantubieran una compatibilidad SEO era un aspecto complejo.

**Primera visita:** la primera visita a la aplicación era lenta, para resolver esto había que realizar una serie de trabajos de desarrollo adicionales.

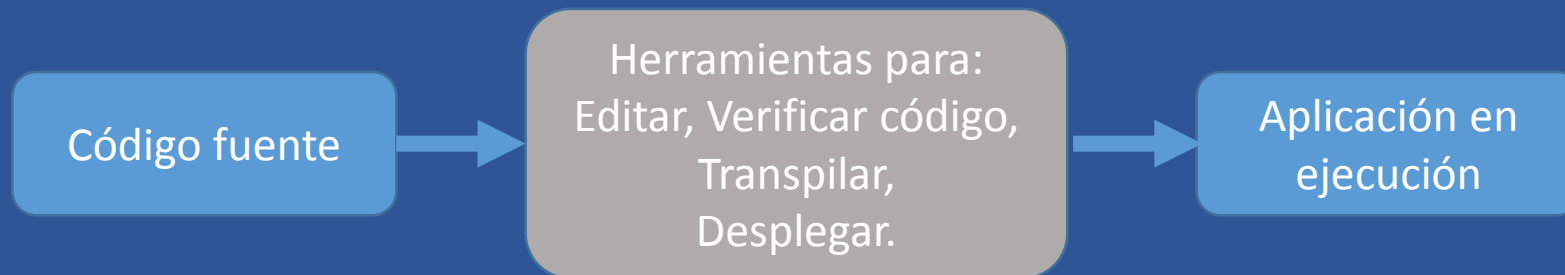
## Características

**Enlace de datos:** se han introducido los mecanismos necesarios para que el enlace de datos entre la vista y el controlador sea mucho más rápida y escalable.

**Componentes:** los componentes tienen todavía mayor importancia, y no pueden acceder al \$scope.

## Forma de trabajo

La forma de trabajo ya no consiste en descargar el javascript de Angular e incluirlo en el html. Ahora el proceso de generación de una aplicación tiene un flujo de trabajo más sofisticado y profesionalizado que está basado en una serie de herramientas.



## Configuración del proxy de NPM

Se puede introducir una configuración diferente para http y https.

```
npm config set proxy http://proxy.indra.es:8080
```

```
npm config set https-proxy http://proxy.indra.es:8080
```

Crear una nueva  
aplicación

Para generar la estructura de una nueva aplicación se emplea la herramienta **@angular/cli (Common Line Interface)** cuya url es la siguiente: <https://cli.angular.io>, aunque se puede instalar mediante npm.

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

```
ng new nombreAplicacion
```

```
cd nombreAplicacion
```

```
ng serve
```

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

Para editar el código, se puede emplear cualquier editor, pero se recomienda una de las siguientes:

- Visual Studio Code
- Atom

## Archivos generados

A continuación se describen algunos de los archivos generados por Angular CLI.

**package.json:** es el archivo de configuración general de npm donde se indican por ejemplo las dependencias de la aplicación que se está desarrollando. Entre esas dependencias, además de las del propio angular, está *systemjs* que permite cargar archivos js dinámicamente sin emplear la etiqueta *script* de html. También permite incluir scripts que se empleen con frecuencia en el ciclo de trabajo, por ejemplo *start* para iniciar la aplicación.

**dist:** es la carpeta en la que se ubica todo lo que se subirá al servidor.

**src:** es la carpeta en la que se ubica el código fuente desarrollado. En esta carpeta se habrá generado un *index.html* y un *main.ts* que serán los archivos, respectivamente de HTML y Typescript, iniciales de nuestra aplicación.



## Quickstart

Una opción alternativa para crear la estructura base de una aplicación Angular 2 mínima es emplear el *quickstart*. A continuación se enumeran los pasos a seguir.

1.- Descargar desde la siguiente url y descomprimir el archivo descargado:

**<https://github.com/angular/quickstart/archive/master.zip>**

2.- En una consola ubicarse en el directorio en el que se ha descomprimido el archivo y borrar los archivos innecesarios ejecutando:

**for /f %i in (non-essential-files.txt) do del %i /F /S /Q**

**rd .git /s /q**

**rd e2e /s /q**

3.- Descargar las dependencias del proyecto (indicadas en package.json) ejecutando el siguiente comando npm:

**npm install**

4.- Arrancar el proyecto (compila, arranca el servidor y abre el navegador) mediante el siguiente comando npm:

**npm start**

5.- Tras el paso anterior en el navegador en localhost:3000 se mostrará *"Hello Angular"*.

## Ocultar archivos no fuente

El volumen de archivos con el que se trabaja es alto dado que se esta viendo cada archivo \*.ts y su transpilado \*.js. Lo mismo ocurre con archivo \*.map y la carpeta node\_modules. A continuación se indica como ocultar todo no innecesario.

En Visual Studio Code ir al menú Archivo -> Preferencias -> Configuración y seleccionar la carpeta *Configuración de area de trabajo*. Allí insertar la siguiente configuración:

```
{
  "files.exclude": {
    "**/.git": true,
    "**/.DS_Store": true,
    "**/*.js": {"when": "$(basename).ts"},
    "**/*.js.map": true,
    "**/node_modules/": true
  }
}
```

Nota: esto se acabará guardando en la carpeta `[Usuario]\AppData\Roaming\Code\User` en el archivo `settings.json`.

## Carga de Angular en el navegador

Angular emplea System.js como cargador de paquetes. Mediante System.js se carga el archivo (que llamaremos main.js y que será el resultado de transpilar el archivo main.ts) que arranca la infraestructura de Angular en el navegador.

```
<!DOCTYPE html><html><head> ...  
  <script src="node_modules/core-js/client/shim.min.js"></script>  
  <script src="node_modules/zone.js/dist/zone.js"></script>  
  <script src="node_modules/systemjs/dist/system.src.js"></script>  
  <script src="systemjs.config.js"></script>  
  <script>  
    System.import('main.js').catch(function(err){ console.error(err); });  
  </script>  
</head>  
<body>  
  <my-app>Loading AppComponent content here ...</my-app>  
</body></html>
```

main.ts

En el archivo principal, importa y se arranca el módulo principal de la aplicación (en este caso AppModule). Para ello se emplea la función *platformBrowserDynamic* la cual hay que importarla también.

```
// archivo src/main.ts
```

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
```

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

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

## Módulos

Los módulos permiten agrupar varios componentes para gestionar las dependencias de la aplicación.

```
// archivo src/app/app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
@NgModule({
  imports: [ BrowserModule ], /*Importa otros módulos de los que depende.*/
  declarations: [ AppComponent ], /*Indica que componentes están declarados en módulo.*/
  bootstrap: [ AppComponent ] /*El elemento raíz que arranca el módulo.*/
})
export class AppModule { } /*Clase del módulo, en principio vacía.*/
```

Nota: BrowserModule tiene los servicios necesarios para iniciar una aplicación en el navegador.

## Componentes

Los componentes son el elemento central de Angular 2. Son clases a las que se les aplica el decorador `@Component` y estas clases contienen los datos, manejadores de eventos de la aplicación. Además cada componente tiene una vista asociada.

```
// archivo src/app/app.component.ts
```

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

```
@Component({
```

```
    selector: 'my-app', /*Selector html que identificará al componente*/
```

```
    template: `<h1>Hola desde Angular</h1>` /*Plantilla html que sigue el componente*/
```

```
})
```

```
export class AppComponent {
```

```
    name="Angular";
```

```
    ...
```

```
}
```

## Interpolación

Las plantillas de angular pueden incorporar expresiones cuyo resultado se agregará al html resultante de renderizar la plantilla. Esas expresiones se encierran entre dobles llaves: {{expresión}}

En las expresiones angular se pueden emplear atributos y métodos del componente asociado.

```
// archivo src/app/app.component.ts
```

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

```
@Component({  
    selector: 'my-app',  
    template: '<h1>Hola {{nombre}}, son las {{hora()}} </h1>'  
})  
export class AppComponent {  
    nombre="Pepe";  
    hora(){ return new Date(); }  
}
```

Plantillas html en archivo  
externo al componente

Insertar el código html con la propiedad *template* de `@Component`, no es muy operativo. Es mucho más interesante emplear la propiedad *templateUrl* e indicarle allí la ruta del archivo que contiene el html de la plantilla.

```
<!-- archivo src/app/app.component.html -->  
<h1>Hola {{nombre}}, desde una plantilla hmtl</h1>
```

```
// archivo src/app/app.component.ts  
import { Component } from '@angular/core';  
@Component({  
  selector: 'my-app',  
  templateUrl: 'app/app.component.html'  
})  
export class AppComponent { nombre = 'Pepe'; }
```



## [(ngModel)]

Enlace doble de controles de formulario (I)

En el caso de los controles de formulario, el enlace puede viajar en los dos sentidos del componente a la vista y viceversa. Para que esto funcione, en el módulo hay que importar el **FormsModule** de **@angular/forms**

```
// archivo src/app/app.module.ts

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';

@NgModule({
  imports: [ BrowserModule, FormsModule ], /*Importa otros módulos de los que depende.*/
  declarations: [ AppComponent ], /*Indica que componentes están declarados en módulo.*/
  bootstrap: [ AppComponent ] /*El elemento raíz que arranca el módulo.*/
})

export class AppModule { } /*Clase del módulo, en principio vacía.*/
```

## [(ngModel)]

Enlace doble de controles  
de formulario (II)

Para indicar que los datos del control viajan en las dos direcciones se incluye **ngModel** entre **corchetes** y **paréntesis**, igualándolo al nombre del atributo del controlador al cual se enlaza el control.

```
<!-- archivo src/app/app.component.html -->
<p>Nombre: <input type="text" [(ngModel)]="nombre" /></p>
<p>Apellidos: <input type="text" [(ngModel)]="apellidos" /></p>
<h1>Hola {{nombre}} {{apellidos}} !!!</h1>
```

```
// archivo src/app/app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  templateUrl: 'app/app.component.html'
})
export class AppComponent { nombre = 'Pepe'; }
```

Incluso aunque la variable no esté declarada en el Componente, el enlace lectura/escritura funciona.

Ejemplo componente  
con matriz de datos.

```
import { Component } from '@angular/core';  
  
@Component({  
    selector: 'my-app',  
    templateUrl: 'app/app.component.html'  
})  
  
export class AppComponent {  
    nombre = 'Pepe';  
    personas = [{dni:22, nombre:"Daniel", apellidos="Valiente", saldo:800},  
                {dni:32, nombre:"Sergio",apellidos="Valiente", saldo:300},  
                {dni:54, nombre:"Laura",apellidos="Villanueva", saldo:500}, ... ];  
}
```

\*ngFor

Permite repetir una etiqueta para cada uno de los elementos que haya en una matriz (por ejemplo atributo del controlador).

```
<ul>
```

```
  <li *ngFor="let persona of personas">
```

```
    <p> {{persona.nombre}}</p>
```

```
    <p> {{persona.apellidos}}</p>
```

```
  </li>
```

```
</ul>
```

\*ngIf

Permite evaluar una condición para indicar si el elemento html en el que está incluida se debe renderizar o no.

```
<ul>
```

```
  <li *ngFor="let persona of personas">
```

```
    <p> {{persona.nombre}}</p>
```

```
    <p> {{persona.apellidos}}</p>
```

```
    <p *ngIf="persona.saldo<400" style="color:red">{{persona.saldo}}</p>
```

```
    <p *ngIf="persona.saldo>=400" style="color:navy">{{persona.saldo}}</p>
```

```
  </li>
```

```
</ul>
```

Las directivas **\*ngIf** y **\*ngFor** se denominan **Directivas Estructurales** porque modifican la estructura del documento HTML.

## Template Expressions

El lenguaje empleado en las interpolaciones y otros elementos de las plantillas de Angular, se denomina *Template Expressions*. A continuación se describen algunas de sus características.

- Son expresiones parecidas a las de Javascript, pero no todas las instrucciones Javascript pueden ejecutarse.
- No se debe emplear expresiones con: `=`, `++`, `--`, `new`, `;` o `.`
- Las expresiones, por defecto se refieren a miembros (atributos y métodos) del componente asociado a la plantilla. Es decir el contexto por defecto de una plantilla es su componente.
- También puede haber variables definidas directamente en un bloque de la plantilla por ejemplo *let persona* dentro de un bloque `*ngFor`. En ese caso el contexto es el propio bloque donde se ha definido.

## Enlaces de plantilla

En una plantilla, también se pueden declarar variables de ámbito de plantilla para hacer referencia a un elemento HTML de ella.

Esto se lleva a cabo indicando el nombre que se quiere asignar a la variable precedido de #. De ese modo, dentro de la plantilla, esa variable representará al elemento html y se podrá hacer referencia a él (mediante su nombre) dentro de las expresiones de la plantilla.

Texto: `<input type="text" #texto /><br/>`

El texto introducido es: `{{texto.value}}`

Nota: en el ejemplo anterior, hasta que no se dispare un evento de cambio del componente, en la expresión no se verá el nuevo valor recibido desde el control.

## Filtros para formato

Proporcionan la posibilidad de dar formato a los datos que visualiza la aplicación. Se aplican en las expresiones poniendo su nombre a continuación del dato separado por un pipeline |

```
<ul>
```

```
<li *ngFor="let persona of personas">
```

```
<p> {{persona.nombre | uppercase}}</p>
```

```
<p> {{persona.apellidos}}</p>
```

```
<p *ngIf="persona.saldo<400" style="color:red">{{persona.saldo | currency:'EUR'}}</p>
```

```
<p *ngIf="persona.saldo>=400" style="color:navy">{{persona.saldo | currency:'EUR'}}</p>
```

```
</li>
```

```
</ul>
```



## Formateadores (Pipes)

Proporcionan la posibilidad de dar formato a los datos que visualiza la aplicación. Se aplican en las expresiones poniendo su nombre a continuación del dato separado por un pipeline |.

Los argumentos de los formateadores se indican a continuación del nombre del formateador, separados por dos puntos (:).

| Pipe      | Efecto  |
|-----------|---|
| lowercase | Muestra el texto en minúsculas.   |
| uppercase | Muestra el texto en mayúsculas.   |
| date      | Formatea objetos Date con el formato deseado suministrado como argumento.<br>Ejemplo: <code>fechaNacimiento   date : 'dd/MM/yyyy'</code>                  |
| decimal   | Formatea un número. Se le pasa un único parámetro de tipo string con el siguiente formato:<br>"minDigitosEnteros.minDigitosDecimales-maxDigitosDecimales" |
| currency  | Formatea un número. Se le pasa un unico parámetro de tipo string con el siguiente formato:<br>minDigitosEnteros.minDigitosEnteros-maxDigitosEnteros       |
| slice     | Se aplica a una matriz de string o a un string y retorna la submatriz entre las posiciones indicadas en los dos argumentos, suministrados.                |
| json      | Transforma un dato en formato json dentro de un string.   |

## Enlace de [atributos] HTML

Mediante el uso de corchetes encerrando el nombre de un atributo de un elemento HTML, se puede enlazar dicho atributo con una expresión que se indica entre comillas en el lado derecho de la igualdad.

```
<img [src]="nomArchivo + '.jpg'" />
```

## Estilos

Mediante ***styleUrls*** se indican los estilos que se aplicarán exclusivamente al componente.

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  templateUrl: 'app/app.component.html',
  styleUrls: ['app/app.component.css'] /*Matriz de archivos css aplicables al componente*/
})
export class AppComponent {
}
```

## Cambiar los estilos aplicados

En el contexto de la plantilla se dispone de un objeto llamado "class" al que se le pueden asignar atributos correspondientes a las classes de estilo aplicables. Si el valor que se asigna al atributo, la clase de estilo con ese nombre será aplicada al elemento.

```
<p [class.oscuro]="oscuro" />
```

Si a la variable *oscuro* del componente se le asigna **true**, se añadirá la clase de estilo *oscuro* al elemento. Si se le asigna el valor **false** entonces se eliminará dicha clase de estilo.

Variables de referencia  
de plantilla (#nomVar)

A los elementos de una plantilla, se les puede agregar un atributo sin valor y cuyo nombre va precedido de almoadilla (o por "ref-").

Ejemplo:

Apellidos: `<input #ape type="text" required="required"/><br/>`

El valor introducido es: `{{ape.value}} <br/>`

Sus estilos son: `{{ape.className}} <br/>`

`<button (click)="guardarCliente(ape.value)">Guardar</button>`

De este modo con dicha referencia podemos acceder a todos los atributos del elemento como si lo hubieramos obtenido con `document.getElementById`.

## Validación de formularios (I)

A continuación se muestra un formulario de ejemplo sobre el que se va a aplicar la validación. Angular automáticamente creará unos objetos con información relativa al estado de validación.

Los elementos destacados en color naranja son indispensables para la correcta configuración de la validación.

```
<form novalidate="novalidate" #f="ngForm" (ngSubmit)="enviar()">
```

```
  Dni <input type="number" [(ngModel)]="nuevo.dni" name="dni" required="required"/><br/>
```

```
  Nom <input type="text" [(ngModel)]="nuevo.nombre" name="nombre" required="required"/><br/>
```

```
  Ape: <input type="text" [(ngModel)]="nuevo.apellidos" name="apellidos" required="required"/><br/>
```

```
  <button>Insertar</button>
```

```
</form>
```

- [(ngModel)] implementa el enlace doble y facilita la lectura de los datos del formulario.
- Para la validación, es necesario que cada campo tenga asignado su atributo name, ya que, así se creará automáticamente, un objeto de tipo FormControl con la información de validación de ese campo.
- Angular creará también automáticamente un objeto NgForm para centralizar la información de validación del formulario.

## Configuración del formulario para la

```
<form novalidate="novalidate" #f="ngForm" (ngSubmit)="enviar()">
```

```
  <button [disabled]="!f.form.valid">Insertar</button>
```

```
</form>
```

## Validando un campo en concreto

Para obtener todos los detalles de la validación se debe asignar a una variable de referencia de plantilla el objeto ngModel. De este modo la variable tendrá un atributo errors que permite conocer los motivos de validaciones incorrectas.

```
Apellidos <input type="text" [(ngModel)]="nuevo.apellidos" name="apellidos" required="required"
    maxlength="6" minlength="2" #apellidos="ngModel"/>
<span *ngIf="apellidos.errors && (apellidos.dirty || apellidos.touched)">
    <span [hidden]="!apellidos.errors.required">Los apellidos son obligatorios</span>
    <span [hidden]="!apellidos.errors.maxlength">La longitud máxima es 6 caracteres</span>
    <span [hidden]="!apellidos.errors.minlength">La longitud mínima es
        {{apellidos.errors.minlength.requiredLength}} caracteres
        en lugar de {{apellidos.errors.minlength.actualLength}}</span>
</span>
```

La siguiente instrucción muestra los estilos que Angular asigna en función de estado de validación

```
<span>{{apellidos.className}}</span>
```



## Ejemplo de estilos para validación

```
form.ng-valid > p{background-color:#80ff80;padding:25px;}  
input.ng-invalid.ng-dirty{border:2px solid pink;}  
input.ng-pristine{border:2px solid blue;}  
input.ng-valid.ng-dirty{border:2px solid green;}  
input.ng-invalid.ng-dirty + span {background-color:#ff6262;color:white;font-weight:bold;  
padding-left:10px;padding-right:10px; }
```

## Router (I)

El módulo Router de Angular permite asignar urls distintas a cada sección de la aplicación aunque esta sea una aplicación de una sola página (SPA).

1.- Asegurarse que en la cabecera de la página principal (index.html) está indicada la dirección base: `<base href="/">`

2.- En el archivo del módulo principal de la aplicación, importar los siguientes objetos de `@angular/router`:

```
import { RouterModule, Routes } from '@angular/router';
```

3.- Crear una matriz JSON con la configuración de las rutas de la aplicación y los componentes que corresponden a cada una de esas rutas.

```
const routerConfig: Routes = [  
  {path:'insertar', component: InsertarComponent},  
  {path:'listado', component: ListadoComponent, data:{titulo:'Mantenimiento de clientes'}},  
  {path:'detalle/:dni', component:DetalleComponent},  
  {path:'', redirectTo:'/listado', pathMatch:'full'},  
  {path:'**', component: PaginaNoEncontradaComponent}  
];
```

## Router (II)

El módulo Router de Angular permite asignar urls distintas a cada sección de la aplicación aunque esta sea una aplicación de una sola página (SPA).

4.- Agregar al decorador `@NgModule` del módulo principal el import del `RouterModule.forRoot` con la configuración del paso anterior.

```
@NgModule({  
  imports: [  
    RouterModule.forRoot(routerConfig),  
    ...  
  ],  
  ...  
})  
export class AppModule{ }
```

## Router (II)

El módulo Router de Angular permite asignar urls distintas a cada sección de la aplicación aunque esta sea una aplicación de una sola página (SPA).

5.- Agregar a la plantilla del AppComponent el menú de navegación de la aplicación y la etiqueta `<router-outlet>` donde se renderizará el controlador de cada ruta. Los enlaces a las rutas se indican en etiquetas `<a>` mediante el atributo `routerLink`. Además el atributo `routerLinkActive` permite indicar la clase de estilo que se aplica al hipervínculo cuando para destacar que corresponde con la vista que se está mostrando actualmente.

```
<h1>Angular Router</h1>
```

```
<nav>
```

```
  <a routerLink="/listado" routerLinkActive="active">Listado de clientes</a>
```

```
  <a routerLink="/insertar" routerLinkActive="active">Insertar cliente</a>
```

```
</nav>
```

```
<router-outlet></router-outlet>
```

## Observables de rxjs/rx

Angular Framework emplea Observables en lugar de promesas para gestionar tareas asíncronas. A continuación se muestra un ejemplo ejecutable en node.js. Para ello hay que instalar la dependencia: **npm install rx --save**

```
var Rx = require("rx");  
var observable = Rx.Observable.create(observer => {  
    // La función pasada al método create inicia el proceso asíncrono. Cuando este termine lo informará  
    // ejecutando sobre el objeto observer el método onCompleted, cuando tenga algo que notificar  
    // se ejecutará el método onNext y cuando se haya producido algún error, ejecutará onError.  
    // A estos métodos se le suele pasar un dato que recibirá el observador.  
    setInterval( () => { observer.onNext("Segundo actual: " + (new Date()).getSeconds()); }, 2000);  
});  
  
// Me suscribo al observable  
function saludar(dato){ console.log("Dato:",dato);}  
observable.subscribe(saludar); // subscribe puede recibir las funciones onNext, onError y onComplete.
```

## Observables de rxjs/rx

### Ejemplo con onComplete y onNext

```
var Rx = require("rx");
var observable = Rx.Observable.create(observer => {
  setInterval(() => { var sec = (new Date()).getSeconds();
    if(sec%10==0 || sec%10==1){      observer.onCompleted();      }
    else { observer.onNext("Segundo actual: " + sec);    }
  }, 2000);
});
function saludar(dato){ console.log("Dato:",dato);}
function fin(){ console.log("Fin"); process.exit(0);}
observable.subscribe(saludar,undefined,fin);
```

## Http

La el servicio Http permite realizar peticiones asíncronas al servidor mediante el protocolo HTTP. En Angular 2, este servicio implementa su naturaleza asíncrona con Observables de RxJS (rxjs/Rx) en lugar de con Promesas como ocurría en AngularJS.

Ejemplo de clase que encapsula las peticiones http de objetos Persona.

```
@Injectable export class PersonasServicio{  
  constructor(private http:Http) { }  
  buscarPersonas(apellidos:string){  
    let url = "...";  
    return this.http.get(url).map(response => this.leerPersonas(response));  
  }  
  leerPersonas(response:Response){ return response.json().personas; }  
}
```

El método map permite indicar la transformación que se hará con los datos recibidos del servidor.

Uso del servicio:

```
servicio.buscarPersonas(apellidos).subscribe(personas => console.log(personas),error => console.log(error));
```

Al método subscribe se le pasan las funciones que reciben los datos ya procesados.

## Configuración para Http

### app.module.ts

....

```
import { HttpClientModule } from '@angular/http';

@NgModule({  imports: [ BrowserModule, FormsModule, HttpModule, HttpClientModule ],
            declarations: [ AppComponent ], bootstrap: [ AppComponent ] })

export class AppModule { }
```

### personas.servicio.ts

```
import { Injectable } from "@angular/core";
import { Http, Response } from "@angular/http";
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/catch';
import 'rxjs/add/operator/map';
```



Jasmine

## Jasmine Instalación

Jasmine es un framework de pruebas unitarias en Javascript.

Se puede descargar el ejemplo base que contiene las bibliotecas de Jasmine de la siguiente url:

<https://github.com/jasmine/jasmine/releases>

Este ejemplo incluye un archivo *SpecRunner.html* con la estructura para ejecutar las pruebas en el navegador.

Básicamente, importa:

- Las bibliotecas de Jasmine: jasmine.js, jasmine-html.js y boot.js, ubicadas en la carpeta **lib**.
- Los archivos con el código javascript que se desea probar, ubicados en la carpeta **src**.
- Los archivos con las pruebas que se desean ejecutar sobre el código a probar, ubicados en **spec**.

Ejemplo de clases a  
probar con Jasmine

En el siguiente ejemplo a probar partimos de dos clases Pagina y Libro. Un libro se compone de título y varias páginas. A continuación se muestra la declaración de la clase Pagina guardada en Pagina.js

```
function Pagina(texto) {  
    this.texto = texto;  
    this.leida = false;  
}  
  
Pagina.prototype.leer = function () {  
    this.leida = true;  
    return this.texto;  
};
```

Ejemplo de clases a  
probar con Jasmine

En el siguiente ejemplo a probar partimos de dos clases Pagina y Libro. Un libro se compone de título y varias páginas. A continuación se muestra la declaración de la clase Libro guardada en *Libro.js*

```
function Libro(titulo) {  
    this.titulo = titulo; this.paginas = [];  
}  
  
Libro.prototype.imprimir = function () {  
    var s = "#####" + this.titulo + "#####\n";  
    for (pagina in this.paginas) {  
        s += pagina.texto + "\n-----\n";  
    }  
};  
  
Libro.prototype.insertarPagina = function (texto) {  
    this.paginas.push(new Pagina(texto));  
};
```

## Ejemplo de prueba con Jasmine (1/3)

Mediante ***describe*** se declara una suite de pruebas y esta se compone de métodos ***it*** que constituyen cada una de las pruebas a realizar.

```
describe("Un libro", function() { // Con describe se declara una suite de pruebas.  
    var libro; //Variable para el objeto a probar  
    beforeEach(function() { // Se ejecuta antes de cada prueba.  
        libro = new Libro();  
    });  
    // Con "it" se declara cada prueba de la suite.  
    it("debe estar vacío justo después de crearse", function() {  
        expect(libro.paginas.length==0).toBe(true); //Aserción (afirmación)  
    });  
});
```

## Ejemplo de prueba con Jasmine (2/3)

Los describes se pueden anidar para describir un caso particular dentro del caso contenedor y que dicho caso presente varias pruebas.

// Suit anidada

```
describe("Cuando se añade una página", function() {  
    beforeEach(function() {  
        libro.insertarPagina("En un lugar de la Mancha ...");  
    });  
    it("no debe estar vacío", function() {  
        expect(libro.paginas.length).toBe(1);  
    });  
    it("se le pueden añadir varias páginas", function() {  
        libro.insertarPagina("de cuyo nombre no quiero acordarme");  
        expect(libro.paginas.length).toBe(2);  
    });  
});
```

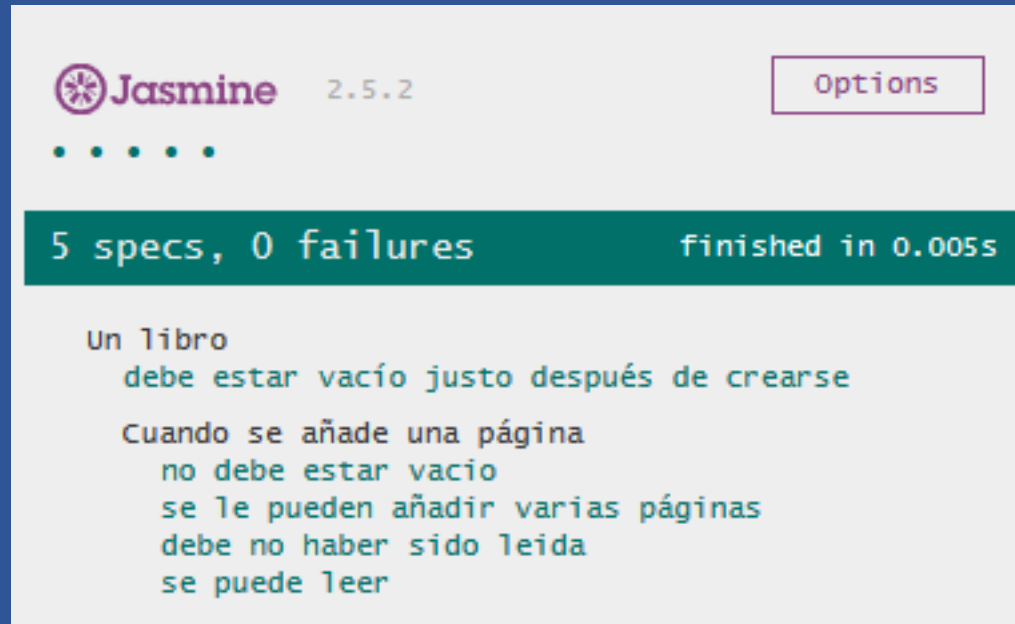
## Ejemplo de prueba con Jasmine (3/3)

```
it("debe no haber sido leida", function() {  
    expect(libro.paginas[0].leida).toBe(false);  
});  
it("se puede leer", function() {  
    libro.paginas[0].leer();  
    expect(libro.paginas[0].leida).toBe(true);  
});  
}); //fin de la suite anidada  
}); //fin de la suite principal
```

## Resultado de ejecutar Jasmine

Para ejecutar las pruebas, se debe abrir el archivo *SpecRunner.html* en el navegador.

Si el resultado es correcto, se mostrará:

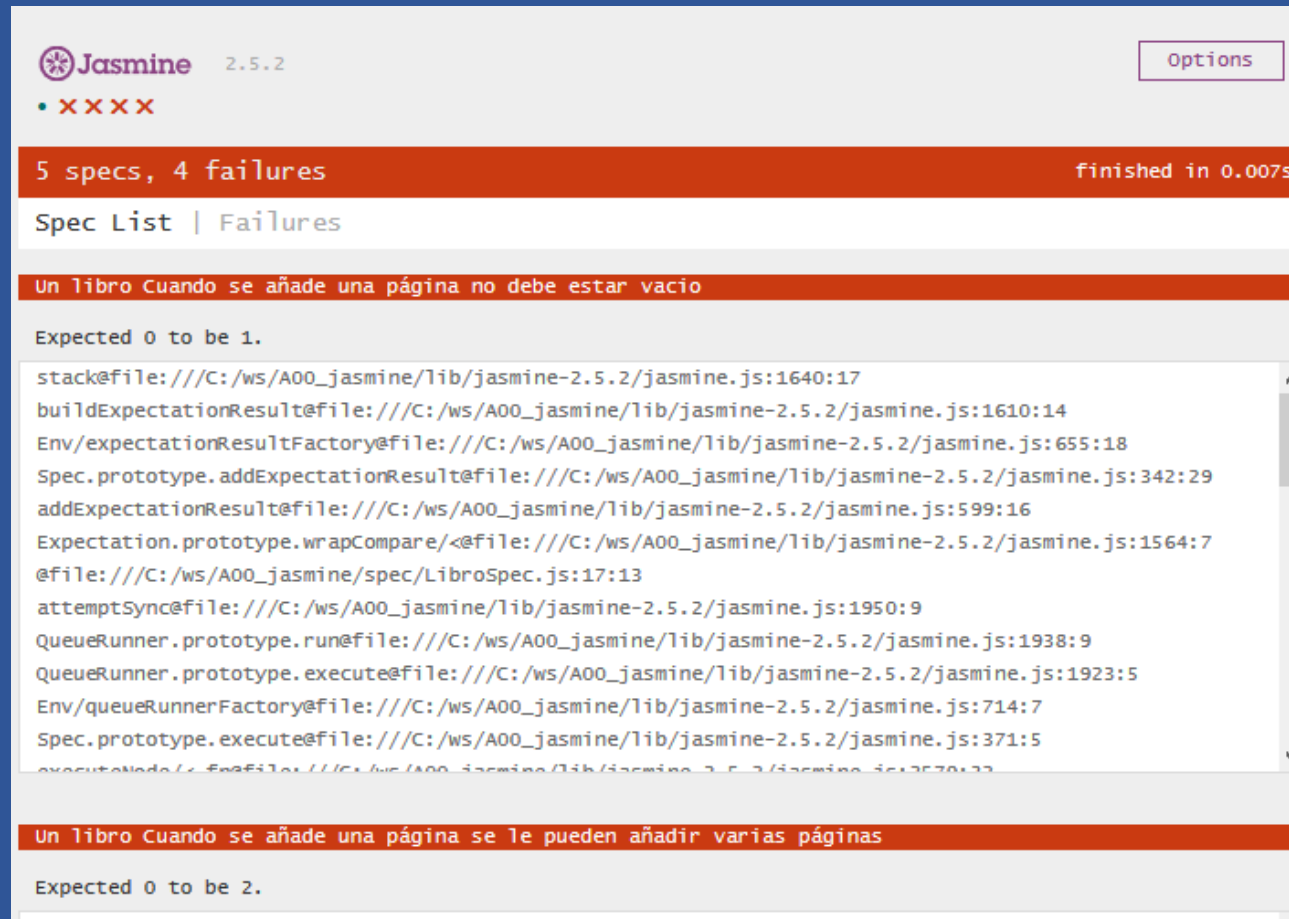




## Resultado de ejecutar Jasmine

Para ejecutar las pruebas, se debe abrir el archivo *SpecRunner.html* en el navegador.

Si la ejecución ha detectado errores, se mostrará:



The screenshot shows the Jasmine 2.5.2 test runner interface. At the top, it says "Jasmine 2.5.2" and "Options". Below that, it shows "5 specs, 4 failures" and "finished in 0.007s". There are two tabs: "Spec List" and "Failures". The "Failures" tab is selected, showing two failure messages:

- Un libro Cuando se añade una página no debe estar vacío**  
Expected 0 to be 1.  
stack@file:///C:/ws/A00\_jasmine/lib/jasmine-2.5.2/jasmine.js:1640:17  
buildExpectationResult@file:///C:/ws/A00\_jasmine/lib/jasmine-2.5.2/jasmine.js:1610:14  
Env/expectationResultFactory@file:///C:/ws/A00\_jasmine/lib/jasmine-2.5.2/jasmine.js:655:18  
Spec.prototype.addExpectationResult@file:///C:/ws/A00\_jasmine/lib/jasmine-2.5.2/jasmine.js:342:29  
addExpectationResult@file:///C:/ws/A00\_jasmine/lib/jasmine-2.5.2/jasmine.js:599:16  
Expectation.prototype.wrapCompare/<@file:///C:/ws/A00\_jasmine/lib/jasmine-2.5.2/jasmine.js:1564:7  
@file:///C:/ws/A00\_jasmine/spec/LibroSpec.js:17:13  
attemptSync@file:///C:/ws/A00\_jasmine/lib/jasmine-2.5.2/jasmine.js:1950:9  
QueueRunner.prototype.run@file:///C:/ws/A00\_jasmine/lib/jasmine-2.5.2/jasmine.js:1938:9  
QueueRunner.prototype.execute@file:///C:/ws/A00\_jasmine/lib/jasmine-2.5.2/jasmine.js:1923:5  
Env/queueRunnerFactory@file:///C:/ws/A00\_jasmine/lib/jasmine-2.5.2/jasmine.js:714:7  
Spec.prototype.execute@file:///C:/ws/A00\_jasmine/lib/jasmine-2.5.2/jasmine.js:371:5  
executeNode/@<@file:///C:/ws/A00\_jasmine/lib/jasmine-2.5.2/jasmine.js:12570:22
- Un libro Cuando se añade una página se le pueden añadir varias páginas**  
Expected 0 to be 2.  
stack@file:///C:/ws/A00\_jasmine/lib/jasmine-2.5.2/jasmine.js:1640:17

## Jasmine: matchers

Los máchers son métodos que se ejecutan sobre el valor de retorno de `expect()` para realizar verificaciones del tipo: **`expect(libro).not.toBeNull();`**

| <code>toBe(expected)</code>                      | <code>toBeUndefined()</code>                      |
|--|---|
| <code>toBeCloseTo(expected, precisionopt)</code> | <code>toContain(expected)</code>                  |
| <code>toBeDefined()</code>                       | <code>toEqual(expected)</code>                    |
| <code>toBeFalsy()</code>                         | <code>toHaveBeenCalled()</code>                   |
| <code>toBeGreaterThan(expected)</code>           | <code>toHaveBeenCalledBefore(expected)</code>     |
| <code>toBeGreaterThanOrEqual(expected)</code>    | <code>toHaveBeenCalledTimes(expected)</code>      |
| <code>toBeLessThan(expected)</code>              | <code>toHaveBeenCalledWith()</code>               |
| <code>toBeLessThanOrEqual(expected)</code>       | <code>toMatch(expected)</code>                    |
| <code>toBeNaN()</code>                           | <code>toThrow(expectedopt)</code>                 |
| <code>toBeNull()</code>                          | <code>toThrowError(expectedopt,messageopt)</code> |
| <code>toBeTruthy()</code>                        |   |

angular-mocks.js

Angular mocks es una biblioteca proporcionada por AngularJS para ayudarnos en el proceso de pruebas unitarias de las aplicaciones Angular.

Se puede descargar de la web de angularjs el archivo angular-mock.js, por ejemplo de la siguiente url:

**<https://code.angularjs.org/1.5.11/>**

Karma

## Karma Instalación

Karma es un ejecutor de pruebas (test runner) que facilita las tareas de automatización de pruebas. A continuación se indica como instalarlo para poder ejecutar pruebas definidas mediante Jasmine.

1.- Instalar globalmente las utilidades Karma de la línea de comandos (Karma-CLI):

```
npm install -g karma-cli
```

2.- Crear un *package.json* para el proyecto, mediante el asistente:

```
npm init
```

2.- En el directorio del proyecto instalar las bibliotecas de Karma:

```
npm install karma --save-dev
```

3.- Instalar las bibliotecas para ejecutar Jasmine desde karma y para el navegador deseado:

```
npm install jasmine-core karma-jasmine karma-firefox-launcher karma-chrome-launcher --save-dev
```

4.- Generación del archivo de configuración de Karma (*karma.conf.js*):

```
karma init
```

5.- Instalación de las bibliotecas del reporter "spec" que muestra más info que el "progress":

```
npm install karma-spec-reporter --save-dev
```

## Karma Ejecución

Para ejecutar las pruebas mediante Karma, se debe ejecutar el comando:

**karma start**

Al ejecutar la ejecución de Karma, en la línea de comandos, si se superan todas las pruebas, se verá algo así:

```
C:\ws\A00_karma_jasmine> karma start
29 03 2017 14:37:15.694:INFO [karma]: Karma v1.5.0 server started at http://0.0.0.0:9876/
29 03 2017 14:37:15.694:INFO [launcher]: Launching browser Firefox with unlimited concurrency
29 03 2017 14:37:15.709:INFO [launcher]: Starting browser Firefox
29 03 2017 14:37:21.407:INFO [Firefox 52.0.0 (Windows 10 0.0.0)]: Connected on socket jqvTAOqAmmjPBkY3AAAA with
id 57830999
Firefox 52.0.0 (Windows 10 0.0.0): Executed 5 of 5 SUCCESS (0.04 secs / 0.005 secs)
```

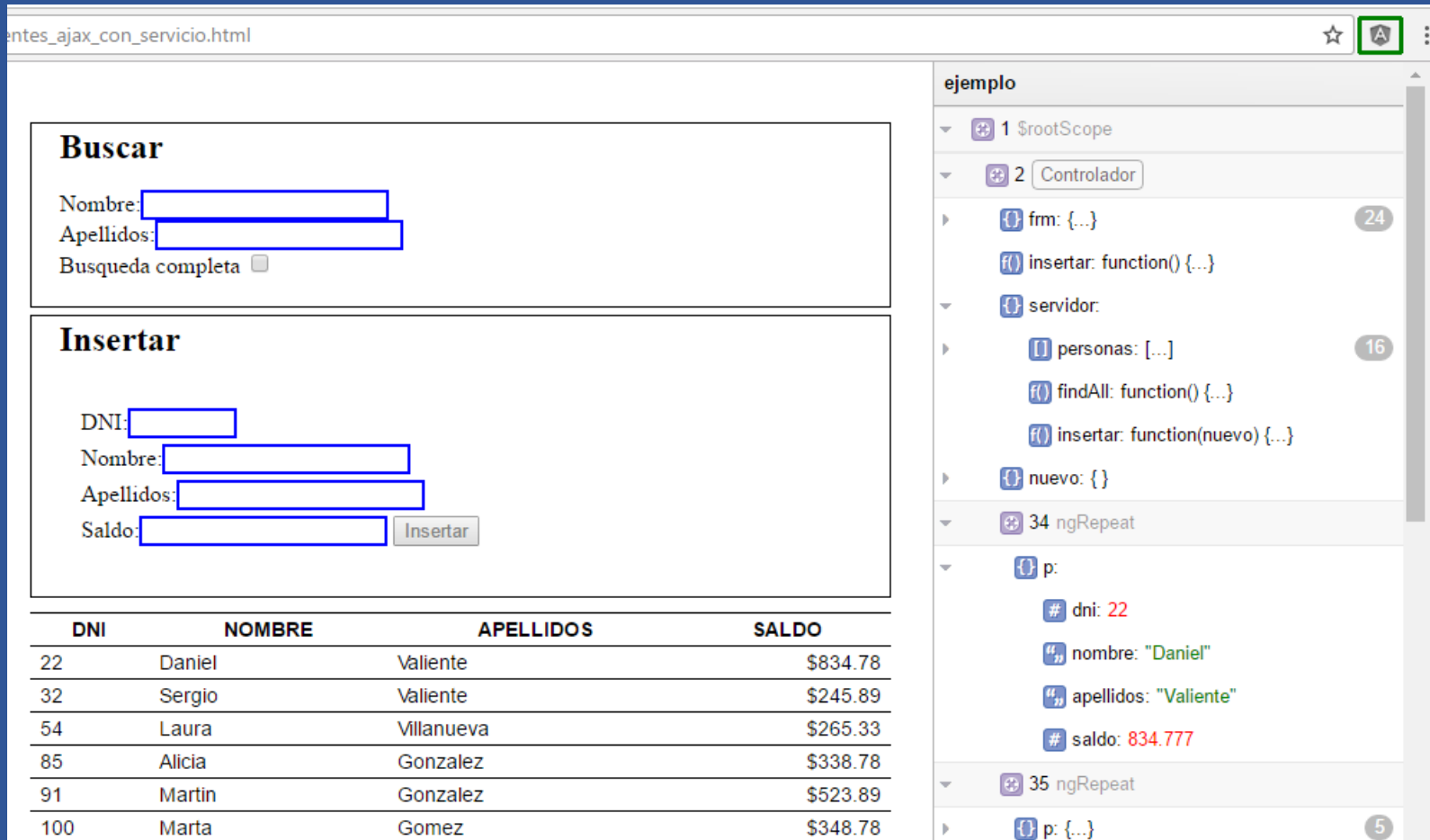
# ng-inspector

Depuración de aplicaciones AngularJS

## ng-inspector

ng-inspector es un complemento para el navegador Google Chrome que permite ver los scopes de una aplicación y como van cambiando sus valores.

Se puede instalar en Chrome, Safari y Firefox desde su página web oficial: <http://ng-inspector.org/>



ejemplo

- 1 \$rootScope
- 2 Controlador
  - frm: {...} 24
    - insertar: function() {...}
  - servidor:
    - personas: [...] 16
      - findAll: function() {...}
      - insertar: function(nuevo) {...}
  - nuevo: {}
- 34 ngRepeat
  - p:
    - # dni: 22
    - # nombre: "Daniel"
    - # apellidos: "Valiente"
    - # saldo: 834.777
- 35 ngRepeat
  - p: {...} 5

| DNI | NOMBRE | APELLIDOS  | SALDO    |
|-----|--------|------------|----------|
| 22  | Daniel | Valiente   | \$834.78 |
| 32  | Sergio | Valiente   | \$245.89 |
| 54  | Laura  | Villanueva | \$265.33 |
| 85  | Alicia | Gonzalez   | \$338.78 |
| 91  | Martin | Gonzalez   | \$523.89 |
| 100 | Marta  | Gomez      | \$348.78 |



# Protractor

Pruebas End to End (e2e)

## Protractor

Permite automatizar pruebas funcionales programandolas como test unitarios de Jasmine. Para ello emplea un Selenium server configurado con el paquete webdriver-manager. Reduce los tiempos de espera porque es Angular-aware.

Instalación de protractor (incluye webdriver-manager):

**npm install -g protractor**

Actualizar las bibliotecas de Selenium server:

**webdriver-manager update**

Arrancar Selenium server:

**webdriver-manager start**

Ejecución de las pruebas cuya configuración se haya en un archivo (archivo por defecto: protractor.conf.js)

**protractor protractor.conf.js**