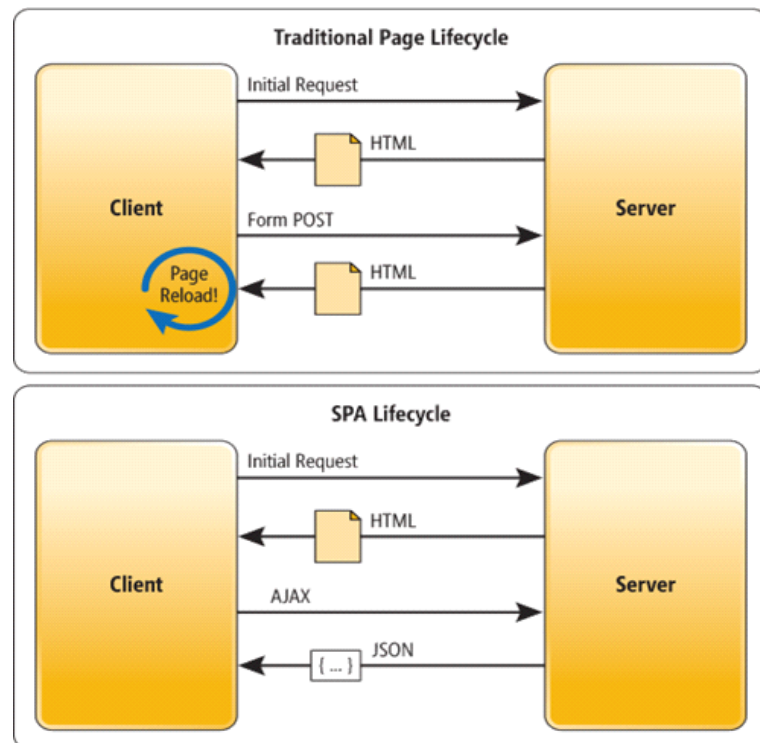


# Angular. Tercera parte

jueves, 14 de septiembre de 2017 20:31

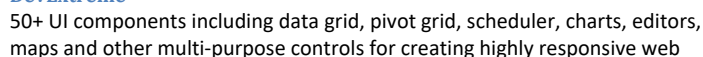
## Aplicaciones SPA

- Enrutamiento y navegación
- Acceso al servidor (Comunicaciones HTTP con APIs REST)



jueves, 14 de septiembre de 2017 23:00

- Entre las primeras en aparecer, la versión de Bootstrap realizada por Valor Software



applications for touch devices and traditional desktops.

#### **jqWidgets**

Angular UI Components including data grid, tree grid, pivot grid, scheduler, charts, editors and other multi-purpose components

#### **Kendo UI**

One of the first major UI frameworks to support Angular

#### **ng-bootstrap**

The Angular version of the Angular UI Bootstrap library. This library is being built from scratch in Typescript using the Bootstrap 4 CSS framework.

#### **ng-lightning**

Native Angular components & directives for Lightning Design System

#### **ngx-bootstrap**

Native Angular directives for Bootstrap

#### **Onsen UI**

UI components for hybrid mobile apps with bindings for both Angular & AngularJS.

#### **Prime Faces**

PrimeNG is a collection of rich UI components for Angular

#### **Semantic UI**

UI components for Angular using Semantic UI

#### **Vaadin**

Material design inspired UI components for building great web apps. For mobile and desktop.


#### **Wijmo**

High-performance UI controls with the most complete Angular support available. Wijmo's controls are all written in TypeScript and have zero dependencies. FlexGrid control includes full declarative markup, including cell templates.

<https://ng-bootstrap.github.io/#/home>

[ng-bootstrap](#) [Getting Started](#) [Components](#)


[Star](#) 3,887 [Tweet #ngbootstrap](#)



## Bootstrap 4 components, powered by Angular


Currently at v1.0.0-beta.5

[Demo](#) [Installation](#)



### Native

Angular - specific widgets built from ground and using Bootstrap 4 CSS APIs that makes sense in the Angular ecosystem. No dependencies on 3rd party JavaScript.



### Widgets

All the Bootstrap widgets (ex. carousel, modal, popovers, tooltips, tabs, ...) and several additional goodies (datepicker, rating, timepicker, typeahead).

## Components

### Accordion

Alert

Buttons

Carousel

Collapse

DatePicker

Dropdown

Modal

Pagination

Popover

Progressbar

Rating

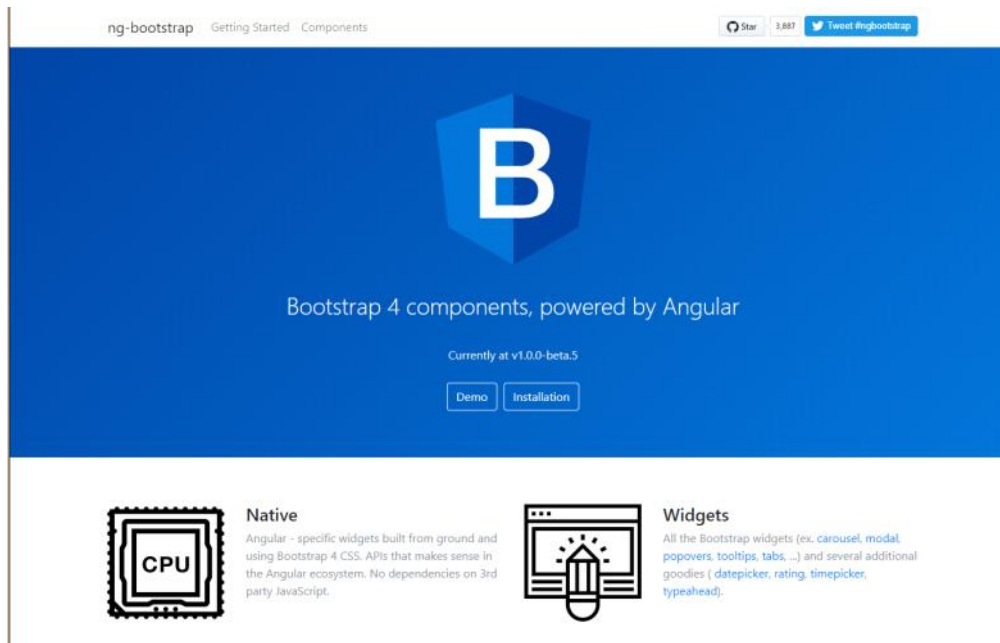
Tabs

Timepicker

Tooltip

Typeahead

<https://ng-bootstrap.github.io/#/home>



## Components

Accordion

Alert

Buttons

Carousel

Collapse

Datepicker

Dropdown

Modal

Pagination

Popover

Progressbar

Rating

Tabs

Timepicker

Tooltip

Typeahead

## Instalación

Se instalan mediante npm

```
npm install @ng-bootstrap/ng-bootstrap
```

## Configuración

Para utilizarlo, se importa en el módulo principal, ejecutando el método `forRoot()`:

```
import {NgbModule} from '@ng-bootstrap/ng-bootstrap';
```

```
@NgModule({
  ...
  imports: [NgbModule.forRoot(), ...],
```

Y se importa normalmente en otros módulos que tengan que utilizar los componentes

```
import {NgbModule} from '@ng-bootstrap/ng-bootstrap';
```

```
@NgModule({
  ...
  imports: [NgbModule, ...],
```

```
...
```

## Ejemplo de Utilización

En el Módulo `about`, se añade un componente `info`, que utiliza a su vez el componente "acordeón" de `ng-bootstrap`

Componentes

- `ngb-accordion`
- `ngb-panel`

```
<ngb-accordion #acc="ngbAccordion" activeIds="ngb-panel-0">
  <ngb-panel title="Información">
    <ng-template ngbPanelContent>
      Anim pariatur cliche reprehenderit, enim eiusmod high life accusamus terry richardson ad squid. 3 wolf moon officia aute, non cupidatat skateboard dolor brunch. Food truck quinoa nesciunt laborum eiusmod. Brunch 3 wolf moon tempor.
```

ngb-accordion

- ngb-panel

Componente de Angular

- ng-template

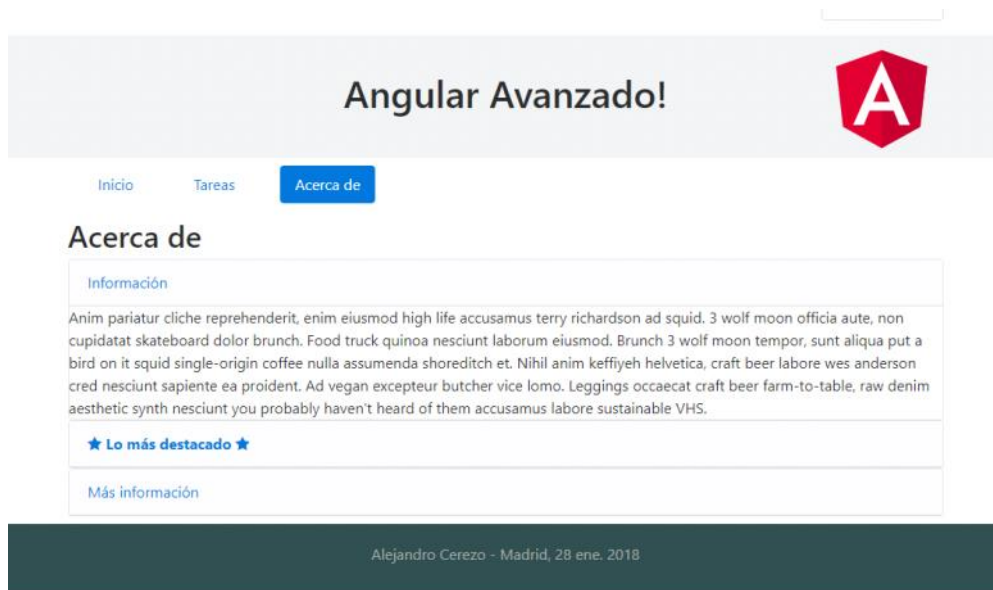
Ejemplo del uso de font awesome

```

<ng-template ngbPanelContent>
  Anim pariatum cliche reprehenderit, enim eiusmod high life accusamus terry richardson ad squid. 3 wolf moon officia aute, non cupidatat skateboard dolor brunch. Food truck quinoa nesciunt laborum eiusmod. Brunch 3 wolf moon tempor, sunt aliqua put a bird on it squid single-origin coffee nulla assumenda shoreditch et. Nihil anim keffiyeh helvetica, craft beer labore wes anderson cred nesciunt sapiente ea proident. Ad vegan excepteur butcher vice lomo. Leggings occaecat craft beer farm-to-table, raw denim aesthetic synth nesciunt you probably haven't heard of them accusamus labore sustainable VHS.
</ng-template>
</ngb-panel>

<ngb-panel>
  <ng-template ngbPanelTitle>
    <span><i class="fa fa-star" aria-hidden="true"></i>
    <b>Lo más destacado</b>
    <i class="fa fa-star" aria-hidden="true"></i>
    </span>
  </ng-template>
  <ng-template ngbPanelContent>
    Anim pariatum cliche reprehenderit, enim eiusmod high life accusamus terry richardson ad squid. 3 wolf moon officia aute, non cupidatat skateboard dolor brunch. Food truck quinoa nesciunt laborum eiusmod. Brunch 3 wolf moon tempor, sunt aliqua put a bird on it squid single-origin coffee nulla assumenda shoreditch et. Nihil anim keffiyeh helvetica, craft beer labore wes anderson cred nesciunt sapiente ea proident. Ad vegan excepteur butcher vice lomo. Leggings occaecat craft beer farm-to-table, raw denim aesthetic synth nesciunt you probably haven't heard of them accusamus labore sustainable VHS.
  </ng-template>
</ngb-panel>
<ngb-panel title="Más información">
  <ng-template ngbPanelContent>
    Anim pariatum cliche reprehenderit, enim eiusmod high life accusamus terry richardson ad squid. 3 wolf moon officia aute, non cupidatat skateboard dolor brunch. Food truck quinoa nesciunt laborum eiusmod. Brunch 3 wolf moon tempor, sunt aliqua put a bird on it squid single-origin coffee nulla assumenda shoreditch et. Nihil anim keffiyeh helvetica, craft beer labore wes anderson cred nesciunt sapiente ea proident. Ad vegan excepteur butcher vice lomo. Leggings occaecat craft beer farm-to-table, raw denim aesthetic synth nesciunt you probably haven't heard of them accusamus labore sustainable VHS.
  </ng-template>
</ngb-panel>
</ngb-accordion>

```



PRIME NG

GET STARTEDTHEMESUPPORT

Input

Button

Data

Panel

Overlay

File

Menu

Charts

Messages

Multimedia

DragDrop

Misc

The Most Complete User Interface Suite for Angular

GET STARTED

Why PrimeNG?

Congratulations! 🎉 Your quest to find the UI library for Angular is complete.

PrimeNG is a collection of rich UI components for Angular. All widgets are open source and free to use under MIT License. PrimeNG is developed by PrimeTek Informatics, a vendor with years of expertise in developing open source UI solutions. For project news and updates, please follow us on [twitter](#) and visit our [blog](#).

70+ COMPONENTS

The most complete set of native widgets featuring 70+ easy to use components for all your UI requirements.

OPEN SOURCE

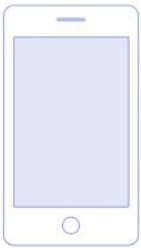
Hosted at [GitHub](#), all widgets are open source and free to use under MIT license. Feel the power of open source.

PRODUCTIVITY

Allocate your valuable time on business logic rather than dealing with the complex user interface requirements.

Parte 3 página 7

<https://material.angular.io/>



Sprint from Zero to App

Hit the ground running with comprehensive, modern UI components that work across the web, mobile and desktop.

Component Categories

Form Controls	Navigation	Layout
Buttons & Indicators	Popups & Modals	Data table



# Font Awesome

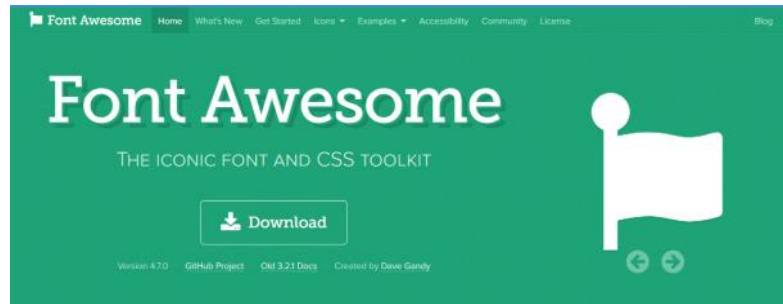
domingo, 28 de enero de 2018 22:07

## Instalación

npm install font-awesome

## Utilización

Ejemplo librería que proporciona elementos de interfaz basándose únicamente en la incorporación de CSS



Se añaden como CSS en `[styles]` o como `import` (igual que los CSS de Bootstrap)

```
@import "../node_modules/font-awesome/css/font-awesome.min.css";
```

Se insertan los iconos como elementos HTML vacíos, e.g. `<i></i>` a los que se aplica la clase adecuada

```
<i class="fa fa-star" aria-hidden="true"></i>
```

# Gráficos: ng2-chart

domingo, 28 de enero de 2018 21:59

<https://valor-software.com/ng2-charts/>

Ejemplo de librería que proporciona una serie de directivas, que aplicadas a la etiqueta *canvas* permiten generar diversos tipos de gráficos

## Instalación

npm install ng2-charts

## Configuración

La dependencia con "chart.js" obliga a incluir una referencia explícita a este. Para ello se utiliza [scripts] en angular-cli.json

```
"scripts": [  
  "../node_modules/chart.js/dist/Chart.bundle.min.js"  
],
```

## Utilización.

Como ejemplo, se crea un componente en el módulo inicio para que muestre un gráfico.



## i18n en Angular

Configuración de los parámetros de i18n para usar correctamente pipes como *Date* (o *Currency*)

```
import { LOCALE_ID, NgModule } from '@angular/core';
import { registerLocaleData } from '@angular/common';
import localeEs from '@angular/common/locales/es';

registerLocaleData(localeEs);

providers: [ { provide: LOCALE_ID, useValue: 'es' } ],
...

```

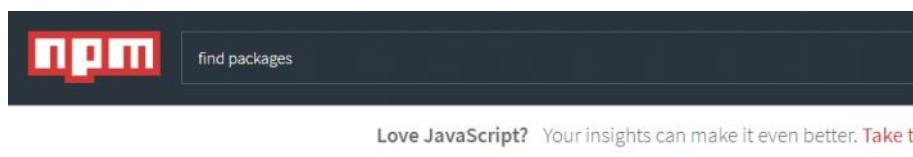
Ejemplo

Modificamos el *footer* para que utilice una variable de tipo *Date()* con el formato adecuado

Alejandro Cerezo - Madrid, 28 ene. 2018

## Librería de traducción

<https://www.npmjs.com/package/ng2-translate>



Librería para facilitar la traducción de la aplicación

### Instalación

```
npm install ng2-translate
```

### ★ ng2-translate public

An implementation of angular translate for Angular 2.

Simple example using ng2-translate: <http://plnkr.co/edit/btpW3l0jr5beJVjohy1Q?p=preview>

Get the complete changelog here: <https://github.com/ocombe/ng2-translate/releases>

- Installation
- Usage
- API
- FAQ
- Plugins
- Additional Framework Support

### Configuración

```
import { TranslateModule, TranslateLoader, TranslateStaticLoader } from 'ng2-translate';
import { HttpClientModule, HttpClient } from '@angular/http';

@NgModule({
  ...
  imports: [
    ...
    TranslateModule.forRoot({
      {
        provide: TranslateLoader,
        useFactory: tralationFactory,
        deps: [HttpClient]
      }
    )
  ],
  ...
})

```

Declara como *provider* un servicio incluido en la librería

Función que se declara en el módulo con los datos de la configuración

```

provide: TranslateLoader,
useFactory: tralationFactory,
deps: [Http]
}),
...

```

con los datos de la configuración

Tipo de acceso a los  
datos de traducción

```

export function tralationFactory(http: Http) {
  return new TranslateStaticLoader(http, '/assets/i18n', '.json');
}

```

Establece la localización de la carpeta en la que se podrá  
acceder via Http a los ficheros *json* con el diccionario de  
traducción específico de cada idioma

Fichero con los datos para un idioma, en esta caso en .json

```

{
  "Inicio": "Home" ,
  "Tareas": "ToDo",
  "Acerca de": "About"
}

```

El servicio `TranslateService` se inyecta en el componente principal y se utiliza  
para definir el idioma en que se renderizará la aplicación

```

import { TranslateService } from 'ng2-translate';

constructor(public translate: TranslateService) {
  this.translate.use('en');
}

```

Directiva responsable de indicar que elementos de la aplicación deben traducirse,  
en esta caso las opciones del menú

```

<a class="nav-link" [routerLinkActive]="['active']" routerLink="inicio"
  translate>Inicio</a>

```

# Selección de Idioma

domingo, 28 de enero de 2018 22:34

Añadimos la opción de seleccionar dinámicamente el idioma

```
import { TranslateService } from 'ng2-translate';

constructor(public translate: TranslateService) {
  this.aIdiomas = [
    {name: 'Español', code: 'es'},
    {name: 'Inglés', code: 'en'},
    {name: 'Francés', code: 'fr'}
  ]
  this.selectIdioma = {name: 'Español', code: 'es'};
  this.translate.use(this.selectIdioma.code);
}
```

Idiomas posibles

Idioma establecido a partir de uno de los posibles

Método manejador del evento de cambio en el select/options

```
seleccionarIdioma() {
  this.translate.use(this.selectIdioma.code);
}
```

Idioma establecido a partir de la selección del usuario

HTML del select/options

```
<div class="form-group">
  <label for=""></label>
  <select class="form-control" name="idioma" id="idioma"
    [(ngModel)] = "selectIdioma" (change)="seleccionarIdioma()">
    <option *ngFor="let idioma of aIdiomas" [ngValue]="idioma">{{idioma.name}}</option>
  </select>
</div>
```

# Refactorización

domingo, 28 de enero de 2018

23:03

Modelo de datos maestro: fichero maestros.models.ts

- Interface
- Clase

Datos correspondientes al modelo: fichero maestros.data.ts

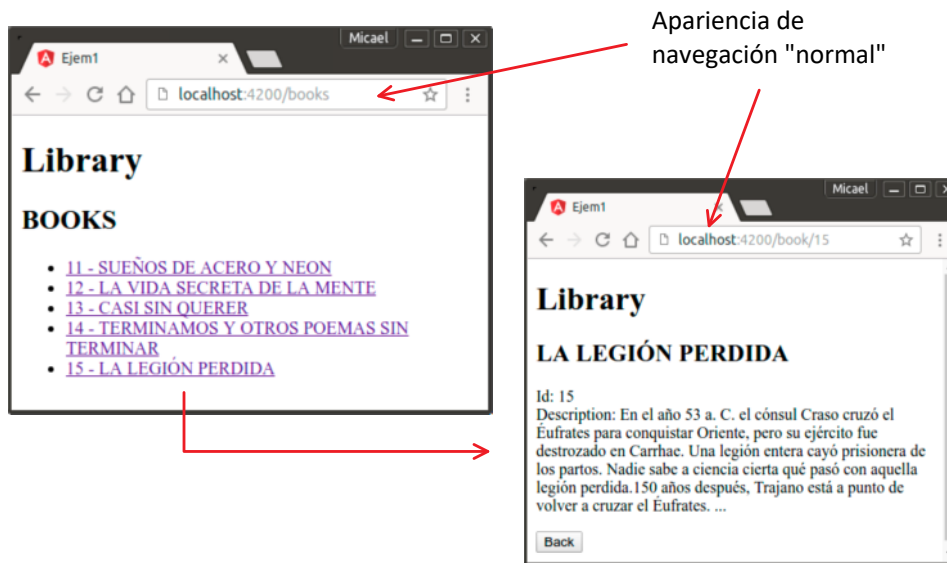
Componente idioma con el template del HTML

- Idioma seleccionado como @output
- Manejador del evento incluido en el componente

# Enrutamiento

jueves, 14 de septiembre de 2017 20:36

Las webs SPA (*single page application*) pueden tener varias pantallas simulando la navegación por diferentes páginas



<https://angular.io/docs/ts/latest/guide/router.html>

**Victor Savkin on Angular 2**  
In-depth articles about Angular 2 by a core team member.

**ANGULAR 2 ROUTER**

**Angular Router: Understanding Router State**  
An Angular 2 application is a tree of components. Some of these components are reusable UI components (e.g., list, table), and some are...

Victor Savkin  
Oct 30

<https://vsavkin.com/>

## Principios generales

- El componente principal de la aplicación (*app-root*) tiene una parte fija (cabecera, footer) y una parte cuyo contenido depende de la URL "salida" (*<router-outlet>*)
  - En *app.routing.ts* se define qué componente se muestra para cada URL, es decir las "rutas"

- Existen varias formas de recorrer la aplicación (navegar) :
  - Desde la URL indicada al navegador, escribiendo la ruta correcta
  - Desde los links específicos para navegar dentro de la aplicación web (*routerLink*)
  - Desde el código, de forma programática, gracias al método (*Router.navigate*)



## Procedimiento (1)

miércoles, 27 de septiembre de 2017 20:36

### Definición de las Rutas

*app.routing.ts*

Para cada URL se indica un nombre y el componente que será visualizado

valor por defecto si no se indica ninguna ruta

valor si se indica cualquier ruta distinta de las anteriores

```
import { AboutComponent } from '../about/about.component';
import { EnlacesComponent } from '../enlaces/enlaces.component';
import { AutoresComponent } from '../autores/autores.component';
import { CatalogoComponent } from '../catalogo/catalogo.component';
import { HomeComponent } from '../home/home.component';
import { RouterModule, Routes } from '@angular/router';
// cada ruta se identifica por su path y su componente
// en este ejemplo inicio, catalogo, autores, enlaces about
const routes: Routes = [
  { path: 'inicio', component: HomeComponent },
  { path: 'catalogo', component: CatalogoComponent },
  { path: 'autores', component: AutoresComponent },
  { path: 'enlaces', component: EnlacesComponent },
  { path: 'about', component: AboutComponent },
  { path: '', pathMatch: 'full', redirectTo: 'inicio' },
  { path: '**', redirectTo: 'inicio' }
];
export const appRouting = RouterModule.forRoot(routes);
```

"Configuración"

La definición de rutas puede hacerse en

- un fichero de rutas incorporado al módulo principal (después de crearlo puede completarse con el snippet Routes)
- un módulo de enrutamiento que define las rutas (es así como lo hace angular cli, cuando se utiliza `ng new --routing`)

### Configuración del módulo

*app.module.ts*

```
// Modulos de Angular
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { RouterModule } from '@angular/router';
import { NgModule } from '@angular/core';
// Modulos propios
import { SharedModule } from '../shared/shared.module';
import { appRouting } from '../app.routing';
;
// Componentes
...;

@NgModule({
  declarations: [
    ... componentes ...
  ],
  imports: [
    BrowserModule,
    FormsModule,
    appRouting,
    SharedModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Las rutas se consideran un módulo que debe importarse en la aplicación

## Procedimiento (2)

jueves, 14 de septiembre de 2017 21:27

### Componente principal

La vista (*template*) del componente principal define la posición de la "salida" del enrutado "router outlet"

*app.component.ts*

```
<header>
  <h1 class="title">Library</h1>
</header>

<router-outlet></router-outlet>

<footer>
  <p>...</p>
</footer>
```

Se pueden refactorizar como componentes

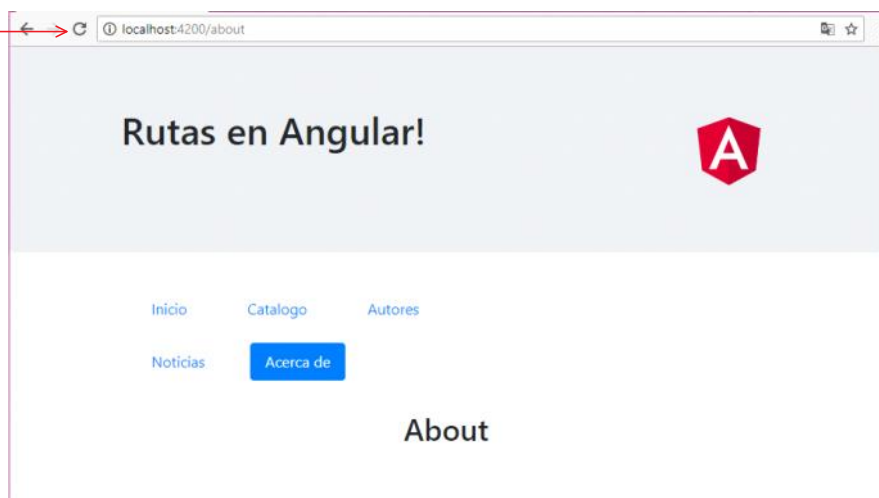
### Enlaces a las rutas

*app.component.ts*

```
<nav>
  <ul>
    <li><a routerLink="inicio" routerLinkActive="active">Inicio</a></li>
    <li><a routerLink="catalogo" routerLinkActive="active">Catálogo</a></li>
    <li><a routerLink="autores" routerLinkActive="active">Autores</a></li>
    <li><a routerLink="enlaces" routerLinkActive="active">Enlaces</a></li>
    <li><a routerLink="about" routerLinkActive="active">Acerca de</a></li>
  </ul>
</nav>
```

Aplica a la ruta activa la clase active para que resalte frente a las otras rutas

Además, indicando el nombre de la ruta en la barra del navegador también es posible acceder al destino



## Ejemplo

miércoles, 27 de septiembre de 2017

20:57



```
[routerLinkActive]="['active']"
```

Desde <<https://stackoverflow.com/questions/35422526/how-to-set-bootstrap-navbar-active-class-in-angular-2>>

# Parámetros

jueves, 14 de septiembre de 2017 21:45

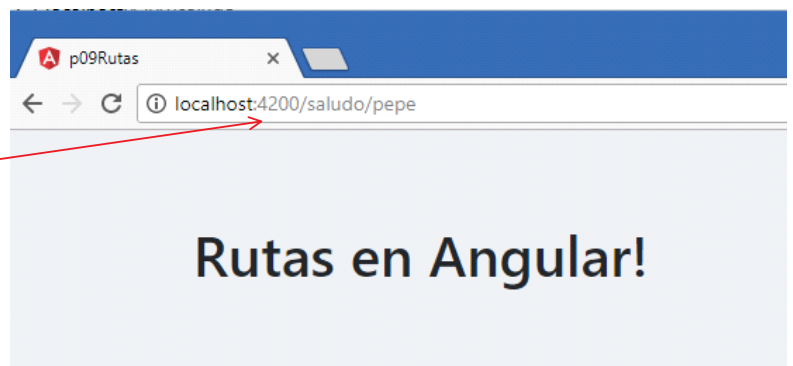
Una ruta puede incluir parámetros de forma estática  
En la constante Routes aparecerá como

```
{  
  path: 'saludo/:amigo',  
  component: SaludoComponent  
},  
}
```

El path incluye junto a si nombre la referencia a una parte variable

El componente correspondiente tendrá que ser capaz de recoger esa parte variable, como los parámetros que acompañan a la ruta

La URL para acceder a ella incluirá el valor asignado al parámetro de entrada



El componente indicado en la ruta accede al parámetro a través del servicio **ActivatedRoute**. donde existe una propiedad **params** de tipo **Observable**, que puede ser accedida de dos maneras

mediante una instantánea del estado del servicio, denominada **snapshot** que incluye el objeto (array asociativo) con cada uno de los parámetros

```
const user = this.activatedRoute.snapshot.params['amigo'];
```

declarando un observable que corresponde a la propiedad **params** y suscribiéndose a él para recoger los valores de cada parámetro concreto

```
let user;  
const user$: Observable<any> = this.activatedRoute.params;  
user$.subscribe ((parametros) => {  
  user = parametros['amigo'] || 'amigo';  
});
```

## Parámetros dinámicos en las URL

En vez de href, los links usan [routerLink].

La URL se puede indicar

- como un *string* (completa)
- como un array de *strings* si hay parámetros

```
<a [routerLink]="['/enlaces', enlace.id]">
```

En el siguiente ejemplo se generan en un \*ngFor enlaces específicos a cada uno de los libros incluidos en un array, donde cada ítem incluyen el id y el título de un libro

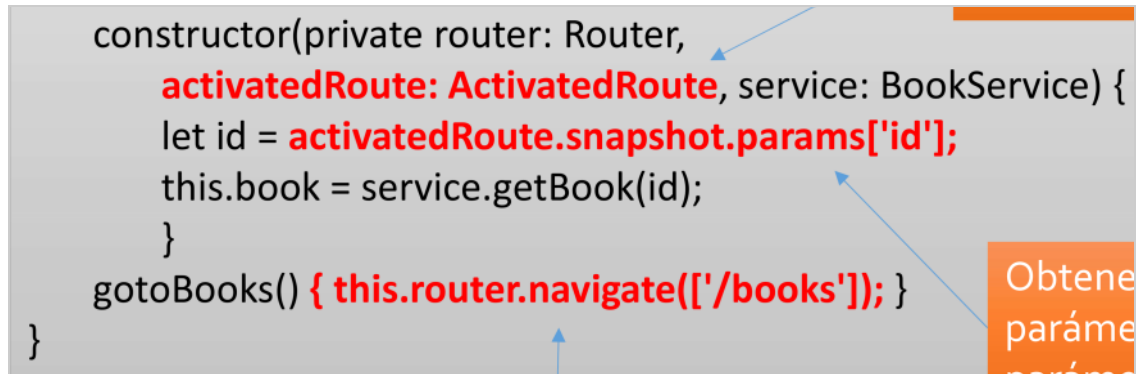
```
<a [routerLink]="['/book', book.id]">
  {{book.id}}-{{book.title}}
</a>
```

# Navegar desde el código

jueves, 7 de diciembre de 2017 10:45

Para navegar de forma programática o imperativa (desde código) usamos la dependencia Router y el método `navigate`.

```
constructor(private router: Router,  
             activatedRoute: ActivatedRoute, service: BookService) {  
    let id = activatedRoute.snapshot.params['id'];  
    this.book = service.getBook(id);  
}  
gotoBooks() { this.router.navigate(['/books']); }
```

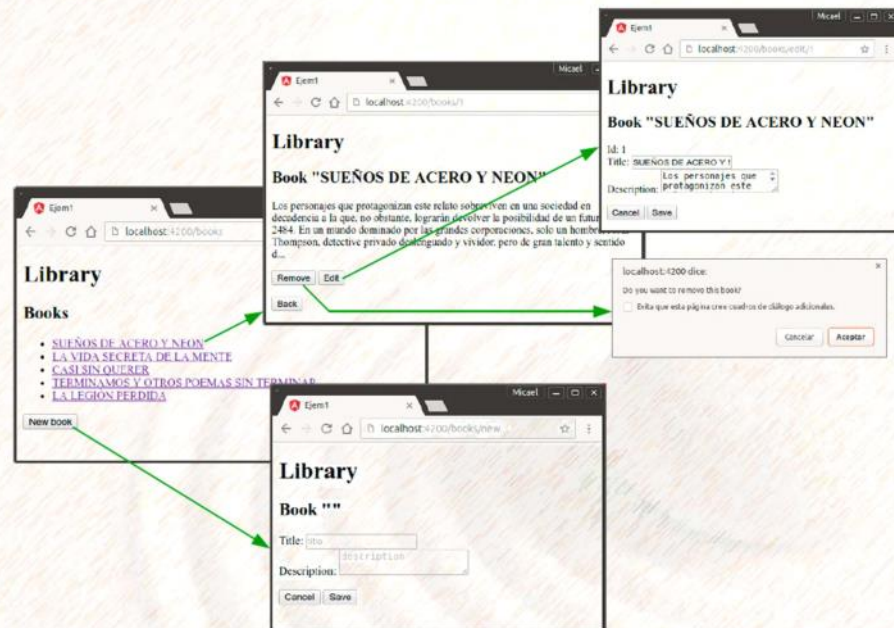


Obtene  
paráme  
paráme

## Ejemplo

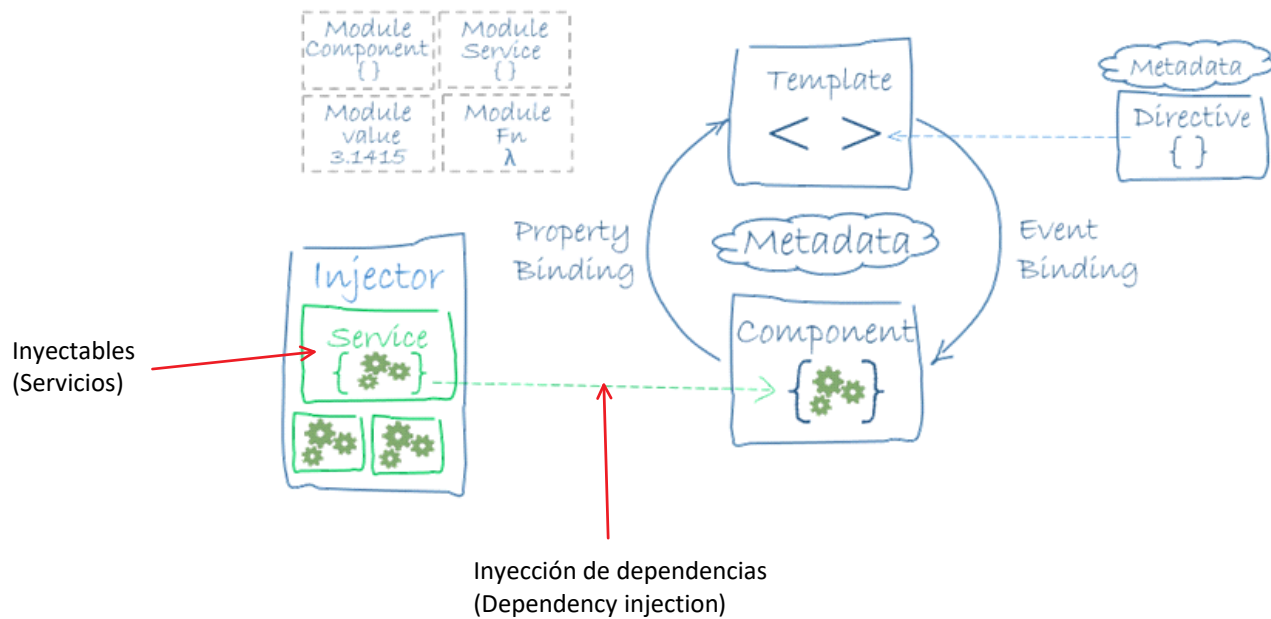
jueves, 14 de septiembre de 2017 21:43

# CRUD de libros



# Injectables (Servicios)

miércoles, 13 de septiembre de 2017 22:25



## Servicios

Elementos de la aplicación que no se encargan del interfaz de usuario

- Son clave para modularizar la aplicación en elementos que tengan una única responsabilidad
  - Componente: Interfaz de usuario
  - Servicios (e.g. Peticiones http)
- Permiten la buena práctica de NO acoplar en el componente la lógica de negocio, e.g. las peticiones http
- Permiten reducir la complejidad de los componentes y facilitar que estos sean ampliados / modificados
- Facilitan la implementar tests unitarios al reducir el número de responsabilidades que tiene el componente

## Servicios: características técnicas

- En principio se instancian según el **patrón singleton**: permiten compartir información entre componentes
- Los servicios mantienen el estado de la aplicación y los componentes ofrecen el interfaz de usuario
- Están anotados como **injectable** para que puedan ser inyectados en los componentes que necesitan utilizarlos
- Angular 2 ofrece muchos servicios predefinidos como la clase http utilizada para el acceso asincrónico (AJAX) a las APIs REST
- Lo habitual es que en cada proyecto de aplicación se implementen los servicios propios necesarios



# Inyección de dependencias

miércoles, 13 de septiembre de 2017 22:38

- La técnica que permite solicitar objetos al *framework* se denomina inyección de dependencias
- Las dependencias que un módulo necesita son inyectadas por el sistema
- Técnica muy popular en el desarrollo de *back-end* en *frameworks* como Spring o Java EE
- En Angular 2 se realiza mediante el constructor de la clase controladora del componente

<https://angular.io/docs/ts/latest/guide/dependency-injection.html>

La Inyección de Dependencias (DI) es un mecanismo para proporcionar nuevas instancias de una clase con todas aquellas dependencias que requiere plenamente formadas.

La mayoría de dependencias son servicios, y Angular usa la DI para proporcionar nuevos componentes con los servicios que necesitan.

En cada componente, se pueden indicar en el constructor todos aquellos servicios que necesita. A nivel interno, cuando Angular crea un componente, antes de crearlo obtiene de un Injector esos servicios de los que depende el componente.

# Inyector

viernes, 19 de enero de 2018 18:11

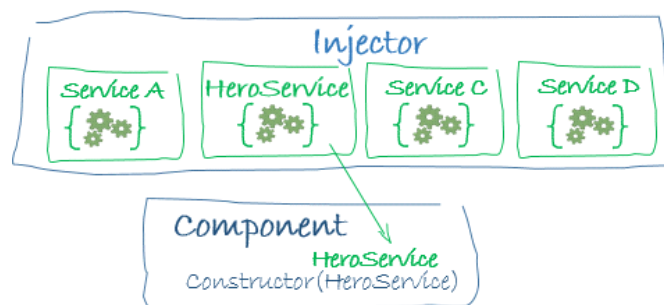
El Inyector es el principal mecanismo detrás de la DI.

A nivel interno, un inyector dispone de un **contenedor** con las **instancias de servicios** que crea él mismo.

- Si una instancia no está en el contenedor, el inyector **crea una nueva** y la añade al contenedor antes de devolver el servicio a Angular.
- Cuando todos los servicios de los que depende el contenedor se han resuelto, Angular puede llamar al constructor del componente, al que le pasa las instancias de esos servicios como argumento.

Dicho de otro modo, **la primera vez** que se inyecta un servicio, **el inyector lo instancia** y lo guarda en un contenedor. Cuando inyectamos un servicio, antes de nada el inyector busca en su contenedor para ver si ya existe una instancia.

Ese es el motivo por el que en Angular los servicios son **singletons**, pero solo dentro del **ámbito de su inyector**, sin olvidar que podemos tener inyectores a distintos niveles.



Cuando el inyector no tiene el servicio que se le pide, sabe cómo instanciar uno gracias a su **Provider**.

# Provider

viernes, 19 de enero de 2018 18:22

En Angular el ***provider*** es la propia clase que define el servicio.

Los *providers* pueden registrarse en cualquier nivel del árbol de componentes de la aplicación a través de los metadatos de componentes, a nivel de un módulo completo, en los metadatos de *NgModule*, o a nivel raíz, en el módulo principal de la aplicación, .

Como el inyector utiliza el *provider* cuando necesita instanciar un servicio, el nivel en el que se registra el *provider* determina a que nivel será *singleton* la instanciación

- Al registrar un *provider* en el *NgModule* del módulo principal , éste estará disponible para toda la aplicación instanciándose de forma *singleton* en toda ella.
- Si un servicio que se necesita declarar solo afecta a una pequeña parte de la aplicación, como puede ser un componente o un componente y sus hijos, tiene más sentido declararlo a nivel de componente.

# Registro en componentes

viernes, 19 de enero de 2018 18:33

- Se puede hacer que servicio no sea compartido entre todos los componentes de la aplicación (no *singleton*)
- Se puede crear un servicio exclusivo para un componente y sus hijos

En vez de declarar el servicio en el atributo providers del @NgModule se declara en el @Component

```
import { Component } from '@angular/core';
import { BooksService } from './books.service';
@Component({
  selector: 'app-algo',
  templateUrl: './app.component.html',
  providers: 'BooksService'
})
export class AlgoComponent {
  constructor(private booksService: BooksService){}
  ...
}
```


## Servicios para 1 componente

jueves, 14 de septiembre de 2017 18:38

- Se puede hacer que servicio no sea compartido entre todos los componentes de la aplicación (no *singleton*)
- Se puede crear un servicio exclusivo para un componente y sus hijos

En vez de declarar el servicio en el atributo providers del @NgModule se declara en el @Component

```
import { Component } from '@angular/core';
import { BooksService } from './books.service';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  providers: 'BooksService'
})
export class AppComponent {
  constructor(private booksService: BooksService){}
  ...
}
```



# Registro en módulos

viernes, 19 de enero de 2018 18:33

# Procedimiento

jueves, 14 de septiembre de 2017 18:41

Implementación de un servicio

- Se importa de *@angular/core* la clase *injectable*
- Se crea una nueva clase para el servicio
- Se anota nuestra clase con *@Injectable*
- Se indica esa clase en la lista de *providers* del *NgModule*
- Se pone como parámetro en el constructor del componente que usa el servicio

Se puede automatizar con angular-cli

```
ng g s <nombre>
```

Implementación del servicio en su fichero *<nombre>.service.ts*

Se importa de *@angular/core* la clase *injectable*

clase correspondiente al servicio anotada con *@Injectable*

Funcionalidad del servicio, normalmente mediante métodos que retornan determinados valores

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

@Injectable()
export class AlgunService {

  unMetodo() {
    return ...;
  }
}
```

Incorporación del servicio en un módulo, en este caso el módulo principal, *app.module*

Importación del servicio, a partir del fichero que lo contiene

Declaración de que se trata de un servicio, incluyéndolo en la lista de *providers*

```
import ...
import { AlgunService } from './algun.service';

@NgModule({
  declarations: [AppComponent],
  imports: [...],
  bootstrap: [AppComponent],
  providers: [AlgunService]
})
export class AppModule { }
```

Consumo del servicio en un componente, en este caso el componente principal *app-root*

Inyección del servicio en el constructor

Uso de los métodos del objeto correspondiente al servicio, instanciado de forma *singleton* en la aplicación

```
import { AlgunService } from './algun.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {

  constructor(private algunService: AlgunService) { }

  // En algún sitio
  ... this.algunService.unMetodo() ...
}
```

## Ejemplo

*books.service.ts*

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

@Injectable()
export class BooksService {
  getBooks(title: string) {
    return [
      'Aprende Angular 2 en 2 días',
      'Angular 2 para torpes',
      'Angular 2 para expertos'
    ];
  }
}
```

*app.module.ts*

```
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { NgModule } from '@angular/core';
import { HttpClientModule, JsonpModule } from '@angular/http';
import { AppComponent } from './app.component';
import { BooksService } from './books.service';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, FormsModule, HttpClientModule,
    JsonpModule],
  bootstrap: [AppComponent],
  providers: [BooksService]
})
export class AppModule { }
```

*app.component.ts*

(componente que  
utiliza o consume el  
servicio)

```
import { Component } from '@angular/core';
import { BooksService } from './books.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  private books: string[] = [];
  constructor(private booksService: BooksService) { }
  search(title: string) {
    this.books = this.booksService.getBooks(title);
  }
}
```

*app.component.html*

```
<h1>Memory Books</h1>

<input #title type="text">
<button (click)="search(title.value); title.value=''">
  Buscar
</button>

<p *ngFor="let book of books">{{book}}</p>
```



Con frecuencia los servicios encapsulan el acceso al backend con API REST (**buena práctica**),

No pueden devolver información de forma inmediata, teniendo que esperar para devolver información cuando llega la respuesta del servidor

En JavaScript los métodos no se pueden bloquear esperando la respuesta. Son

asíncronos reactivos

```
private service: BooksService = ...  
let books =  
this.booksService.getBooks(title);  
console.log(books);
```

NO se puede hacer

- Callbacks
- Promesas
- Observables

## Callbacks

Al consumir el servicio, se le pasa como parámetro una función (de *callback*), frecuentemente en forma de función anónima.

Esta función

- será ejecutada cuando llegue el resultado.
- recibe como primer parámetro el error (si ha habido)

```
service.getBooks(title, (error, books) =>  
  {  
    if(error) {  
      return console.error(error);  
    }  
    console.log(books);  
  }  
);
```

## Promesas

El método devuelve un objeto *Promise*.

- Con el método *then* de ese objeto se define la función que se ejecutará cuando llegue el resultado.
- Con el método *catch* de ese objeto se define la función que se ejecutará si hay algún error

Creación de la promesa

```
getBooks(): Promise<Book[]> {  
  return Promise.resolve(aBooks);  
}
```

Consumo del servicio

```
service.getBooks(title)  
  .then(books => console.log(books))  
  .catch(error => console.error(error));
```

recomendada si la funcionalidad es suficiente

## Observables

Similares a las promesas pero con más funcionalidad. (Más complejos de programar)  
Con el método subscribe se definen las funciones que serán ejecutadas cuando llegue el resultado o si se produce un error

```
service.getBooks(title).subscribe(  
  books => console.log(books),  
  error => console.error(error)  
);
```

Es la forma recomendada por Angular 2 por ser la más completa (aunque más compleja)

## Ejemplo: "Maqueta" con promesas

sábado, 9 de diciembre de 2017 16:22

```
aLibros: Array<string>;
constructor() {
  this.aLibros = [
    'Angular básico',
    'Angular en 19 minutos',
    'Angular avanzado'
  ];
}

buscarLibrosAsync(clave: string) {
  return new Promise(
    (resolve, reject) => {
      setTimeout(
        () => { resolve(this.aLibros); }, 2000
      );
    }
  );
}
```

Array de datos para simular el resultado de una búsqueda

Se instancia y se devuelve un objeto "promesa"

La promesa recibe dos funciones *callback* que serán responsables de que sea resuelta o rechazada

En este caso, después de un tiempo definido por `setTimeout` para generar asincronía, la promesa se resuelve siempre correctamente, devolviendo el array de datos que simula el resultado de la búsqueda

### Consumo del servicio

```
btnBuscar() {
  this.aLibros = [];
  this.librosMockService.buscarLibrosAsync(this.sClave)
    .then(
      (response) => { this.aLibros = response; }, // función OK
      (error) => { console.log(error); } // función error
    );
}
```

El método `then` permite definir las dos funciones que se ejecutarán tanto si la promesa se resuelve como si es rechazada

## Cliente REST (AJAX)

Angular utiliza como cliente de API REST un objeto correspondiente a un servicio (inyectable) que lleva a cabo las peticiones asíncronas al servidor

- vía el objeto XMLHttpRequest (nativo de JavaScript)
- vía JSONP.

Angular 2/4 utiliza objetos de la clase **Http**, que incluye diversos métodos alternativos o atajos (shortcuts)

```
http()  
http.get()  
http.post()  
http.put()  
http.delete()  
http.jsonp()  
http.head()  
http.patch()
```

Angular 5 incorpora un nuevo servicio, **HttpClient**, de uso más sencillo

En Angular 1.x, devolvía una promesa (similar a JQ)

En Angular devuelve un observable, u opcionalmente, una promesa

<https://angular.io/docs/ts/latest/guide/server-communication.html>

<https://angular.io/docs/ts/latest/api/http/index/Http-class.html>

## Servicio Http

### Inyección del servicio en el componente

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

@Component({...})
export class NameComponent implements OnInit {
  constructor(public http: Http) { }
```

Con frecuencia el proceso es algo más complejo, al estar mediado por un servicio propio del usuario que a su vez hace uso del servicio Http.

## Servicio HttpClient

Aparece en Angular 4.3, en el módulo HttpClientModule incluido en @angular/common/http, como una completa reimplementación de HttpModule, que en la versión 5 pasa a estar deprecado.

En el nuevo servicio se accede directamente a la respuesta en formato JSON, sin necesidad de manipulaciones previas para obtener dicho formato

### Inyección del servicio en el componente

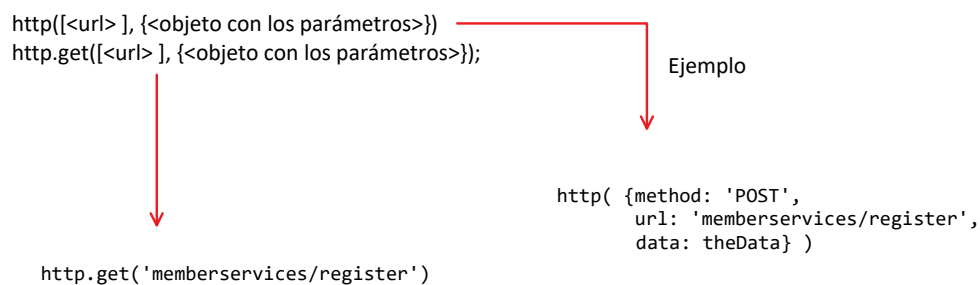
```
import { HttpClient } from '@angular/common/http';

@Component({...})
export class NameComponent implements OnInit {
  constructor(public http: HttpClient) { }
```

<https://medium.com/codingthesmartway-com-blog/angular-4-3-httpclient-accessing-rest-web-services-with-angular-2305b8fd654b>

## Consumo (uso) del servicio

Se utiliza el propio servicio o alguno de los métodos que proporciona (*get*, *post*...)



# Procesamiento de promesas

sábado, 9 de diciembre de 2017 20:09

El modificador `asPromise()` permite transformar la respuesta en una promesa, como las utilizadas por el servicio `$http` en *AngularJS*

```
http.get(url).toPromise()
```

## Respuesta al servicio Http

```
this.http.get(url)
  .toPromise()
  .then(
    response => console.log(response.json()), // promesa resuelta
    error => console.error(error); // promesa rechazada
  );
```

ejecutaremos alguna de las funciones definidas según el resultado de la promesa

Si se ha resuelto correctamente la promesa, para obtener los datos enviados por el servidor usamos el método `json()` del objeto `response`

## Respuesta al servicio HttpClient

```
this.http.get(url)
  .toPromise()
  .then(
    response => console.log(response), // promesa resuelta
    error => console.error(error); // promesa rechazada
  );
```

En el nuevo servicio se accede directamente a la respuesta en formato JSON, sin necesidad de manipulaciones previas para obtener dicho formato

## Ejemplo

jueves, 14 de septiembre de 2017 22:43

### API de Google para la búsqueda de libros

[https://www.googleapis.com/books/v1/volumes?q=intitle:"clave"](https://www.googleapis.com/books/v1/volumes?q=intitle:)

```
search(title: string) {  
  this.books = [];  
  let url = "https://www.googleapis.com/books/v1/volumes?q=intitle:"+title;  
  
  this.http.get(url)  
    .map(response => response.json())  
    .subscribe(  
      response => {  
        const data = response.items;  
        data.forEach ( (item) => {  
          const bookTitle = item.volumeInfo.title;  
          this.books.push(bookTitle);  
        }  
      )  
    },  
    error => console.error(error)  
  );  
}
```

Se procesa la información proporcionada por el API consultada, en este caso para crear un array únicamente con los títulos de los libros

Las operaciones concretas de esta etapa de procesamiento de la respuesta dependen siempre de la estructura concreta de los datos proporcionados por la API que es consultada

### Busqueda de libros en Google

Clave de búsqueda

peces

BuscarBuscar RxBuscar RxFull

### Libros encontrados

The precautionary approach to fisheries with reference to straddling fish stocks and highly migratory fish stocks
100 Peces Argentinos
Peces en bandeja
El día que cambié a mi padre por dos peces de colores
El Sueño de Los Peces
David, peces, pingüinos...
Manuales del acuario. Peces rojos o carpas doradas
Los Panes y los Peces
Peces del llano
El pan y los peces

### API con datos geográficos de países de un continente

<http://restcountries.eu/rest/v1/region/> <continente>

africa  
europe  
americas...

# Países: consumo de un servicio

Países del Mundo

Selecciona un continente:

Africa  
Europa  
America

Alejandro L. Cerezo  
CLE Formación  
Madrid - 2015



## Servicios propios

jueves, 14 de septiembre de 2017 22:52

Al hacer una petición REST con `Http` / `HttpClient` obtenemos un objeto `Response` que debe ser procesado de acuerdo con las características del API concreto del que proceden los datos

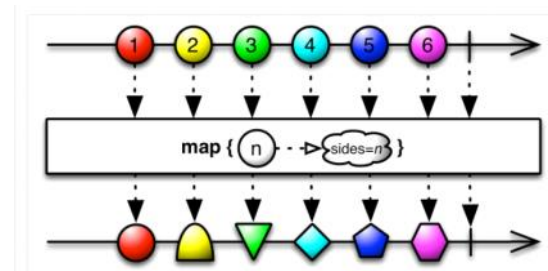
En lugar de utilizar directamente dichos servicios conviene encapsularlos en otros, capaces de ofrecer objetos de alto nivel a los clientes del servicio (e.g. el array de títulos ya procesado en el ejemplo que hemos visto)

Nuestro propio servicio realizara varias operaciones

- La consulta al API via `Http` / `HttpClient`
- La transformación del objeto `Response` en el conjunto de datos adecuado (e.g. el array de títulos) cuando llegue la respuesta
- El envío de los datos ya transformados con el mismo formato de llegada, para conservar la asincronía del proceso: un `Observable` o una `Promesa`, como los que proporcionan los servicios nativos

```
buscarRx (clave: string) {  
  const url = this.sURL + clave;  
  return this.http.get(url)  
    .map(response => this.extractTitles(response));  
}  
  
private extractTitles(response: any) {  
  if (response.items) {  
    return response.items.map(book => book.volumeInfo.title);  
  } else {  
    return response;  
  }  
}
```

el operador map



## Servicios *stateless* (sin estado)

- No guardan información
- Sus métodos devuelven valores, pero no cambian el estado del servicio
- Ejemplo: BooksService con llamadas a Google

## Servicios *statefull* (con estado)

- Mantienen estado, guardan información
- Al ejecutar sus métodos cambian su estado interno, y también pueden devolver valores
- Ejemplo: LoginService con información en memoria

## ¿*Stateless* vs *statefull*?

- Los servicios *stateless* son más fáciles de implementar porque básicamente encapsulan las peticiones REST al *backend*
- Pero la aplicación es menos eficiente porque cada vez que se visualiza un componente se tiene que pedir de nuevo la información
- Los servicios *statefull* son más complejos de implementar porque hay que definir una política de sincronización entre *frontend* y *backend*
- Pero la aplicación podría ser más eficiente porque no se consulta al *backend* constantemente

# Programación reactiva

sábado, 9 de diciembre de 2017 16:39

# Conceptos

viernes, 19 de enero de 2018 11:48

- Flujo de datos
- Observables

# Flujo de datos

miércoles, 31 de enero de 2018 0:49

la **programación reactiva** se basa en programar con **flujos de datos** (*streams*) **asíncronos**.

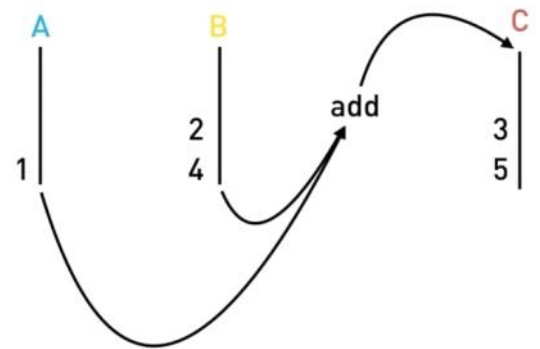
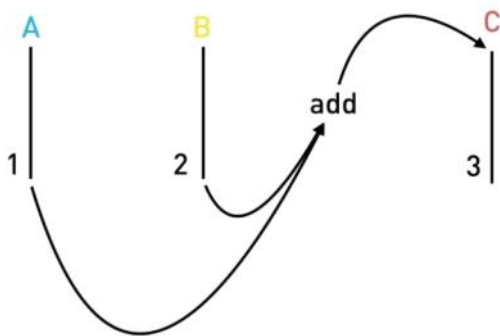
proporciona una alternativa a otras formas de gestionar la asincronía, como *callbacks* o promesas

flujos de datos (*streams*):  
series de datos encadenados que  
pueden ser emitidos en el tiempo.

- el registro del movimiento del ratón
- los datos enviados y recibidos de una base de datos
- los *arrays* en general son también flujos de datos

secuencias de valores a lo largo del tiempo

los **operadores** son las funciones que permiten realizar operaciones sobre estos streams



# Observables

miércoles, 31 de enero de 2018 0:50

## Observables

- son mecanismos creados para representar esos flujos de datos

son un nuevo tipo primitivo, que actúan como un plano (*blueprint*) o representación de cómo podemos crear *streams*, suscribirnos a ellos, responder a nuevos valores o combinar varios *streams* para construir uno nuevo.

Los Observables se basan en dos patrones de programación



- es el patrón "Observer"
- el patrón "Iterator"

De esta manera no debemos pensar en arrays, eventos de ratón, llamadas http al servidor... separados, sino en algo que los agrupa a todos, el Observable. De alguna manera, cuando quieras hacer programación reactiva con un array, habrá un método para poder transformar el array en Observable y poder trabajar con él de esta manera.

Aplicación del patrón *observer*:



Tratar todo tipo de información como un *stream* observable de entrada y de salida, al cual se le pueden agregar operaciones que procesan los flujos de datos.

- un proceso como la comunicación mediante http a un servidor es in flujo de datos que puede ser representado por un **Observable**. De esta forma toda comunicación será "vigilada" por éste.
- Se puede definir un **Observador**, es decir un elemento capaz de "mirar" a un Observable y reaccionar a los cambios que se produzcan en aquel.
- Para ello un Observador se **Suscribe** a un determinado Observable como el mencionado, para que reaccione en concreto a aquellos cambios en la comunicación con el servidor.

Actualmente se está valorando la posibilidad de incorporar este nuevo tipo en el estándar ES7.

Entre tanto, la librería **RxJS** (*Reactive Extensions for JavaScript*) proporciona su implementación en ES6.

<https://codekstudio.com/post-blog/conceptos-observables-rxjs-y-angular-2->



Observables en ES6:  
Implementados  
en la [librería RxJS](#)  
(*Reactive Extensions*  
for JavaScript)



Extensiones reactivas  
para JavaScript incluidas  
en Angular



[ReactiveX](#)

Utilizada principalmente en

- Formularios reactivos
- Emisión de eventos
- Servicio Http

<http://reactivex.io/>





## Uso de RxJS

viernes, 19 de enero de 2018 12:12

```
import * as Rx from 'rxjs/Rx';
```

De esta forma, *Rx* es el objeto que representa la librería RxJS ya importada en la aplicación.

Gracias a ello es posible crear Observables o transformar datos existentes, como *arrays* a Observables,

Teniendo un Observable, *RxJS* proporciona numerosas herramientas (operadores) para poder manejar ese flujo de datos, entre los que destaca el operador *map*

```
Rx.Observable
.from(array)
.map(function(element) {
  return element + 2;
})
.filter(function(e) {
  return e > 10;
});
```

Aumentamos en 2 cada elemento del flujo de datos y filtramos solo aquellos valores que son mayores que 10

Los datos son inmutables, por lo que al modificarlos se crean copias de los mismos

Al crear un observador (*Observer*), se pueden definir las respuestas a diferentes eventos del observable, como son que cambie (*onNext*), que emita un error (*onError*) o que se complete el flujo y termine su emisión (*onCompleted*).

```
const observador = Rx.Observer.create(
  function onNext(x) { console.log('Next: ' + x); },
  function onError(err) { console.log('Error: ' + err); },
  function onCompleted() { console.log('Completed'); }
);
```

*create()* es el método por defecto, por lo que puede obviarse, indicando directamente *Rx:Observer(...)*.

Por último suscribimos a nuestro Observador a nuestro Observable y de esta forma el Observable comunica al Observador sus cambios.

```
observable.subscribe(observador);
```

El array no es emitido ni manejado de ninguna manera por el Observable hasta que un Observador como mínimo se suscriba a él. Esto es importante porque de esta manera no se consumen recursos sin sentido.

Ahora, cualquier cambio, como es que se añada al array un nuevo miembro le será notificado al observador que responderá con la función "*onNext*", en la que podrá definirse la reacción adecuada en cada aplicación.

# Observables

sábado, 9 de diciembre de 2017 17:16

Importamos Observable desde rxjs/Observable.

```
import { Observable } from 'rxjs/Observable';

buscarLibrosAsyncRx(clave: string) {

  return new Observable( ←

    (observer) => {
      setTimeout(() => {
        observer.next(this.aLibros);
      }, 2000);
    }
  );
}
```

Instanciamos un nuevo Observable con el método Observable, que encapsula Rx.Observable.create, definiendo la función que será ejecutada cuando se produzca alguna suscripción, representada por el parámetro observer

create(obs => { obs.next(1); })

1

El equivalente en ES6 puro, usando NodeJS, sería

```
let Rx = require('rxjs');
let observable = Rx.Observable.create(
  (observer) => {
    observer
      .interval(2000)
      .next(this.aLibros);
  }
);
```

El método next() es el responsable de emitir un evento, con los datos indicados, que será recogido por el observador, que habrá sido definido con el método subscribe(). Eventos de otro tipo son emitidos mediante error() y complete()

## Observador y suscripción

método del servicio que devuelve un observable

```
btnBuscarRx() {
  this.aLibros = [];
  this.librosMockService.buscarLibrosAsyncRx(this.sClave)
    .map(response => response)
    .subscribe(
      (response) => {
        console.dir(response);
        this.aLibros = response;
      }
    );
}
```

ejemplo de los operadores que permiten manipular los observables

suscripción al observable

Una vez suscritos ejecutaremos alguna de las tres funciones definidas en función de los estados del observable:

- onNext
- onError
- onComplete

## Angular utiliza Observables en diversas situaciones

Como base de la emisión y vigilancia de **eventos**, uno de los patrones básicos en la comunicación **entre componentes**. Cuando se necesita que el resto de la aplicación conozca un cambio en un componente y reacciones al mismo, se hace uso de **EventEmitter**, que no es más que una clase de Angular que envuelve métodos de *RxJS*.

En los formularios basados en el modelo (*ModelDriven*)

En la comunicación con el servidor

```
http.get('/api/usuario')  
  .map(res: Response => res.txt)  
  .subscribe(res => this.user = res);
```

la respuesta a una solicitud http al servidor es un Observable.

Puede ser procesada mediante operadores

En lugar de crear explícitamente un Observador, se encadena directamente “subscribe” a la cadena de operadores pasándole una función.

Este modo rápido supone realmente la **creación de un Observador** en el que la primera (y única) función indicada se le asigna al evento “*onNext*”, que es el primero de los que pueden definirse.

# Programación funcional

viernes, 19 de enero de 2018 16:52

La programación funcional pretende básicamente actuar de forma **declarativa** en lugar de imperativo, es decir indicar que es lo que quiere hacer en cada momento y no como hacerlo

cuando hemos aplicado la función “map” al array estamos diciendo “quiero crear un nuevo array pero sumando 2 a cada número”. En un modo tradicional, en Javascript tendrías que hacer un bucle para recorrer el array y luego crear un nuevo array para introducir los nuevos valores, estarías diciendo “como” hacerlo, y no “que” quieres solamente.

La programación funcional en realidad trata todo como **funciones matemáticas**. Quiero hacer esto y lo otro mediante esta y la otra función, las combino, las resto... pero además no cambia el estado ni los datos del programa, cada función opera de manera aislada y no utiliza sentencias sino declaraciones. Por ejemplo, los bucles no son buenos amigos de la programación funcional. La **inmutabilidad** de la que antes hemos hablado un poco, es un concepto principal en PF. Nunca se modifican los datos. ¿Como se consigue?, básicamente realizando copias de los mismos cuando se deben alterar.

Además los datos de salida de una función solo dependen de los argumentos que son introducidos en la función, y solo de estos, asegurando que una función siempre produce los mismos resultados **eliminado efectos colaterales**, que son efectos producidos en el exterior de una función pero producidos por ella (cambios de estado). Por ejemplo cuando una función cambia un valor de una variable global exterior. Esto resulta en algo impredecible puesto que cualquiera puede “tocar” esa variable exterior, cambiando el estado.

## Aproximaciones en el desarrollo Web

Pues bien, todo esto es para decir que realmente ni Angular 2, ni RxJS ni si quiera React ni otras muchas librerías y Frameworks que hay por ahí son programación funcional. Lo son **Hope, Haskell, Erlang o F#**, que son mayoritariamente usados en **ámbitos académicos**. Todo este rollo entonces ¿para qué?, pues porque este paradigma de programación es algo a lo que muchos nuevos lenguajes y frameworks intenta emular puesto que las ventajas son bastante importantes.

Cualquier frameworks (casi) usado hoy en día para el desarrollo web está de una manera u otra basado en programación de objetos, por lo que no tiene cabida la programación funcional. No obstante veamos el caso de Angular. Ha pasado de tener el “two way data binding” (vínculos de dos vías entre datos y vista) como piedra angular (o una de ellas), para pasar en Angular 2 a proclamar la vía única (“one way”) como santo grial, algo que React lleva en la sangre desde siempre. Es decir ahora los datos fluyen en un solo sentido y no se pueden modificar (o deben) desde diferentes lados porque no se podría razonar bien sobre ellos, perderíamos el control más fácilmente.

No obstante, dicho esto, sí que debemos de tomarnos en serio este giro hacia la programación funcional y reactiva, puesto que todo apunta a un futuro prometedor. Mi recomendación es que si ves artículos que hablen de estos temas, te los leas para ver que se cuece y entender mejor estos apasionantes conceptos que te hemos presentado.

## Formas de comunicación

Comunicación entre un componente padre (contenedor) y un componente hijo (incluido en el anterior)

- Configuración de propiedades (Padre → Hijo)
- Envío de eventos (Hijo → Padre)

- Invocación de métodos (Padre → Hijo)
  - Con variable *template*
  - Inyectando hijo con *@ViewChild*
- Compartiendo el mismo servicio (Padre ↔ Hijo)

Los inyectables (servicios) son objetos *singleton* y por tanto compartidos entre las distintas clases que los instancian

<https://angular.io/docs/ts/latest/cookbook/component-communication.html>

# Inyección de servicios observables

miércoles, 31 de enero de 2018 0:08

Comunicación entre componentes completamente desacoplada  
"en medio hay algo", donde un componente puede escribir y otro puede enterarse de los cambios al estar suscrito a un *stream* de datos.

El problema es como un componente se entera de que un dato ha cambiado en otro componente.

Una alternativa al patrón *pull*, en el que se consulta cada cierto tiempo una variable para poder detectar los cambios.

- Observable → permite la suscripción a cambios de un stream
- Subject → se comporta como un observable y además permite la emisión de datos como un observable, que puede ser leído por los subscriptores pero no modificado

Se utiliza la librería **rxjs** de Microsoft

En Angular, la detección del cambio pasa a ser una tarea del programador  
En este modelo reactivo, los cambios corresponden a la emisión de un evento

## Ejemplo

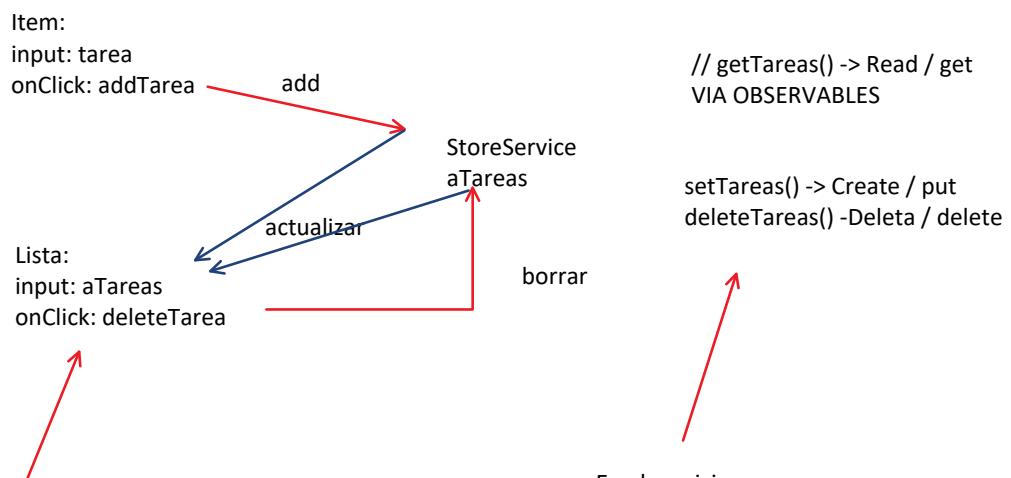
miércoles, 31 de enero de 2018 0:15

En una interpretación muy imprecisa del patrón REDUX

Componte 1 - entrada de datos

Componente 2 - presentación de los datos

Ambos usan el servicio que define los datos



En el componente lista

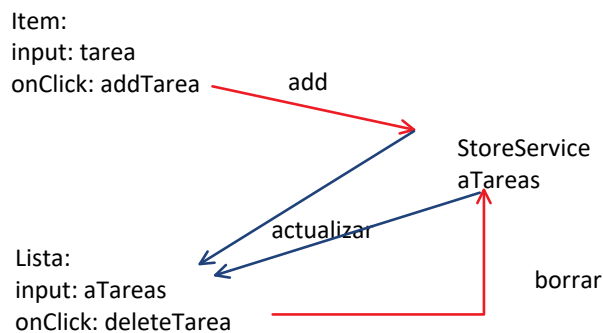
- Se crea un observable correspondiente al array datos
- El observable refleja el subject del servicio
- Así es posible suscribirse al servicio, que actúa como emisor de eventos
  - Las respuestas del servicio son similares a las de HttpClient cuando accede a un API REST
  - El método `suscribe()` permite procesar la respuesta, pasando los valores recibidos al array de datos

En el servicio

- Se crea un Subject correspondiente al store. Una variable capaz de informar de los cambios que se produzcan en la anterior
- Un método del servicio devuelve el Subject, de modo que desde fuera sea posible suscribirse a él
- Cada vez que el store cambia (se añaden o se eliminan items), se emite un evento mediante el método `next()` del subject
- De esa forma los elementos suscritos al subject son informados de los cambios

# Procedimiento

martes, 30 de enero de 2018 23:54



- Se declara un observable correspondiente al array datos:

```
aItems: Array<any>;
aItems$: Observable<any[]>;
```

- En el observable se recoge el Subject del servicio

```
this.aItems$ = this.datosService.getSubjectDatos();
```

En la suscripción Subject/Observable se establece la respuesta a los cambios

```
this.aItems$.subscribe(
  (data) => this.aItems = data
);
```

Como alternativa al procesamiento se puede enviar directamente el observable al template HTML usando el pipe async para su correcta visualización

geSubjectTareas() -> Read / get  
VIA OBSERVABLES

setTareas() -> Create / put  
deleteTareas() -Delete / delete

- Se crea un Subject correspondiente al store

```
aDatos: Array<any>;
aDatos$: Subject<any[]>;
```

variable capaz de informar de los cambios que se produzcan en la anterior

- Se instancia en el constructor

```
this.aDatos$ = new Subject<any[]>();
```

Una función permite suscribirse al store

```
suscribeDatos() {
  return this.aDatos$;
}
```

Es posible devolver el Subject como un observable, para encapsular la forma en que se ha implementado el proceso

```
return this.aDatos$.asbservable
```



## Clase 5b - Observables (1:32:40)

viernes, 29 de septiembre de 2017 20:21

### En el servicio

- se declara una variable que almacenara información: un "almacén".  
Si existe un interfaz con el modelo de los datos, se declara como array de este tipo

```
// Almacén de movimientos en memoria  
private movimientos: MovimientoModel[]
```


- se declara el emisor de eventos correspondiente; por convenio su nombre es el de la variable seguido de \$.  
Su tipo se crea mediante un **genérico** del objeto *Subject* específico del tipo del "almacén"  
Se instancia el objeto correspondiente

```
// Emisor de eventos relacionados con el almacén de movimientos  
private movimientos$: Subject<MovimientoModel []> = new Subject<MovimientoModel []>();
```

- cualquier modificación en el almacén va acompañada por la **emisión de un evento** que genera un nuevo valor en el observable

```
this.movimientos. push (movimientos) ;  
// se genera un nuevo valor en el observable  
this.movimientos$.next ( this.movimientos) ;
```

el objeto *Subject* emite un evento que concuerda con su tipo especificado



- definimos un método que devuelve el *Subject*

```
Devuelve un observable que notifica cambios en el  
getMovimientos$(): Observable<MovimientoModel[]> {  
  // se comporta como un observable  
  return this.movimientos$.asObservable() ;  
}
```

### En el componente 2

- declaro la propiedad correspondiente al objeto *Observable* sólo para el tipo específico con el que estamos trabajando

```
movimientos$: Observable<MovimientoModel []>
```

- inyector en el constructor el servicio

```
Este componente depende del objeto DatosService  
constructor(private datosService: DatosService) { }
```

enlazo el componente con el "emisor de eventos" en el servicio

```
ngOnInit() {  
  // Al iniciar el componente se enlaza con el almacén de datos  
  this.movimientos$ = this.datosService.getMovimientos$();  
  // si se quiere se puede suscribir programáticamente  
  this.movimientos$.subscribe(  
    d=>console.log("Dato recibido:")  
  // como alternativa a esta última línea  
  // se puede usar el pipe async en la vista  
  )  
}
```

En la vista del componente

El pipe async implica una suscripción a un observable

```
<!--el pipe async se suscribe a un observable-->  
<!--los pipes se pueden entubar-->  
{{ movimientos$ | async | json }}
```

## Servicio Http

### Inyección del servicio en el componente

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

@Component({...})
export class NameComponent implements OnInit {
  constructor(public http: Http) { }
```

Con frecuencia el proceso es algo más complejo, al estar mediado por un servicio propio del usuario que a su vez hace uso del servicio Http.

## Servicio HttpClient

Aparece en Angular 4.3, en el módulo HttpClientModule incluido en @angular/common/http, como una completa reimplementación de HttpModule, que en la versión 5 pasa a estar deprecado.

En el nuevo servicio se accede directamente a la respuesta en formato JSON, sin necesidad de manipulaciones previas para obtener dicho formato

### Inyección del servicio en el componente

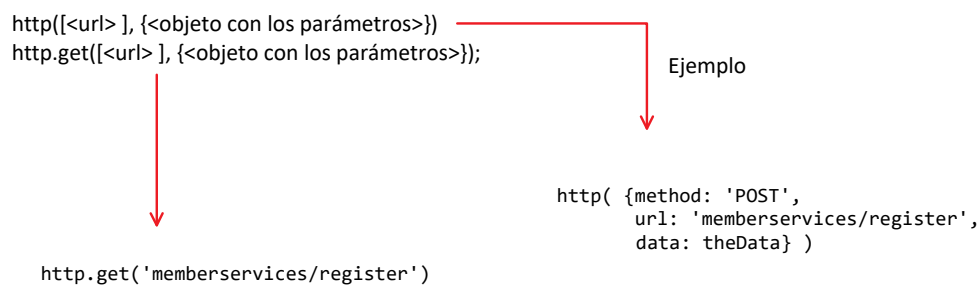
```
import { HttpClient } from '@angular/common/http';

@Component({...})
export class NameComponent implements OnInit {
  constructor(public http: HttpClient) { }
```

<https://medium.com/codingthesmartway-com-blog/angular-4-3-httpclient-accessing-rest-web-services-with-angular-2305b8fd654b>

## Consumo (uso) del servicio

Se utiliza el propio servicio o alguno de los métodos que proporciona (*get*, *post*...)



## Procesamiento de observables

jueves, 7 de diciembre de 2017 20:07

La respuesta por defecto a una petición http en cualquiera de los servicios Angular (Http o HttpClient) es un **observable**.  
Por tanto necesitamos **suscribirnos** a él para gestionar la respuesta.

### Respuesta al servicio Http

```
this.http.get(url)
.subscribe(
  response => console.log(response.json()), // fin del metodo onNext
  error => console.error(error); // fin del metodo onError
);
```

Una vez suscritos ejecutaremos alguna de las funciones definidas en función de los estados del observable

Si todo ha sido correcto, para obtener los datos enviados por el servidor usamos el método `json()` del objeto `response`

Siendo más correcto en el uso de la **sintaxis reactiva**

```
this.http.get(url)
.map(response => response.json())
.subscribe(
  response => console.log(response), // fin del mettdodo onNext
  error => console.error(error); // fin del mettdodo onNext
);
```

Utilizamos el operador `map` para aplicar una transformación al observable antes de suscribirnos a él.

### Respuesta al servicio HttpClient

```
this.http.get(url)
// .map(response => response.json())
.subscribe(
  response => console.log(response), // fin del mettdodo onNext
  error => console.error(error); // fin del mettdodo onNext
);
```

En el nuevo servicio se accede directamente a la respuesta en formato JSON, sin necesidad de manipulaciones previas para obtener dicho formato

```
this.aDatos = this.http.get(url)
// .map(response => response.json())
.subscribe(
  response => {
    console.log(response), // fin del mettdodo onNext
    this.aDatos = response
  }
  error => console.error(error); // fin del mettdodo onNext
);
```

## Servicios propios

jueves, 14 de septiembre de 2017 22:52

Al hacer una petición REST con `Http` / `HttpClient` obtenemos un objeto `Response` que debe ser procesado de acuerdo con las características del API concreto del que proceden los datos

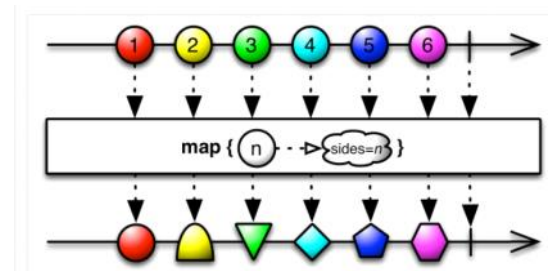
En lugar de utilizar directamente dichos servicios conviene encapsularlos en otros, capaces de ofrecer objetos de alto nivel a los clientes del servicio (e.g. el array de títulos ya procesado en el ejemplo que hemos visto)

Nuestro propio servicio realizara varias operaciones

- La consulta al API via `Http` / `HttpClient`
- La transformación del objeto `Response` en el conjunto de datos adecuado (e.g. el array de títulos) cuando llegue la respuesta
- El envío de los datos ya transformados con el mismo formato de llegada, para conservar la asincronía del proceso: un `Observable` o una `Promesa`, como los que proporcionan los servicios nativos

```
buscarRx (clave: string) {  
  const url = this.sURL + clave;  
  return this.http.get(url)  
    .map(response => this.extractTitles(response));  
}  
  
private extractTitles(response: any) {  
  if (response.items) {  
    return response.items.map(book => book.volumeInfo.title);  
  } else {  
    return response;  
  }  
}
```

el operador map



## Servicios *stateless* (sin estado)

- No guardan información
- Sus métodos devuelven valores, pero no cambian el estado del servicio
- Ejemplo: BooksService con llamadas a Google

## Servicios *statefull* (con estado)

- Mantienen estado, guardan información
- Al ejecutar sus métodos cambian su estado interno, y también pueden devolver valores
- Ejemplo: LoginService con información en memoria

## ¿*Stateless* vs *statefull*?

- Los servicios *stateless* son más fáciles de implementar porque básicamente encapsulan las peticiones REST al *backend*
- Pero la aplicación es menos eficiente porque cada vez que se visualiza un componente se tiene que pedir de nuevo la información
- Los servicios *statefull* son más complejos de implementar porque hay que definir una política de sincronización entre *frontend* y *backend*
- Pero la aplicación podría ser más eficiente porque no se consulta al *backend* constantemente

# Formularios reactivos

miércoles, 4 de octubre de 2017 6:54

- en lugar de FormsModule, se utiliza ReactiveFormsModule, también incluido en @angular/forms
- el desarrollo declarativo (en la vista es mínimo)
  - el atributo [formGroup] en el elemento form
  - el atributo formControlName para identificar a cada uno de los controles, en cierto modo en lugar del [(ngModel)]
- la gestión del formulario se traslada al controlador, donde se crea un objeto de la clase FormGroup para que se ocupe de ello invocándolo desde el correspondiente atributo de la vista
- Existen 2 posibilidades para instanciar ese objeto
  - Crear el objeto directamente, instanciando cada uno de sus componentes como FormControl
  - utiliza el método group del servicio FormBuilder, que tiene que ser inyectado como cualquier otro servicio
    - este método tiene como parámetro un objeto en el que se definen cada uno los formControlName de cada uno de los controles del formulario, de acuerdo con los valores asignados en la vista. Si es necesario, se puede indicar el valor inicial de los controles

```
<form [formGroup]="formLibros" (ngSubmit)="enviarFormLibros()">
  <label for="titulo">Titulo</label>
  <input type="text" id="titulo" formControlName="titulo">
  <label for="autor">Autor</label>
  <input type="text" id="autor" formControlName="autor">
  <label for="editorial">Editorial</label>
  <input type="text" id="editorial" formControlName="editorial">
  <label for="fecha">Fecha (Año)</label>
  <input type="text" id="fecha" formControlName="fecha">
  <label></label>
  <button type="submit">Enviar</button>
</form>
```

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

@Component({
  selector: 'app-formulario',
  templateUrl: './formulario.component.html',
  styleUrls: ['./formulario.component.css']
})
export class FormularioComponent implements OnInit {

  // propiedad de tipo FormGroup (grupo de controles)
  // que se asociara a un formulario o subformulario (en casos complejos)
  formLibros: FormGroup;

  // Se inyecta FormBuilder para instanciar el FormGroup
  // correspondiente a la propiedad que se acaba de definir
  constructor(private FormBuilder: FormBuilder) { }
```

```

ngOnInit() {
    // Gracias al servicio FormBuilder, se instancia un FormGroup
    // p ndole como par metro el objeto con la definici n del formulario
    // con los formControlNames asignados en la vista
    // forControlName="titulo"
    // forControlName="autor"
    // forControlName="editorial"
    // forControlName="fecha">
    this.formLibros = this.formBuilder.group({
        titulo: [],
        autor: [],
        editorial: [],
        fecha: ['2017']
    });
} // Fin del ngOnInit

enviarFormLibros () {}
}

```



# Validación

miércoles, 13 de septiembre de 2017 0:16

**form** → la propia etiqueta HTML está ligada a la directiva Angular **ngForm** (i.e. es su selector), por lo que se instancia automáticamente el correspondiente objeto, que permitirá conocer en todo momento el estado del formulario y de cualquiera de sus controles.

Esta instancia es oculta, pero puede ser accedida en la propia **vista** mediante una **referencia local**

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

← referencia local que acceda a la instancia del formulario

El acceso desde el **modelo/controlador** se consigue gracias al decorador **@ViewChild**

```
@ViewChild('myform') form: any;  
...  
console.log(this.form);
```

```
▼ NgForm {_submitted: false, ngSubmit: EventEmitter, form: FormGroup} ⓘ  
  control: (...)  
  controls: (...)  
  dirty: (...)  
  disabled: (...)  
  enabled: (...)  
  errors: (...)  
  form: FormGroup {validator: null, asyncValidator: null, _onCollectionChange: f,  
    formDirective: (...)  
    invalid: (...)  
  }  
  ngSubmit: EventEmitter {_isScalar: false, observers: Array(1), closed: false, is  
    path: (...)  
    pending: (...)  
    pristine: (...)  
    statusChanges: (...)  
    submitted: (...)  
    touched: (...)  
    untouched: (...)  
    valid: (...)  
    value: (...)  
    valueChanges: (...)  
    _submitted: false  
  }  
  __proto__: ControlContainer
```

## Requerimientos y estado

Los requerimientos de validación se establecen directamente con los nuevos atributos incorporados en HTML5

- **required**: valor booleano: cuando es true marca un campo como obligatorio.
- **max**: indica el número máximo de caracteres permitidos en un campo.
- **min**: indica el número mínimo de caracteres permitidos en un campo.
- **pattern**: Valida un campo frente a una expresión regular (regex).

El estado del formulario y de cada control viene definido por el valor de una serie de propiedades

- **Untouched**: When true, the control has not been interacted with the user
- **Touched**: When true, the control has been interacted with the user
- **Pristine**: The control and its underlying model has not been changed
- **Dirty**: The control and its underlying model has been changed

Estas propiedades permiten no mostrar mensajes de validación hasta que el usuario ha comenzado a rellenar el formulario

- **Valid**: The inner model is valid
- **Invalid**: The inner model is not valid

Estas propiedades permiten determinar la validez de cualquier control para hacer visibles o no los correspondientes mensajes, e.g utilizando el atributo **hidden**.

Cuando se renderiza el HTML, aquellas propiedades que valgan true darán lugar a la aplicación de las correspondientes clases de CSS.

Estas propiedades son accesibles desde la referencia local del formulario

`myform.form.controls.firstname` ← *name* asignado a cada uno de los controles

Pero es más sencillo crear referencias locales específicas para cada control

```
<input name="firstname" ... #firstnameState="ngModel">
```

## Información al usuario

Si no se cumplen los requerimientos de validación, el navegador responderá en la forma que tenga predefinida en función de la validación HTML5. Para evitar estos mensajes se utiliza el atributo **novalidate** en la etiqueta `form`.

El siguiente paso es crear los mensajes específicos de cada situación y ocultarlos o mostrarlos en función del valor de las propiedades antes citadas. Para acceder a ellas se definen referencias locales (#) en cada uno de los controles

```
<form name="myform" novalidate (ngSubmit)="enviar()">

  <input type="text" id="firstname"
    name="firstname"
    [(ngModel)]="user.firstname"

    required="true"
    minlength="2"

    #firstnameState="ngModel">

  <!--Mensajes de validación-->
  <div class="error-message"
    [hidden]="firstnameState.valid || firstnameState.pristine">
    El nombre es obligatorio</div>
```

Anulamos la validación estándar HTML5

input con doble binding

requerimientos de validación

referencia local

condiciones en las que se oculta el mensaje de error de validación

Para cada directiva de validación existe una propiedad `errors` que tomara un valor según las circunstancias, creándose un objeto correspondiente al primero de los errores que se esté produciendo

```
{{firstnameState.errors?.required}}
{{firstnameState.errors?.minlength}}
```

Para mostrarlos se utiliza el **operador Elvis**, para que solo se intente llamar la propiedad de la derecha si la de la izquierda no es nula