



# Angular S7: Pipes | Lazy loading

## Después de esta lección podrás:

1. Trabajar con Pipes custom en aplicaciones Angular.
2. Implementar Lazy loading en los modulos Angular de nuestra aplicación.

¡Y aquí estamos! Como los otros capítulos cubren la mayor parte del contenido Angular necesario para desarrollar una aplicación, estamos listos para comenzar a desarrollar, pero bueno, aprenderemos dos pequeñas cosas antes de ir de frente y convertirse en un profesional.

## Pipes

Los pipes son una herramienta realmente útil que podemos aplicar a nuestro código angular. Es posible que ya haya visto algunos, ya que se pueden aplicar directamente en la template para cambiar la salida de las variables que vinculamos.

Vamos a mostrar el poder de los pipes a través de un ejemplo. Tendremos una variable **message** en **app.component.ts**:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  message: string;

  constructor() {
    this.message = 'This is a default message.';
  }
}
```

Y mostrarlo en la plantilla:

```
<h2>{{ message }}</h2>
```

**This is a default message.**

Entonces, ¿dónde está ese Pipe y cómo puedo usarlo?

Fácil, usaremos una los Pipes predeterminados en Angular, **uppercase**, que hace lo que dice el nombre, cambiando un texto a mayúscula:

```
<h2>{{ message | uppercase }}</h2>
```

**THIS IS A DEFAULT MESSAGE.**

## Pipes personalizadas

Para mostrar lo que podemos lograr con un Pipe personalizado, ¿por qué no intentamos crear una entrada de filtro?

En primer lugar, creemos un componente de lista:

```
ng g c components/list --module=app.module
```

Hay que tener en cuenta que aquí estamos usando un indicador - **module**, esta es una manera de resolver un problema al crear un nuevo componente después de tener diferentes **app.module.ts** en la carpeta de la aplicación.

Agregamos una lista a nuestro componente y una función para filtrar los elementos de la lista y devolver una nueva lista:

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

@Component({
  selector: 'app-list',
  templateUrl: './list.component.html',
  styleUrls: ['./list.component.scss']
})
export class ListComponent implements OnInit {
  countryList: string[];
  filteredList: string[];
  filter: string;

  constructor() {
    this.filter = '';
    this.countryList = ['Spain', 'France', 'Portugal', 'Germany', 'Switzerland', 'Sweden', 'Holland', 'Italy', 'United Kingdom'];
    this.filteredList = this.countryList;
  }

  ngOnInit() {}

  onChangeFilter(filter: string) {
    const newList: string[] = this.countryList.filter(el => el.toLowerCase().includes(filter.trim().toLowerCase()));
    this.filteredList = newList;
  }
}
```

¿Qué tenemos aquí?

1. Una lista de países, que será la fuente de la verdad para nuestro filtro.
2. Una lista filtrada, que almacenará la lista que veremos en la plantilla.
3. Un filtro, que se asignará a una entrada.

Y la plantilla para el componente de la lista:

```
<h2>This is the country list</h2>
<div>
  <h4 *ngFor="let country of filteredList">{{ country }}</h4>
</div>

<h3>Change the filter by writing here ✎</h3>
<input type="text" [(ngModel)]="filter" (ngModelChange)="onChangeFilter($event)" />
```

El resultado:

---

## This is the country list

Spain

France

Portugal

Germany


Switzerland

Sweden

Holland

Italy

United Kingdom

Change the filter by writing here 

¿Pero dónde están las Pipes prometidos?

No te preocupes! Esta fue la forma de preparar un gran escenario de Pipes. ¿Qué tal un Pipe que hace todo este filtro automáticamente?

¡Suenan bien! ¡Sigamos y comencemos a trabajar en eso! 🚀

Antes de crear el Pipe, crearemos un nuevo archivo **app-pipes.module.ts**, que será el módulo donde almacenaremos todas nuestros Pipes personalizados:

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

@NgModule({
  imports: [],
  declarations: [],
  exports: []
})
export class AppPipesModule { }
```

Y importarlo en **app.module.ts**:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { ReactiveFormsModule, FormsModule } from '@angular/forms';

import { AppRoutingModule } from './app-routing.module';
import { AppPipesModule } from './app-pipes.module';

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

@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
    AppRoutingModule,
    AppPipesModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

A partir de ahora, todos nuestros Pipes se importarán en **app-pipes.module.ts** y se importarán a nuestra aplicación a través de la importación en app.module.ts. Esto conducirá a un código y una arquitectura más limpios.

Ahora crearemos un Pipe a través de la CLI:

```
ng generate pipe pipes/filter-names --module=app-pipes.module
```

Como antes, estamos eligiendo un módulo, pero el nuevo en este caso, que se ve así:

```
import { NgModule } from '@angular/core';
import { FilterNamesPipe } from './pipes/filter-names.pipe';

@NgModule({
  imports: [],
  declarations: [
    FilterNamesPipe
  ],
  exports: [
    FilterNamesPipe,
  ]
})
export class AppPipesModule { }
```

**Recuerda exportar la declaración FilterNamesPipe! No será el comportamiento predeterminado de la CLI y debemos exportarlo con este patrón de módulo.**

Pasemos ahora al archivo de Pipe y creemos el método de filtro dentro de la función de transformación que viene

creada por defecto:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({ name: 'priorityName' })
export class PriorityNamePipe implements PipeTransform {
  transform(list: string[], filter: string = '') {
    const lowerCaseFilter: string = filter.toLowerCase().trim();

    const filteredList: string[] = list.filter((el: string) => {
      return el.toLowerCase().includes(lowerCaseFilter);
    });

    return filteredList;
  }
}
```

El método de transformación recibió el elemento al que estamos aplicando el Pipe como primer argumento, que es la lista en este caso.

Los otros argumentos, se envían a el Pipe utilizando una nomenclatura especial, **{valor | pipe: arg}**.

¡Y así, creamos un Pipe de filtro personalizado! Podemos usarlo con el nombre **filterNames**, así que vamos a nuestro componente y ¡aplicarlo!

En primer lugar, limpia el archivo ts del componente:

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

@Component({
  selector: 'app-list',
  templateUrl: './list.component.html',
  styleUrls: ['./list.component.scss']
})
export class ListComponent implements OnInit {
  countryList: string[];
  filter: string;

  constructor() {
    this.filter = '';
    this.countryList = ['Spain', 'France', 'Portugal', 'Germany', 'Switzerland', 'Sweden', 'Holland', 'Italy', 'United Kingdom'];
  }

  ngOnInit() { }
}
```

¡Esto es mucho más limpio, nos deshacemos de la lista de filtros de respaldo y solo tenemos el original!

En la plantilla, aplicar el Pipe:

```
<h2>This is the country list</h2>
<div>
  <h4 *ngFor="let country of (countryList | filterNames: filter)">{{ country }}</h4>
</div>

<h3>Change the filter by writing here ✎</h3>
<input type="text" [(ngModel)]="filter" />
```

Esto es un buen pipe!!!

## This is the country list

Spain

France

Portugal

Germany


Switzerland

Sweden

Holland

Italy

United Kingdom

Change the filter by writing here 

¡Estas son los Pipes personalizados! ¡La forma correcta de lograr un código y una arquitectura limpios y organizados en Angular!

## Lazy Loading

**Esta técnica usada en Angular** nos permite cargar sólo, el o los componentes que necesitemos al inicio de nuestra aplicación, estos componentes no cargan cada vez que entres, sino que solo cargan una sola vez.

Cuando usamos Lazy Loading hacemos llamado de un módulo mediante el sistema de rutas de Angular y este módulo a su vez tiene rutas hijas que se encargan de cargar el componente solicitado por el usuario, más adelante entenderemos esto mejor.

Vamos a crear el proyecto:

```
ng new angular-lazy-loading --routing --style=css && cd angular-lazy-loading
```

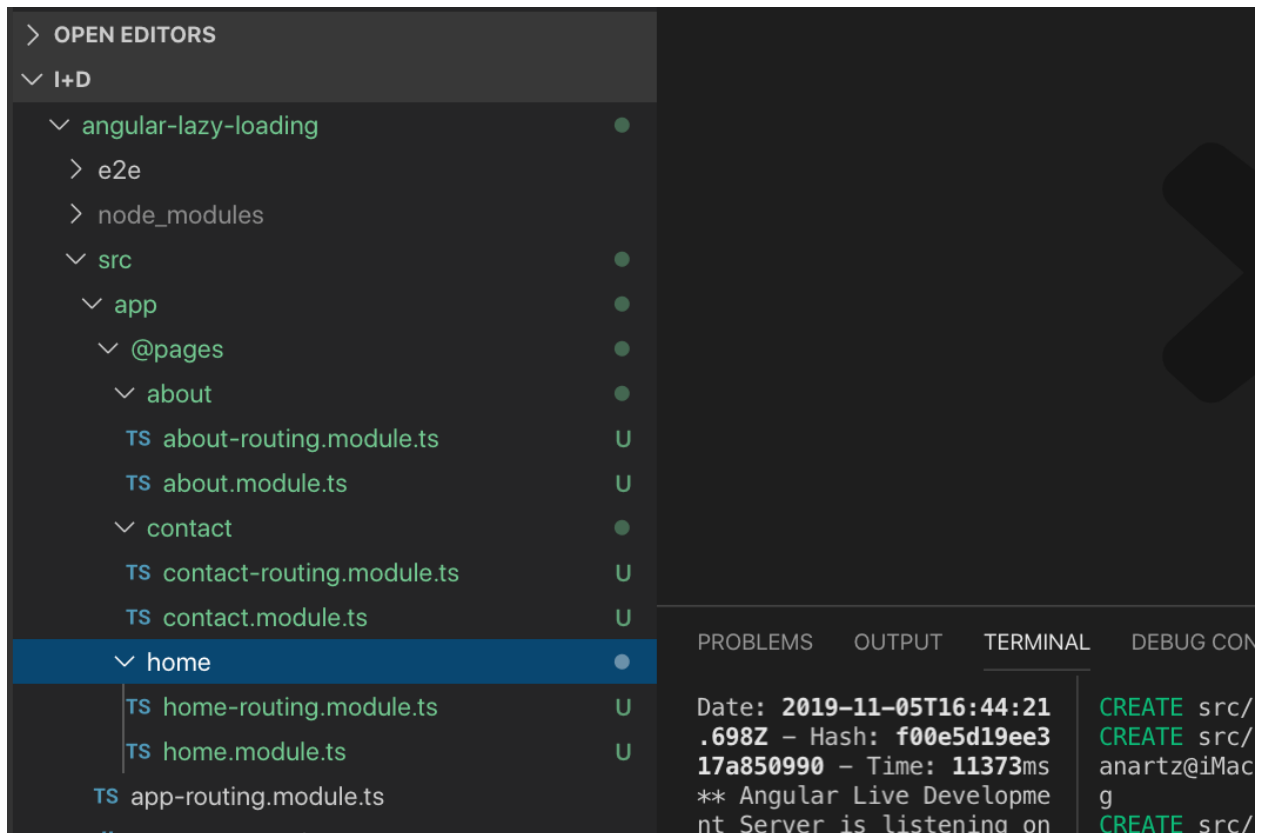
`--routing`: Para que nos genere un módulo de ruta listo para funcionar.

`--style=css`: Para que nos añada las hojas de estilos CSS.

Ahora que ya tenemos nuestro proyecto creado, el siguiente paso que vamos hacer es generar 3 componentes. Vamos a crear los componentes llamados home, about, contact con sus módulos que se encargarán de gestionar la carga de la información mediante la gestión de sus rutas hijas

```
ng g m @pages/home --routing
ng g m @pages/about --routing
ng g m @pages/contact --routing
```

El resultado será el siguiente donde podremos ver que se han creado dos ficheros por carpeta:





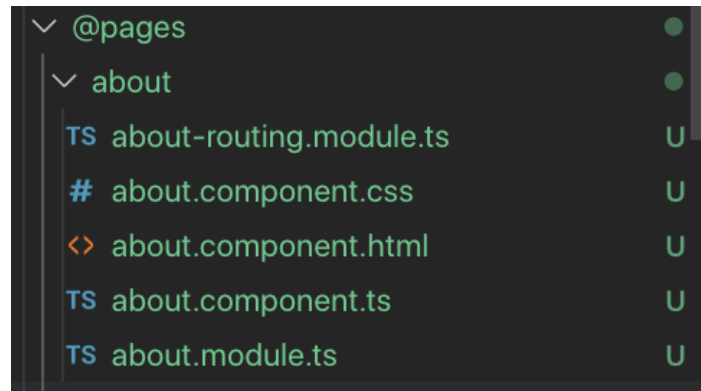
Uno de ellos es el módulo principal donde vamos a cargar los componentes, módulos que necesitemos y el modulo de enrutamiento.

El otro fichero es el que corresponde al módulo de enrutamiento donde vamos a cargar el componente con una ruta hija.

Creamos los componentes:

```
ng g c @pages/home --skipTests=true
ng g c @pages/about --skipTests=true
ng g c @pages/contact --skipTests=true
```

Estos serían los ficheros que compondrían una página que se va a cargar mediante Lazy Loading. En este caso os muestro el que corresponde a "about"



Ahora vamos a trabajar sobre el módulo **about-routing.module.ts**. Dentro de este archivo creamos la ruta hija que servirá para cargar el componente **about.component.ts** desde el módulo **about.module.ts**.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { AboutComponent } from './about.component';

const routes: Routes = [
  {
    path: '', component: AboutComponent
  }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class AboutRoutingModule { }
```

Primero importamos los componentes que van a depender de ese módulo, creamos las rutas y luego dentro de **imports** hacemos uso de **forChild** para especificar que ese módulo va a servir rutas hijas.

En el fichero **about.module.ts** ya tenemos importado el AboutRoutingModule.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { AboutRoutingModule } from './about-routing.module';
import { AboutComponent } from './about.component';

@NgModule({
  declarations: [AboutComponent],
  imports: [
    CommonModule,
    AboutRoutingModule // AQUÍ PODEÍS VERLO
  ]
})
export class AboutModule { }
```

Hacemos lo mismo pero con las páginas home y contact.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home.component';

const routes: Routes = [
  {
    path: '', component: HomeComponent
  }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class HomeRoutingModule { }
```

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ContactComponent } from './contact.component';

const routes: Routes = [
  {
    path: '', component: ContactComponent
  }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class ContactRoutingModule { }
```

Ahora vamos a nuestro módulo de rutas principales `app-routing.module.ts` que quedará de la siguiente manera:

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

const routes: Routes = [
  {
    path: 'home', loadChildren: () =>
      import('./pages/home/home.module').then(m => m.HomeModule)
  },
  {
    path: 'about', loadChildren: () =>
      import('./pages/about/about.module').then(m => m.AboutModule)
  },
  {
    path: 'contact', loadChildren: () =>
      import('./pages/contact/contact.module').then(m => m.ContactModule)
  },
  { path: '', redirectTo: 'home', pathMatch: 'full' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Dentro de nuestra variable **routes** vamos almacenar todas las rutas principales de nuestra aplicación.

1. Ruta **home**, aquí le decimos a Angular, cuando la ruta sea igual a **home** carga el **hijo haciendo referencia al HomeModule**. Si os fijáis aquí ya está aplicando el **Lazy Loading**.
2. Ruta **about** y **contact**, hacemos lo mismo pero haciendo las referencias a los módulos que les corresponden.
3. La última ruta le dice a Angular, cuando no especifique una url, llévame a la ruta **home**.

El último paso es visualizar la información navegando entre diferentes páginas. Accedemos al fichero **app.component.html** y borramos todo el contenido para añadir el siguiente:

```
<div style="text-align:center">
  <h1>
    Lazy Loading
  </h1>
  <ul>
    <li><a routerLink="/home">Home</a></li>
    <li><a routerLink="/about">About</a></li>
    <li><a routerLink="/contact">Contact</a></li>
  </ul>
</div>
<router-outlet></router-outlet>
```

Dejamos tres enlaces para navegar entre las diferentes páginas y el elemento "router-outlet" para cargar esas páginas.

Para notar el cambio abrimos la consola del navegador y vamos a la pestaña “Red” (es necesario reiniciar la página para poder capturar los archivos en esta pestaña).

home works!

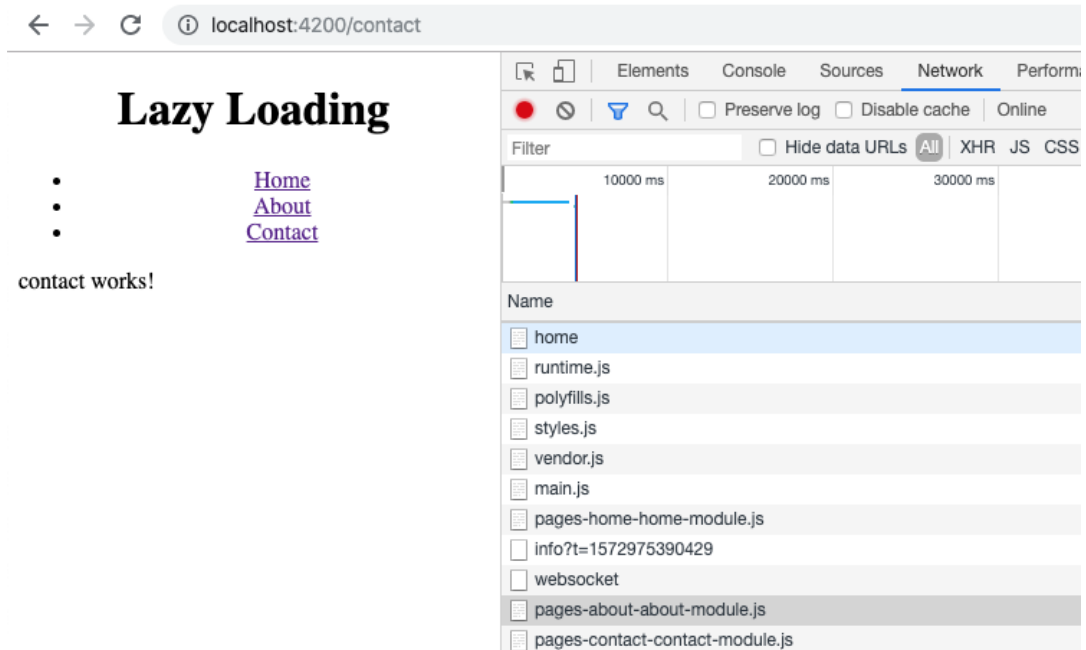
Name	Status	Type
home	200	document
runtime.js	200	script
polyfills.js	200	script
styles.js	200	script
vendor.js	200	script
main.js	200	script
pages-home-home-module.js	200	script
info?t=1572975390429	200	xhr
websocket	101	websocket

Ahora vamos hacer click a nuestro enlace para ir a la página “about”

about works!

Name	Status	Type
home	200	document
runtime.js	200	script
polyfills.js	200	script
styles.js	200	script
vendor.js	200	script
main.js	200	script
pages-home-home-module.js	200	script
info?t=1572975390429	200	xhr
websocket	101	websocket
pages-about-about-module.js	200	script

Y lo mismo con la página “contact”



Si volvemos a cargar cualquiera de las páginas, ya no se cargarán los ficheros porque ya los tendremos cargados con la ejecución inicial de cada uno.