TEMA 5-6-7

PWA, Angular Universal, Deploy

Desarrollo front-end avanzado

Máster Universitario en Desarrollo de sitios y aplicaciones web

Universitat Oberta
de Catalunya

Contenido

Introducción

En esta sección de teoría vamos a estudiar tres grandes bloques.

Primero estudiaremos que es una **progressive web app**, en este bloque es en el que pondremos mayor énfasis ya que es un tema muy interesante y que actualmente se está utilizando mucho. Después estudiaremos una introducción a **angular universal** y finalmente también estudiaremos una introducción a **schematics**. Concluimos esta PEC viendo como se debe **desplegar** nuestra aplicación frontend en un servidor de producción.

Progressive Web App

1.1. ¿Qué es una Progressive Web App?

Una **Progressive Web App** (PWA) es una aplicación web que puede ser consultada en un dispositivo móvil como si fuera una aplicación nativa. Realmente las PWAs no están centradas exclusivamente en el desarrollo de aplicaciones web que puedan funcionar en entornos nativos, sino que se centran en el concepto de ser progresivas. De hecho, una PWA debería cumplir las siguientes propiedades:

- **Progresiva**: debe funcionar para todos los usuarios, sin importar la elección del navegador.
- **Adaptable**: se adapta a cualquier factor de forma, es decir, debe poder ejecutarse tanto en escritorio, móvil, tablet o lo que venga en el futuro (p.ej. smartTV).
- **Independiente de la conectividad**: la conectividad del dispositivo no debe ser un limitante, sino que se debe poder disponer de la misma experiencia de usuario tanto si se dispone de conectividad a internet alta, media o baja. Esta característica es conseguida gracias a los **service workers**.
- **Estilo app**: la experiencia de usuario debe proporcionar un **look&feel** como el de una app nativa con interacciones y estilo de navegación clásico de una app móvil. Para ello se ha utilizado un patrón de diseño denominado App Shell.
- **Fresca**: siempre se encuentra actualizada, al ser una aplicación web que está constantemente siendo cacheada utilizando los **service workers**.
- **Segura**: solo funciona sobre HTTPS.
- **Instalable**: debe poderse instalar en los dispositivos de modo fácil.
- **Vinculable**: se puede compartir fácilmente proporcionando la URL, no requiere instalación compleja, ni descarga de datos para su instalación.

Si quieres profundizar en el concepto de lo que es una PWA, así como por qué deberían utilizarse, puedes leerse el siguiente enlace:

https://blog.bitsrc.io/what-is-a-pwa-and-why-should-you-care-388afb6c0bad

1.2. Tecnología que utiliza una PWA

Las PWAs están compuestas principalmente de tres elementos:

1. Service Worker.
2. Manifest.json.
3. HTTPS.

A continuación, vamos a describir cada uno de ellos.

Service Worker

Un **Service Worker** es parte del estándar de JavaScript. Éste actúa como proxy entre el navegador y la conexión del usuario. Un **Service Worker** gestiona las notificaciones **push** y permite construir aplicaciones offline, dando lugar al concepto de **offline first web apps** gracias al uso de la API caché del navegador.

Es decir, el **Service Worker** es el encargado de permitir que la aplicación no esté nunca offline (i.e. error 404) y de proporcionar las notificaciones **push** (a través de la API de notificaciones del propio estándar JavaScript).

Otra interesante característica es la ganancia de velocidad de ejecución (i.e. rendimiento) de la aplicación debido a que la aplicación se carga desde la caché, sin necesidad de tener en cuenta si existe conectividad en el dispositivo.

Manifest.json

El fichero **manifest.json** es un fichero de configuración JSON el cual contiene información de la aplicación como, por ejemplo, el icono que se mostrará en la pantalla del dispositivo una vez instalada, el nombre corto de la aplicación, el color de fondo o incluso el tema.

Cuando existe un fichero **manifest.json**, el navegador disparará automáticamente un banner de instalación en la web de modo que si el usuario acepta, la PWA se instalará. Actualmente el navegador que mejor soporta esta funcionalidad es Chrome.

Un ejemplo de un fichero **manifest.json** es el siguiente:

```
{
 "short_name": "AirHorner",
 "name": "Kinlan's AirHorner of Infamy",
 "icons": [
   {
     "src": "launcher-icon-1x.png",
     "type": "image/png",
     "sizes": "48x48"
   },
   {
     "src": "launcher-icon-2x.png",
     "type": "image/png",
     "sizes": "96x96"
   },
   {
     "src": "launcher-icon-4x.png",
     "type": "image/png",
     "sizes": "192x192"
   }
 ],
 "start_url": "index.html?launcher=true"
}
```

HTTPS

Las PWAs requieren HTTPS para funcionar, dejando de lado el antiguo HTTP.

1.3. Herramientas y bibliotecas

Existen diversos proyectos que facilitan la creación de PWAs. Los dos principales proyectos son los siguientes:

1. **Lighthouse** es una herramienta para auditar las PWAs, se genera un reporte completo de buenas prácticas para construir una PWA. Se instala como un plugin en el navegador Google Chrome.

2. **Workbox** es una biblioteca Open Source también desarrollada por Google que puede generar el fichero de Service Worker.

Vamos a crearnos nuestra **pwa** de ejemplo para introducirnos en esta tecnología.

Creación del proyecto base

Después de haber visto un poco de teoría, vamos a implementar nuestra propia **PWA**. Vamos a implementar un ejemplo sencillo, para tener una visión inicial de las posibilidades que ofrece la tecnología **PWA**. Un punto interesante que quedaría por tratar, pero que queda fuera del alcance de este temario, sería la gestión de notificaciones **push**, lo cual recomendamos que indaguéis un poco por vuestra cuenta.

Con todo esto, lo primero que haremos será crearnos un proyecto nuevo e implementaremos una vista maestro detalle utilizando la api que vimos en el tema 3, concretamente:

https://picsum.photos/

Se trata de una API pública que podemos utilizar para devolver un listado de imágenes, una imagen concreta en función de un identificador, …

Suficiente para implementar una aplicación sencilla con la típica vista lista-detalle y poder estudiar una introducción a cómo transformar esta aplicación a **PWA**.

A continuación, enumeramos una propuesta de paso a paso a seguir:

- **Paso 1:**

Creamos un nuevo proyecto ejecutando el siguiente comando desde la consola:

```
C:\Projectes>ng new angular-pwa-introduction
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? CSS
```

ng new << nombre_proyecto >>

Podemos indicar que si queremos que nos añada el **routing** de **Angular** y por ejemplo que utilizaremos el tipo de estilos **CSS**.

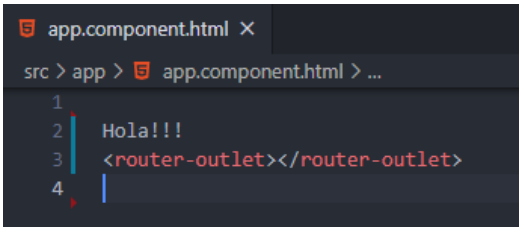
Ejecutamos el proyecto con la instrucción:

```
C:\Projectes\angular-pwa-introduction>ng serve --open
```

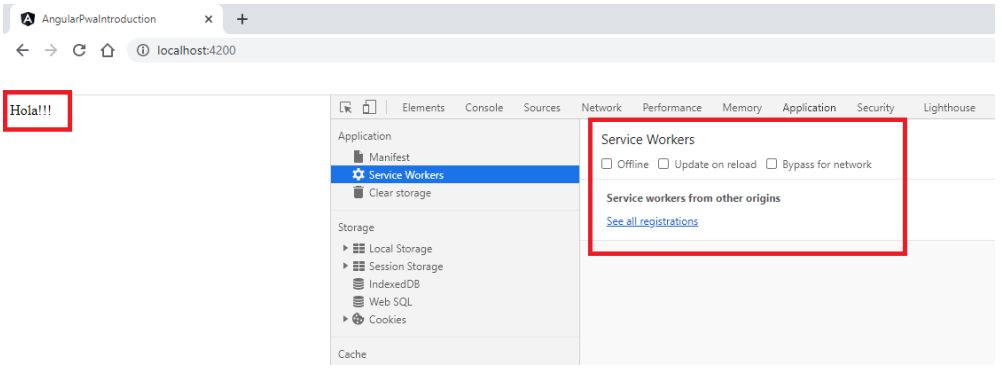
ng serve --open

Y validamos que vemos la pantalla inicial del proyecto en el navegador.

Veremos la pantalla inicial con mucha información de **Angular** que en principio no nos interesa, así que iremos al fichero **app.component.html** y eliminaremos todo su contenido a excepción de la línea del **router-outlet** y, además, pongamos algún texto, simplemente para que se muestre alguna cosa para mostrar el modo **offline**:

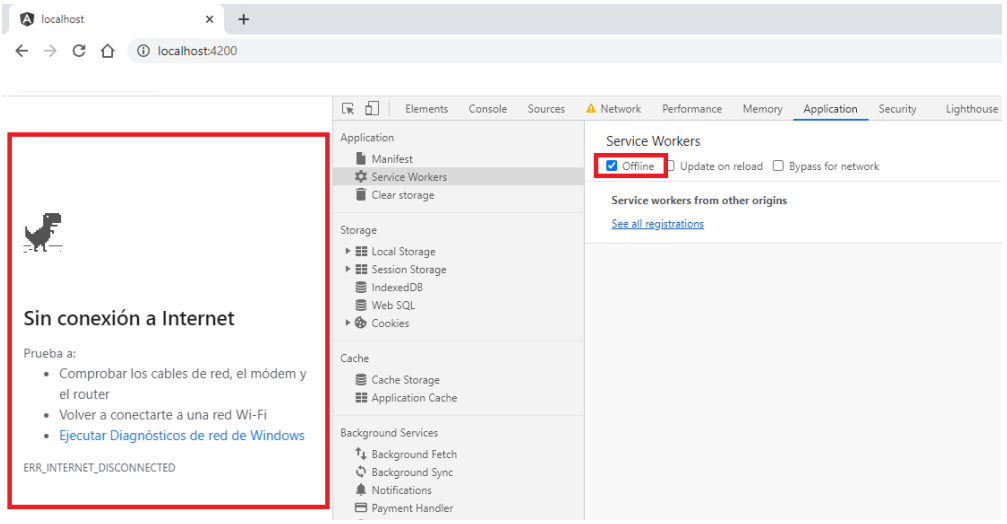


Ahora en el navegador veríamos:



Veremos el mensaje en el navegador y podemos ver en la pestaña **Application** (si pulsamos F12 en el **Google Chrome**) como por ahora, no se gestiona ningún **service worker** como es normal, ya que hasta el momento tenemos una aplicación **Angular** normal. (Tampoco tenemos ningún **manifest**)

Hagamos una prueba, si pulsamos el botón de **offline**, para simular que no tenemos conexión a internet y refrescamos, ¿qué pasará?



Pues que la aplicación dejará de funcionar ya que no tiene conexión a internet. Posteriormente, cuando tengamos la **PWA** configurada, podremos observar cómo,

aunque no tengamos conexión a internet, la aplicación continuará funcionando ya que trabajará con la caché, esto lo gestionaremos mediante el **service worker**.

- **Paso 2:**

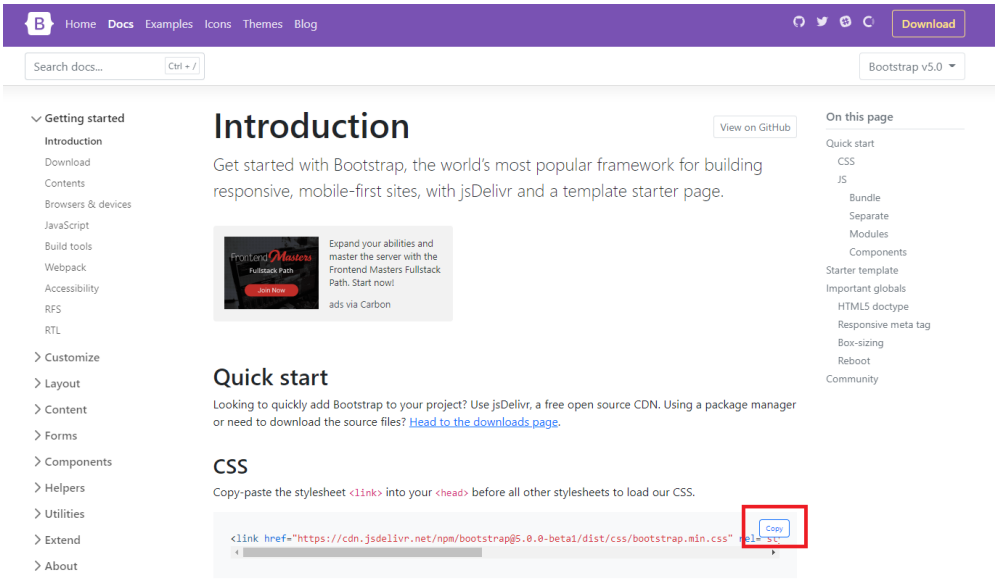
Ahora lo que haremos es añadir algunas bibliotecas a nuestro proyecto para entender cómo gestionar la cache una vez tengamos configurada nuestra **PWA**. Nos vincularemos una librería **dinámica** y otra **estática** para entender cómo gestionarlas mediante una **PWA**.

Por ahora, lo que haremos será añadir los estilos de **Bootstrap** directamente a nuestro **index.html**, de esta manera, tendremos una librería **dinámica**.

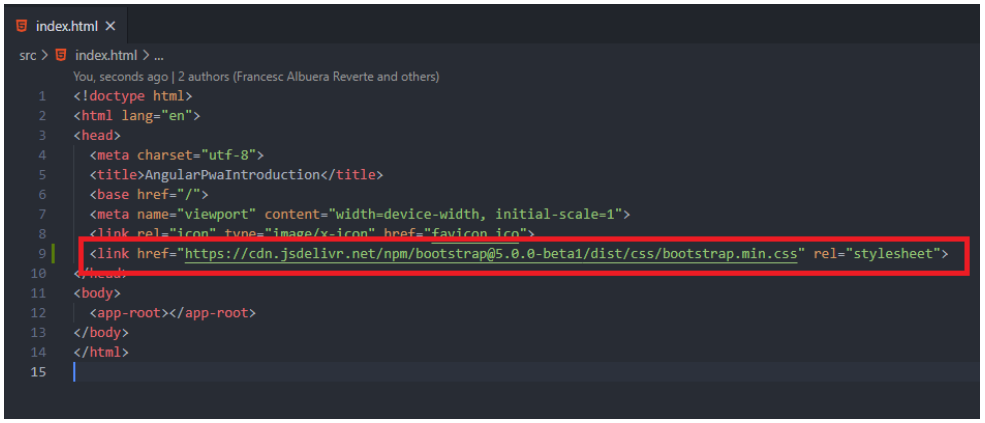
Podemos ir a la página de **Bootstrap**:

https://getbootstrap.com/docs/5.0/getting-started/introduction/

Y copiar directamente los estilos usando el CDN:



Y pegarlos en nuestro **index.html**:



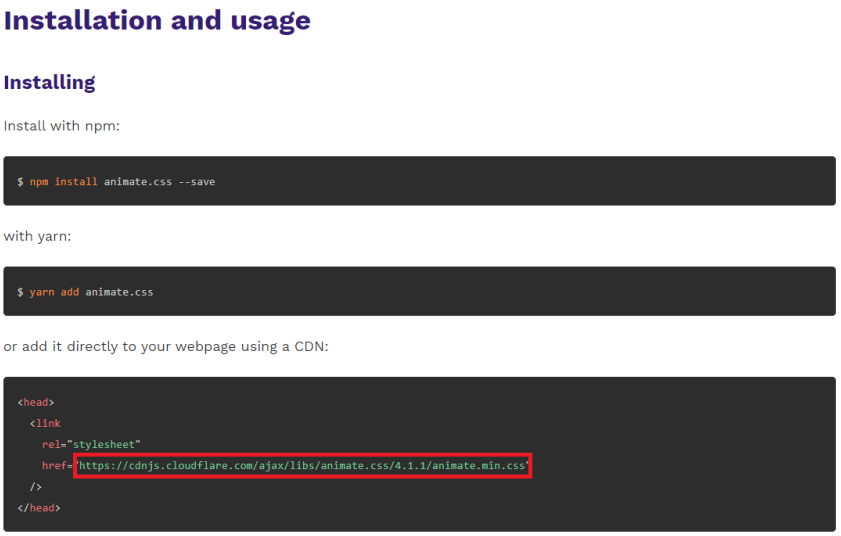
Las propiedades **integrity** y **crossorigin** podemos quitarlas.

Ahora que tenemos enlazada una biblioteca de estilos a nuestro fichero **index.html** y, por lo tanto, la tendremos que cargar/gestionar de manera **dinámica**, vamos a descargarnos otra biblioteca de estilos directamente a nuestro proyecto para de esta manera, tener una biblioteca **estática**, es decir, una biblioteca que tenemos en nuestro propio proyecto, que no será necesario salir a pedirla a internet.

Aquí tenemos muchas posibilidades, por ejemplo, podemos ir a la url:

https://animate.style/

Y en el apartado de *instalación y uso*:



Installation and usage

Installing

Install with npm:

```
$ npm install animate.css --save
```

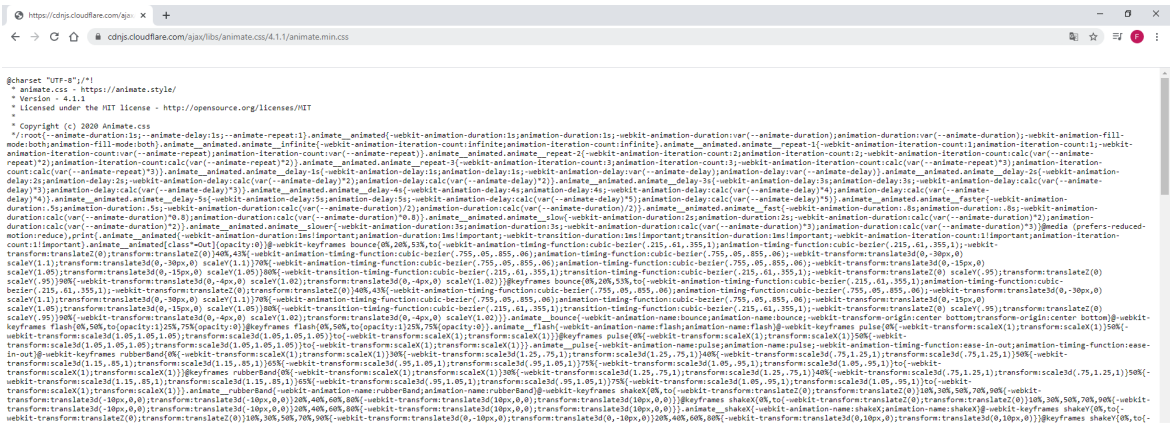
with yarn:

```
$ yarn add animate.css
```

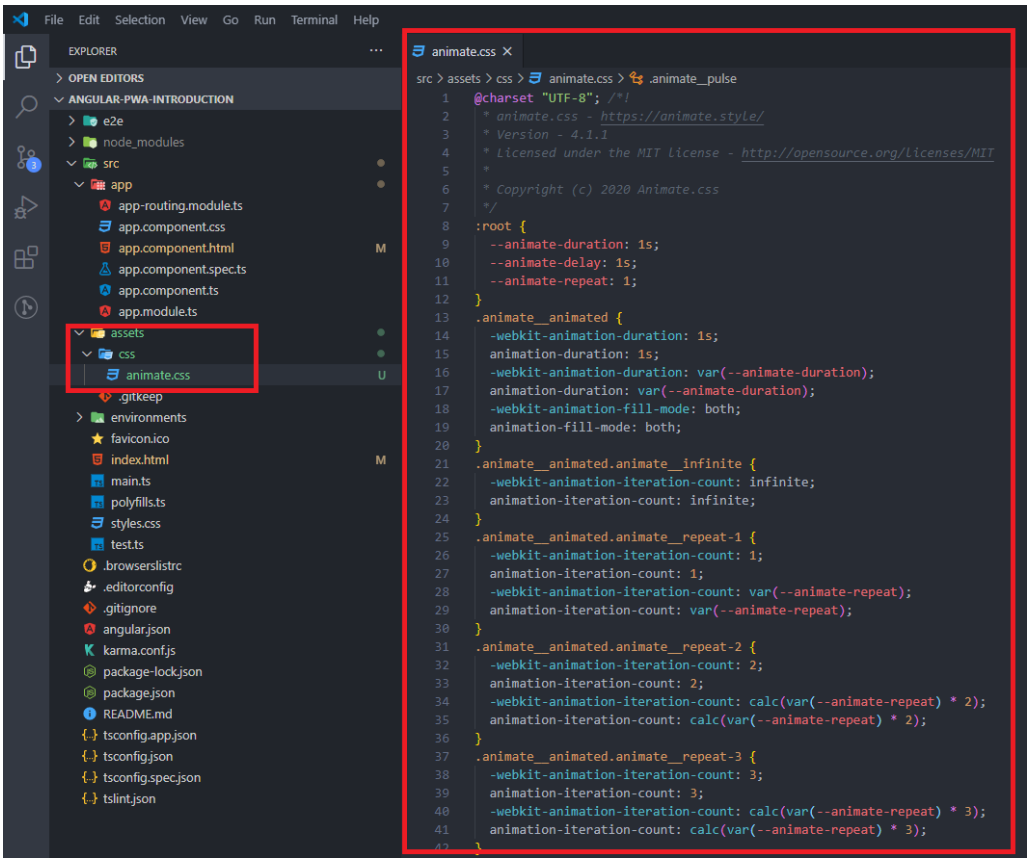
or add it directly to your webpage using a CDN:

```
<head>
  <link
    rel="stylesheet"
    href="https://cdnjs.cloudflare.com/ajax/libs/animate.css/4.1.1/animate.min.css"
  />
</head>
```

Podemos copiar directamente la **url** que está marcada en rojo en el navegador para ver el código completo de la biblioteca:

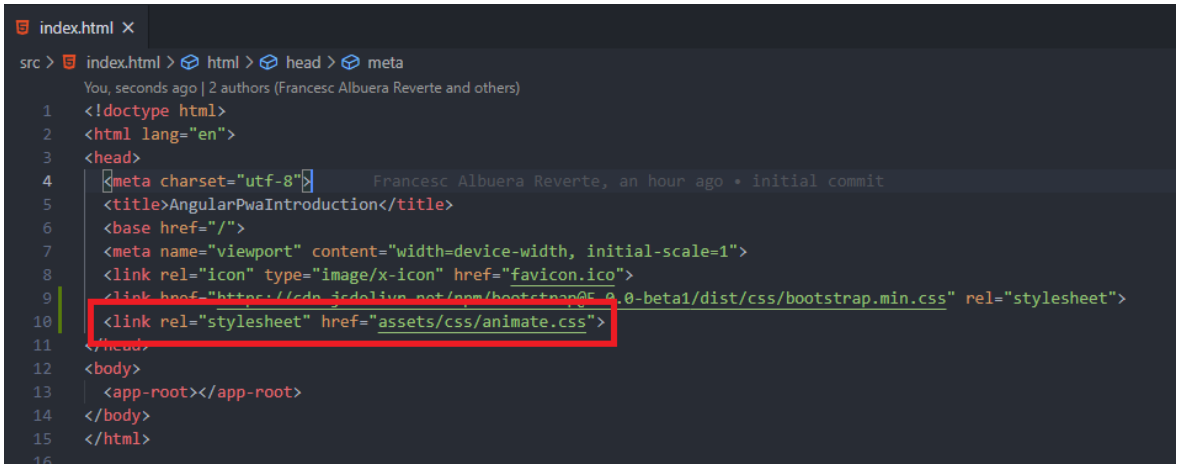


Copiaremos todo el código de esta librería a nuestro proyecto de la siguiente manera:



Dentro de la carpeta **assets creamos** una carpeta **css** y dentro un fichero **animate.css** donde pegaremos el contenido de la biblioteca. Podemos hacer clic derecho en el propio fichero **animate.css** una vez el código copiado y dar a **Format document** para que nos formatee el fichero ya que viene en formato reducido, pero no es importante esto, simplemente queremos una biblioteca en nuestro propio proyecto para manejarla de manera **estática**.

Por lo tanto, lo último que nos quedaría es vincular a nuestro **index.html** esta biblioteca estática:



- **Paso 3:**

Vamos a crear los componentes iniciales.

```
C:\Projectes\angular-pwa-introduction>ng g c components/images
CREATE src/app/components/images/images.component.html (21 bytes)
CREATE src/app/components/images/images.component.spec.ts (626 bytes)
CREATE src/app/components/images/images.component.ts (275 bytes)
CREATE src/app/components/images/images.component.css (0 bytes)
UPDATE src/app/app.module.ts (486 bytes)

C:\Projectes\angular-pwa-introduction>ng g c components/image
CREATE src/app/components/image/image.component.html (20 bytes)
CREATE src/app/components/image/image.component.spec.ts (619 bytes)
CREATE src/app/components/image/image.component.ts (271 bytes)
CREATE src/app/components/image/image.component.css (0 bytes)
UPDATE src/app/app.module.ts (575 bytes)
```

El componente **images** nos servirá para mostrar el listado de imágenes y el componente **image** nos servirá para mostrar el detalle de cada una de las imágenes.

- **Paso 4:**

Vamos a crearnos el servicio inicial.

Ahora nos crearemos el servicio para poder llamar a la API, esto será muy parecido a lo que hicimos en el tema 3:

```
C:\Projectes\angular-pwa-introduction>ng g s services/images
CREATE src/app/services/images.service.spec.ts (357 bytes)
CREATE src/app/services/images.service.ts (135 bytes)
```

- **Paso 5:**

Vamos a crearnos las rutas de navegación necesarias:

app-routing.module.ts

src > app > app-routing.module.ts > ...

You, seconds ago | 2 authors (Francesc Albuera Reverte and others)

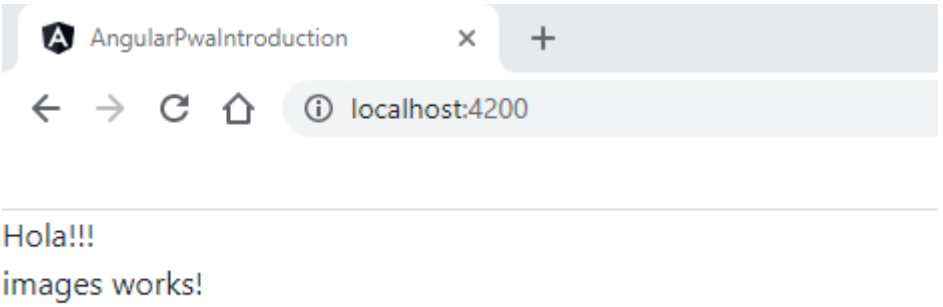
```typescript
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ImageComponent } from './components/image/image.component';
import { ImagesComponent } from './components/images/images.component';

const routes: Routes = [
  { path: '', component: ImagesComponent},
  { path: 'image/:id', component: ImageComponent},
  { path: '**', component: ImagesComponent}
];

Francesc Albuera Reverte, an hour ago | 1 author (Francesc Albuera Reverte)
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Básicamente le diremos que la ruta vacía redireccionará al listado de imágenes.

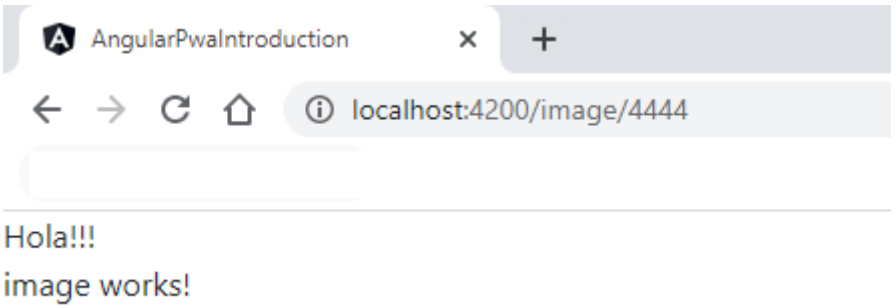
Por otro lado, para el detalle de una imagen definiremos una ruta en la que le pasaremos por ruta el **id** de la imagen determinada y redireccionará al componente **ImageComponent**.

En cualquier otro caso, que redireccione al listado de imágenes.

Si ejecutamos nuestra aplicación deberíamos ver en el navegador algo así:



Si le pasamos por ejemplo por ruta **/image/xxxx** y cualquier cosa como identificador deberíamos ver algo así:



Nótese que vemos **image** en singular, por lo tanto, apuntamos al componente correcto.

- **Paso 6:**

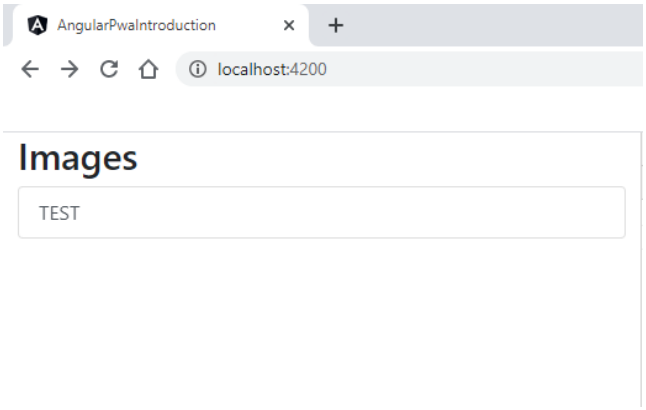
Empecemos a implementar el **images.component.html**:

Creamos un contenedor que contendrá un listado de imágenes, posteriormente esto se cargará con los datos de la API, pero de momento, nos montamos la estructura.

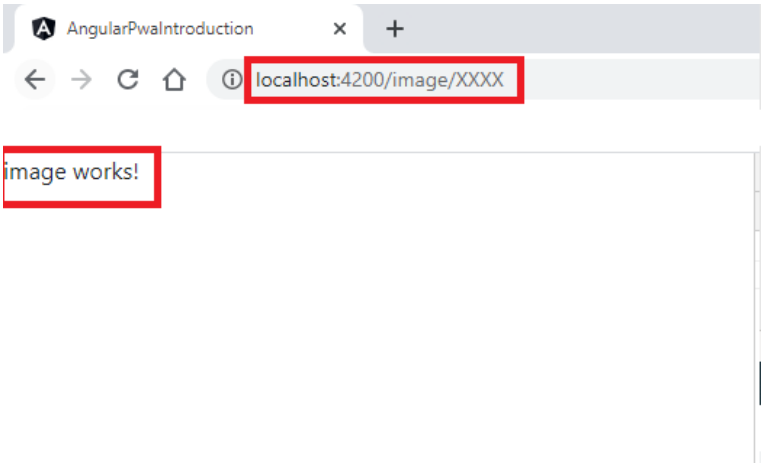
Podemos ver que en cada elemento del listado le hemos añadido las clases **animate__animated animate__bounce animate__fadeIn** para que cada elemento de la lista tenga una pequeña animación. Esto nos servirá para posteriormente ver que aún no teniendo conexión a internet nos funcionará correctamente. Esta biblioteca recordemos que la tenemos en nuestro propio proyecto.

Por otra parte, vemos que cada elemento de la lista que posteriormente contendrá el nombre de cada imagen tiene el **routerLink** para ir a la vista detallada, vemos que le pasaremos dinámicamente el identificador de cada imagen, ahora por el momento, le pasamos la cadena **XXXX** solo para validar que la navegación es correcta.

Con esto, podemos ver que inicialmente tenemos:



Solo tenemos un elemento en la lista de momento, pero podremos observar cómo aparece con una pequeña animación, además, si hacemos clic en este elemento, navegaremos a su vista detalle:

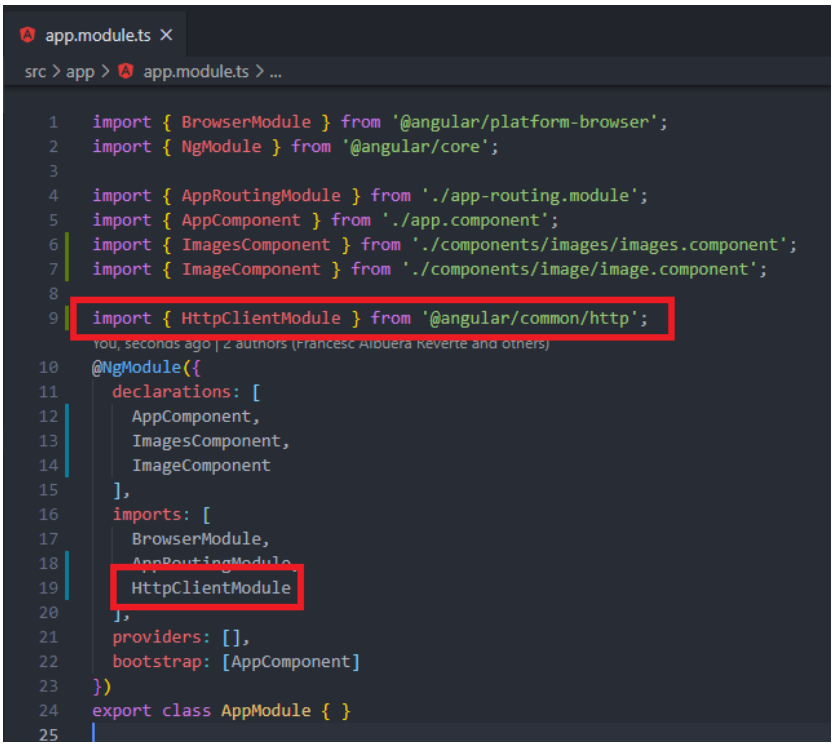


Vemos que navegamos correctamente a la vista detalle, es decir, al componente **image.component.html** y le pasamos por argumento la cadena **XXXX** que posteriormente será el identificador de la imagen.

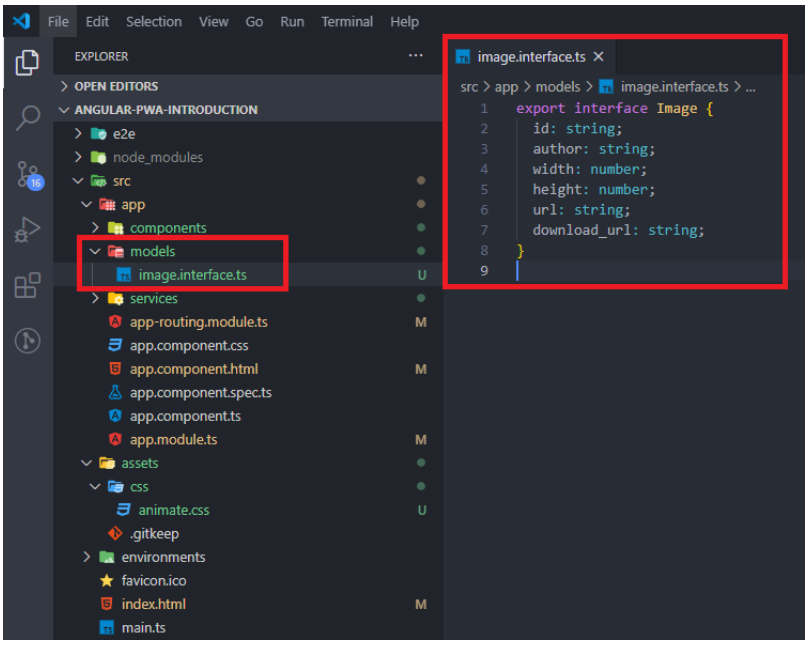
- **Paso 7:**

Implementemos el servicio **images.service.ts**.

Antes de implementar el servicio, como haremos peticiones **http**, necesitaremos importar el módulo:



Como hicimos en el tema 3, creamos la interfaz para mapear la respuesta de la api de la entidad **Image**.



Nos podemos crear una carpeta **models** y dentro el fichero **image.interface.ts** con el código que mostramos en la imagen anterior.

Implementemos el servicio:

images.service.ts

src > app > services > images.service.ts > ...

```ts
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { Image } from '../models/image.interface';

@Injectable({
  providedIn: 'root',
})
export class ImagesService {
  constructor(private http: HttpClient) {}

  getAllImages(): Observable<Image[]> {
    return this.http.get<Image[]>('https://picsum.photos/v2/list');
  }
}

```

- **Paso 8:**

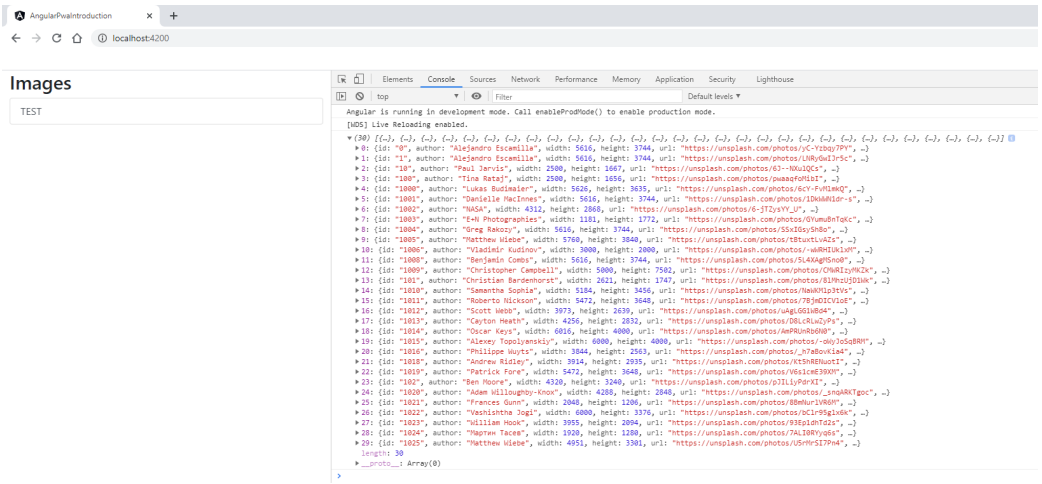
Llamemos al servicio des del componente **images.component**:

images.component.ts

src > app > components > images > images.component.ts > ...

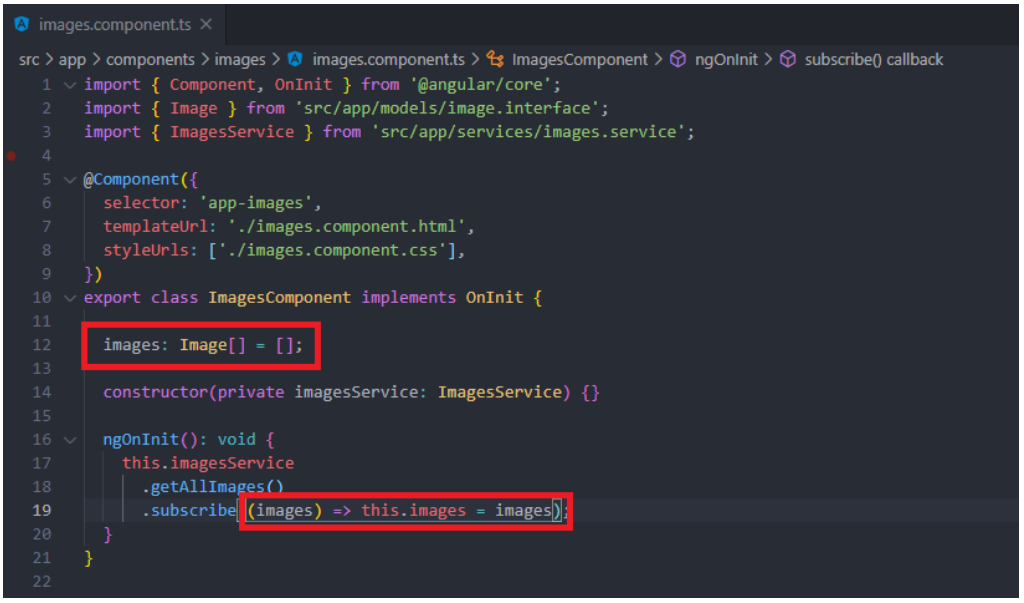
```ts
import { Component, OnInit } from '@angular/core';
import { ImagesService } from 'src/app/services/images.service';

@Component({
  selector: 'app-images',
  templateUrl: './images.component.html',
  styleUrls: ['./images.component.css'],
})
export class ImagesComponent implements OnInit {
  constructor(private imagesService: ImagesService) {}

  ngOnInit(): void {
    this.imagesService
      .getAllImages()
      .subscribe((images) => console.log(images));
  }
}

```

Observemos el navegador:



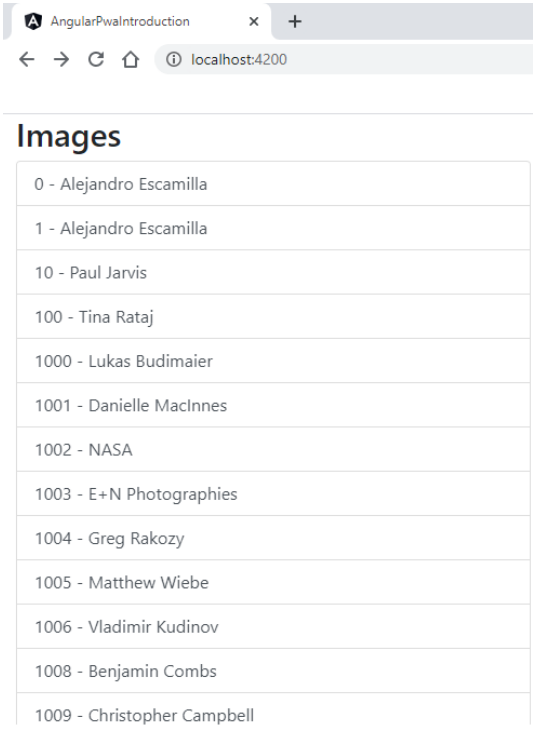
Podemos ver que tenemos la respuesta por consola correctamente.

Ahora vamos a mapear la respuesta de esta llamada a una variable pública para mostrar los datos en el listado de imágenes en el fichero **images.component.html**. Primero modifiquemos un poco el componente **.ts**:

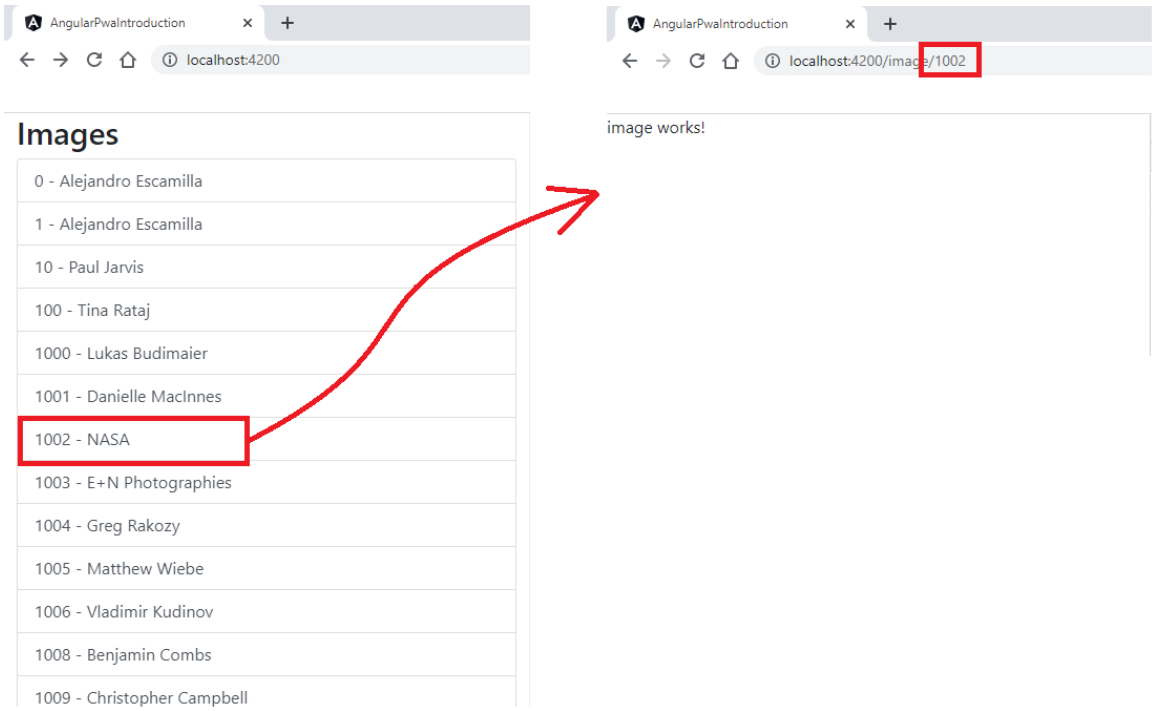


Y ahora modifiquemos la vista para mostrar los nombres de las imágenes y que por cada imagen nos redireccione a la vista detalle pasando por **url** el identificador correcto:

Podemos ver la ejecución:



Y si hacemos clic en cualquier imagen de la lista, podremos ver como iríamos a la vista detalle pasando el identificador correcto por la **url**, por ejemplo:



- **Paso 9:**

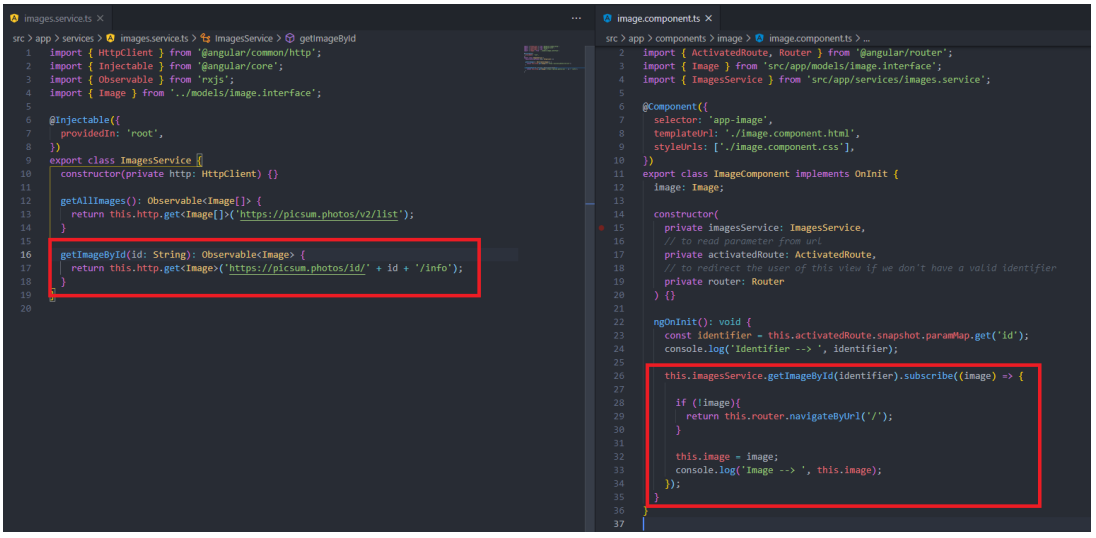
Implementemos la vista detalle **image.component**, primera versión:

En el constructor del **image.component.ts** he añadido un breve comentario para que se entienda el por qué necesitamos los **imports** de **ActivateRoute** y **Router**.

!!! Tenemos que asegurarnos que el nombre de la variable que pasamos por **url** y que recuperamos en el **image.component.ts** mediante el **activateRoute** sea el mismo que definimos en el **app-routing.module.ts**.

Si ejecutamos la aplicación, podremos ver que cada vez que hacemos clic a un elemento de la lista, nos navega a la vista detalle y por consola se muestra el identificador correcto, por lo tanto, somos capaces de capturar el identificador que se pasa por la **url** correctamente.

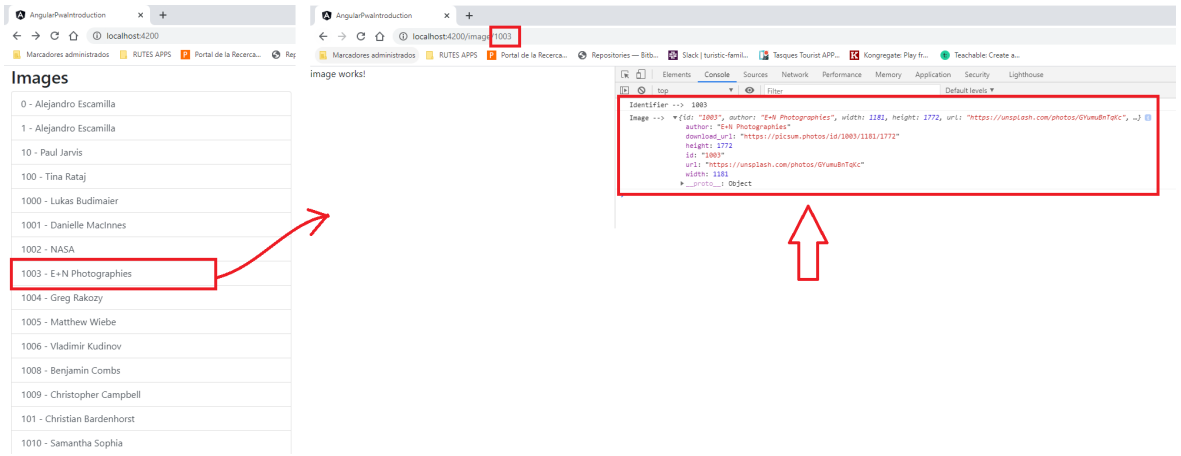
Hagamos lo siguiente:



En el servicio **images.service.ts** implementemos una función **getImageById** en la que pasando un **id** hagamos una llamada a la api y nos devuelva la información concreta de dicha imagen.

Llamemos al controlador **image.component.ts** este servicio que acabamos de implementar enviando por argumento el identificador que nos viene por **url** y que hemos validado que lo capturamos correctamente.

Vemos la ejecución:



Podemos ver que cuando navegamos a la vista detalle se hace la llamada a la api para recuperar la información de la imagen identificada por el **id** que recuperamos por **url** y mandamos a la api.

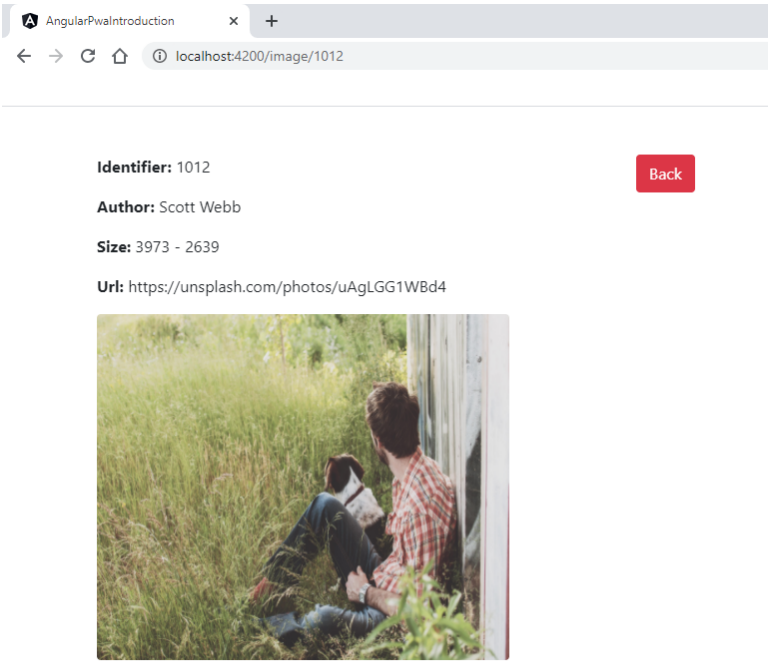
Vamos a maquetar la vista detalle:

image.component.html

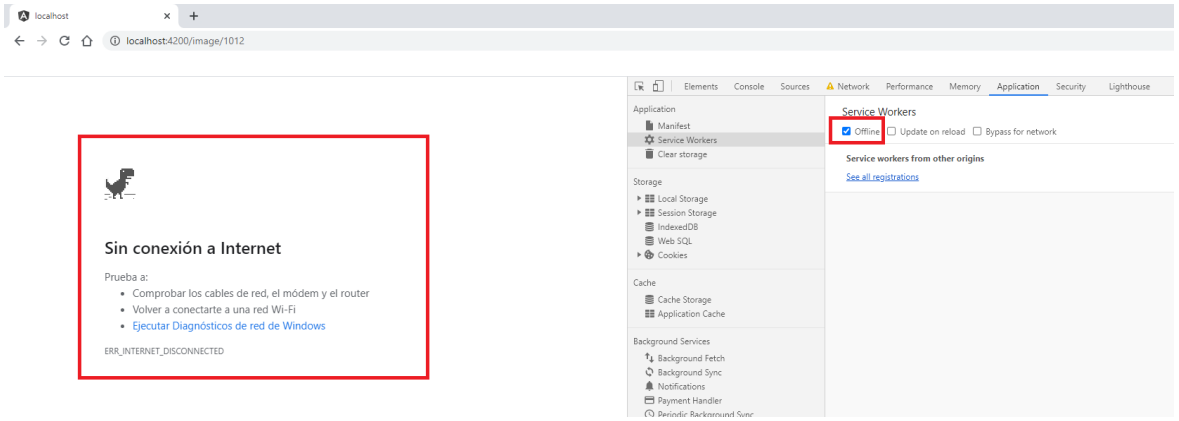
```
src > app > components > image > image.component.html > ...
1
2
3   <div class="container mt-5">
4
5       <div *ngIf="image" class="row animate__animated animate__bounce animate__fadeIn">
6           <div class="col-md-9">
7               <p><strong>Identifier:</strong> {{image.id}}</p>
8               <p><strong>Author:</strong> {{image.author}}</p>
9               <p><strong>Size:</strong> {{image.width}} - {{image.height}}</p>
10              <p><strong>Url:</strong> {{image.url}}</p>
11              <img style="width: 80%; height: 80%;" class="rounded" src="{{image.download_url}}" alt="">
12          </div>
13
14          <div class="col-md-3">
15              <a class="btn btn-danger" routerLink="/">Back</a>
16          </div>
17      </div>
18  </div>
19
```

Vamos a pintar en la vista los campos de la respuesta simplemente para mostrarlos y asegurarnos que funciona bien. Nótese que añadimos un **ngIf=”image”** para asegurarnos que mostramos la información cuando ésta esté cargada, evitando así los típicos errores de **undefined**.

Si ejecutamos la aplicación podremos ver que accediendo a cada detalle de cada imagen veremos algo así:



Llegados a este punto, tenemos una aplicación relativamente sencilla implementada en **Angular**. Nótese que, si asignamos el modo **offline** y refrescamos, no podríamos utilizar nuestra aplicación web, en breve veremos cómo podemos manejar esto:



- **Paso 10:**

Vamos a transformar nuestra aplicación en **PWA**. Como complemento a los pasos que seguiremos a continuación, podemos consultar la documentación oficial en:

https://angular.io/guide/service-worker-intro

El primer paso para transformar nuestra aplicación a **PWA** es ejecutar el siguiente comando:

```
C:\Projectes\angular-pwa-introduction>ng add @angular/pwa
Installing packages for tooling via npm.
Installed packages for tooling via npm.
CREATE ngsw-config.json (624 bytes)
CREATE src/manifest.webmanifest (1372 bytes)
CREATE src/assets/icons/icon-128x128.png (1253 bytes)
CREATE src/assets/icons/icon-144x144.png (1394 bytes)
CREATE src/assets/icons/icon-152x152.png (1427 bytes)
CREATE src/assets/icons/icon-192x192.png (1790 bytes)
CREATE src/assets/icons/icon-384x384.png (3557 bytes)
CREATE src/assets/icons/icon-512x512.png (5008 bytes)
CREATE src/assets/icons/icon-72x72.png (792 bytes)
CREATE src/assets/icons/icon-96x96.png (958 bytes)
UPDATE angular.json (3883 bytes)
UPDATE package.json (1309 bytes)
UPDATE src/app/app.module.ts (865 bytes)
UPDATE src/index.html (657 bytes)
√ Packages installed successfully.
```

Fijémonos en los ficheros que nos ha creado en nuestro proyecto.

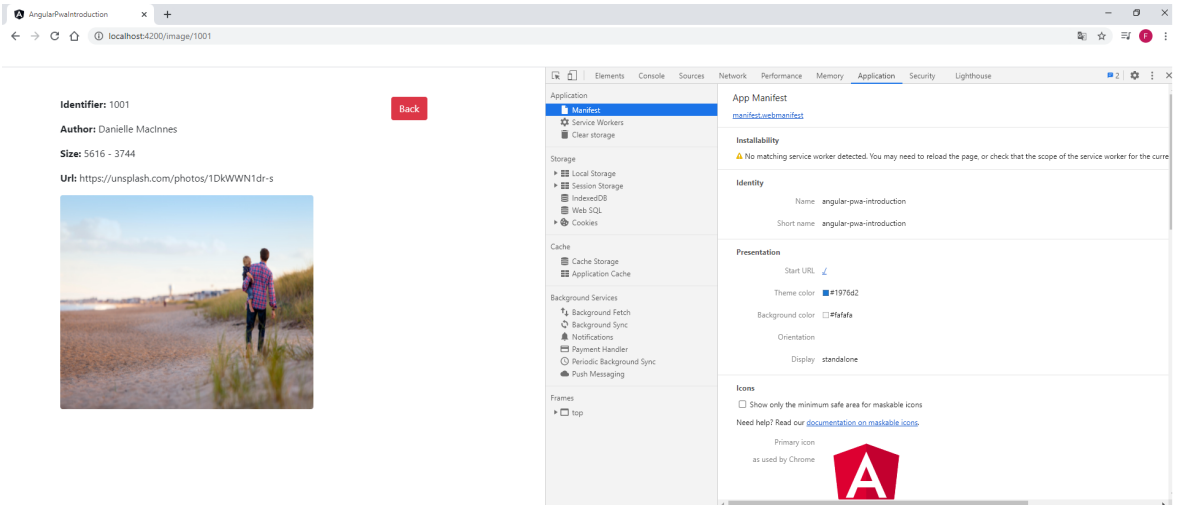
Por una parte nos ha añadido el **ngsw-config.json** el cual es el **json** en el que haremos todas las configuraciones de lo que será nuestro **service worker**, posteriormente lo veremos.

También podemos fijarnos que nos ha añadido el fichero **manifest.webmanifest** el cual contiene el aspecto visual de nuestra aplicación, posteriormente lo podremos observar en el navegador.

También podemos observar que nos ha creado una carpeta **icons** dentro de **assets** con los diferentes iconos que necesitaremos. Utilizaremos los que trae **Angular** por defecto, en una app real, los sustituiremos por los nuestros.

Finalmente, hay que comentar que nos ha actualizado el fichero **index.html** para vincular el **manifest**.

Si ahora hacemos un **ng serve --open** para ver nuestra aplicación en ejecución, podemos ver que si tenemos activo el **manifest** :



Pero en cambio no tenemos registrado ningún **service worker**.

Lo que tenemos que hacer es ejecutar la siguiente instrucción:

ng build

Esto lo que nos hará es crear una carpeta **dist** dentro del proyecto y dentro de esta carpeta lo que tendríamos es todo lo que subiríamos a un servidor real para desplegar la aplicación en un entorno real de producción.

Es en este 'paquete' donde si nos funcionará el **service worker**.

Claro, ¿aquí nos puede surgir una pregunta, como ejecutamos nuestra aplicación en local como si estuviera desplegado en un servidor real? Recordemos que para poder utilizar una **PWA** necesitamos ejecutar nuestra aplicación con un certificado de seguridad **https**.

Aunque si estamos en **localhost** también nos funcionaria.

La manera más sencilla para probar el **frontend** sería instalar el siguiente paquete:

npm install --global http-server

El cual es un servidor ligero que nos permitiría desplegar nuestra aplicación.

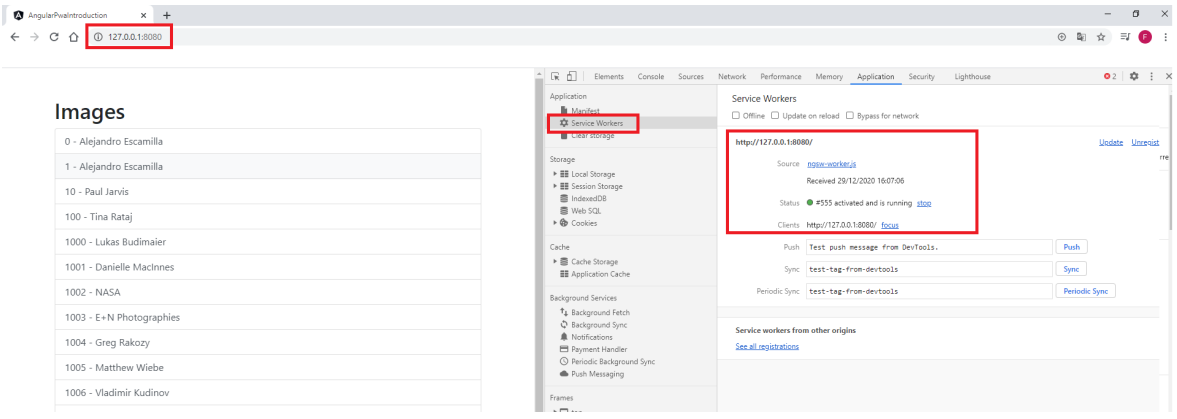
Si vamos a la ruta de donde se nos ha creado nuestro desplegable de **Angular** y ejecutamos **http-server**:

```
C:\Projectes\angular-pwa-introduction\dist\angular-pwa-introduction>http-server
```

Podremos ver:

```
C:\Projectes\angular-pwa-introduction\dist\angular-pwa-introduction>http-server
Starting up http-server, serving ./
Available on:
  http://192.168.1.129:8080
  http://127.0.0.1:8080
Hit CTRL-C to stop the server
```

Esto quiere decir que en **http://127.0.0.1:8080** se está ejecutando nuestra aplicación:



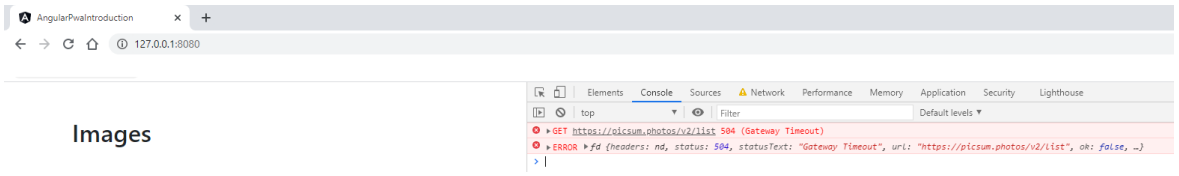
Podemos ver como nuestro **service worker** está registrado y funcionando.

Con este servidor ligero, para hacer pruebas del **frontend** nos valdría.

*No obstante, normalmente necesitaremos montar un **apache** junto con una base de datos local para tener todo el entorno parecido al que podamos tener en el servidor de producción.*

Si ahora ponemos el modo **offline** veremos que la aplicación no funciona correctamente, en breve configuraremos el **service worker** para que podamos trabajar **offline**. Nótese que ahora para poder mostrar la información necesitamos consultar al exterior (internet) los estilos de **Bootstrap**, y las llamadas a los servicios.

Si limpiamos la **caché** y ponemos el modo **offline**:



Veremos que nuestra aplicación sigue funcionando, aunque como podemos ver por la consola, necesita consultar a la api de las imágenes para obtener la información y como no tenemos internet no puede resolver esta información y devuelve error.

Aquí es donde entran las estrategias de la caché que podremos configurar en nuestro **service worker**. Es decir, habrá casos que nos interesará ir a consultar directamente en internet, pero, en otras ocasiones, nos interesará consultar primero en la caché, y si no tenemos los datos que nos interese, nos iríamos a pedirlos a internet.

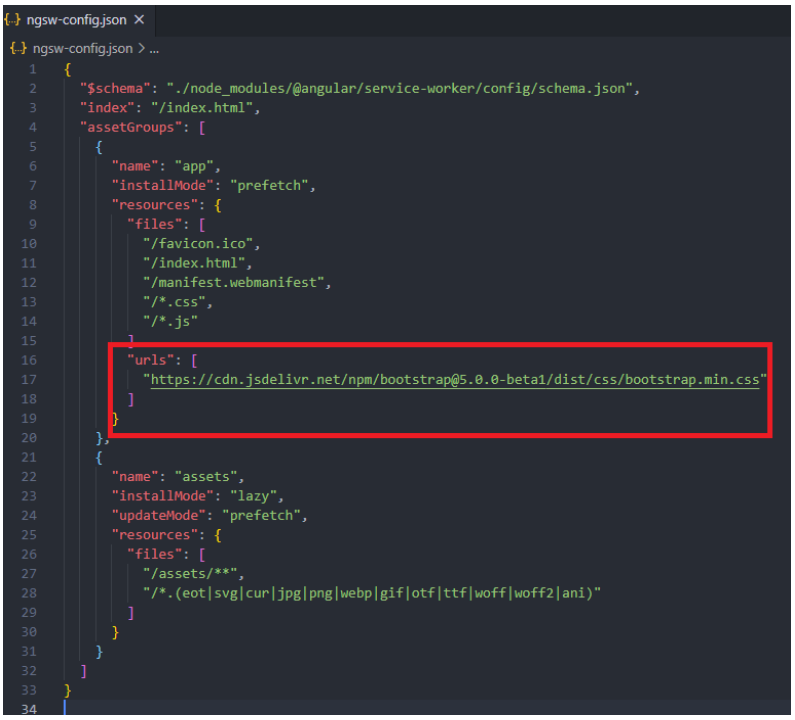
Vamos a hacer las configuraciones necesarias para dejar totalmente funcional nuestra aplicación de ejemplo:

- **Paso 11:**

Configurando nuestro **service worker**:

En el fichero **ngsw-config.json** tenemos:

- **Resources/files**: tenemos lo que sería nuestra **AppShell**, es decir, todos los ficheros indispensables para que funcione nuestra aplicación. Debemos tener en cuenta que todos los recursos de este apartado **files** los busca a partir de la raíz del proyecto.
- ¿Cómo gestionamos la librería de estilos del **Bootstrap** que pusimos dinámicamente en el fichero **index.html**?



Añadimos la propiedad **urls** y aquí podemos definir las **urls** que necesitamos. Con esto lo que les estaremos diciendo es, para esta **url**, cuando sea requerida, la guardas en caché, para que, de este modo, cuando la volvamos a necesitar, la recoja de caché primero en lugar de ir hacia internet.

Con esto, podemos ver que, por una parte, podríamos utilizar la librería sin internet ya que la recogeremos de la caché, y por otro lado, todos los recursos que vengan de la caché son servidos más rápido que si vienen de internet con lo que poco a poco, tendremos un mejor rendimiento global de la aplicación.

Nótese que cuando tengamos las configuraciones hechas en nuestro fichero **ngsw-config.json** tendremos que volver a generar el desplegable haciendo **ng build --prod** para ver los cambios aplicados.

Por lo tanto, iríamos a la raíz del proyecto, ejecutaremos la instrucción anterior, luego iríamos a la carpeta **dist** y volveríamos a levantar el servidor para ver los cambios.

```
C:\Projectes\angular-pwa-introduction\dist\angular-pwa-introduction>cd..

C:\Projectes\angular-pwa-introduction\dist>cd..

C:\Projectes\angular-pwa-introduction>ng build --prod

chunk {} runtime.acf0dec4155e77772545.js (runtime) 1.45 kB [entry] [rendered]
chunk {1} main.9d5796db7fcb6d278074.js (main) 224 kB [initial] [rendered]
chunk {2} polyfills.35a5ca1855eb057f016a.js (polyfills) 36 kB [initial] [rendered]
chunk {3} styles.3ff695c00d717f2d2a11.css (styles) 0 bytes [initial] [rendered]
Date: 2020-12-29T15:40:05.292Z - Hash: 11d18793df2fd06aa999 - Time: 9960ms

C:\Projectes\angular-pwa-introduction>cd dist

C:\Projectes\angular-pwa-introduction\dist>cd angular-pwa-introduction

C:\Projectes\angular-pwa-introduction\dist\angular-pwa-introduction>http-server
Starting up http-server, serving ./
Available on:
  http://192.168.1.129:8080
  http://127.0.0.1:8080
Hit CTRL-C to stop the server
```

Si vamos a la url **http://127.0.0.1:8080/** podremos ver cómo podemos ejecutar nuestra aplicación (marcad **update on reload** para que se actualice el **service-worker**) y si luego ponemos el modo **offline** podremos ver como igualmente tenemos los estilos de **bootstrap** ya que los consulta des de la caché.

La caché la podemos consultar en la misma pestaña **Application** en el apartado **Cache**.

Vamos ahora a manejar las llamadas a la api para poder trabajar sin conexión:

Recomendamos estudiar la documentación oficial para hacerse una idea de todas las posibilidades que tenemos:

https://angular.io/guide/service-worker-config

En este bloque **dataGroups** gestionaremos las estrategias de las llamadas a las apis:

Le pondremos por ejemplo el nombre **images-api**.

En **urls** de momento le ponemos la **url** de la llamada que nos devuelve el listado de imágenes.

Y luego en **cacheConfig** le decimos:

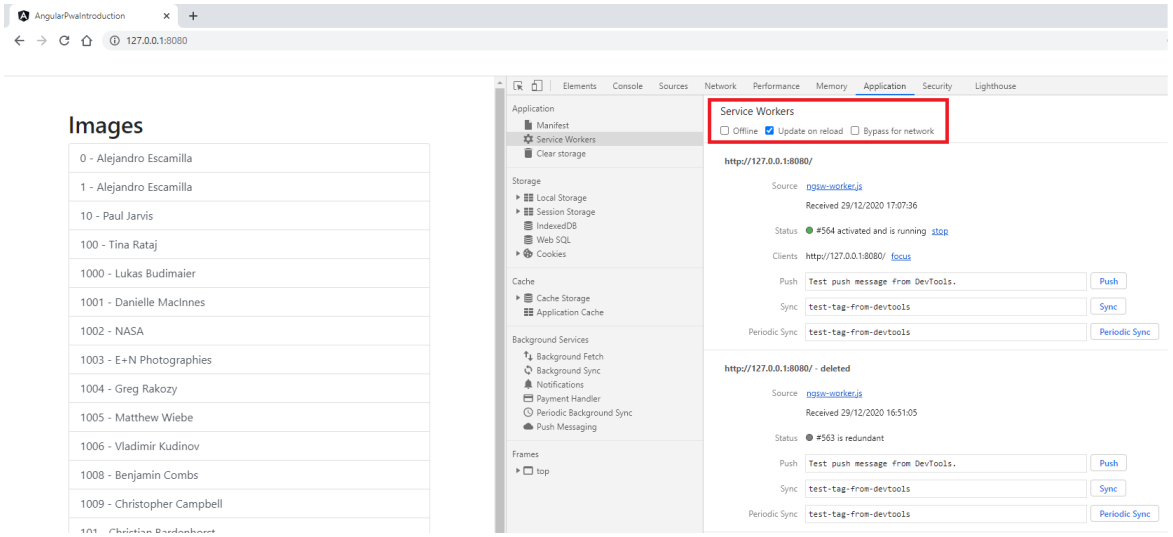
- **maxSize** es el número máximo de respuestas que se guardaran en caché, por ejemplo, le ponemos 10.
- **maxAge** es cuánto tiempo guardaremos en caché dicha información, por ejemplo, le ponemos una hora.
- **timeout** es cuánto tiempo esperas a tener respuesta, si no tenemos respuesta de internet nos vamos a buscar la información en la caché, por ejemplo, le ponemos un segundo.
- **strategy** es la estrategia de caché que queramos utilizar, por ejemplo, le ponemos **freshness** con lo que le estamos diciendo, primero internet, luego caché, por el hecho de que en este caso me interesa tener los últimos datos actualizados, y si por lo que sea no tuviera internet, al menos, podría seguir utilizando la aplicación con los datos de la caché, aunque estos no fueran los últimos.

Este es un simple caso de ejemplo, consulta la documentación oficial para ver las diferentes posibilidades de estrategias y configuraciones para adaptarlas correctamente a vuestros proyectos.

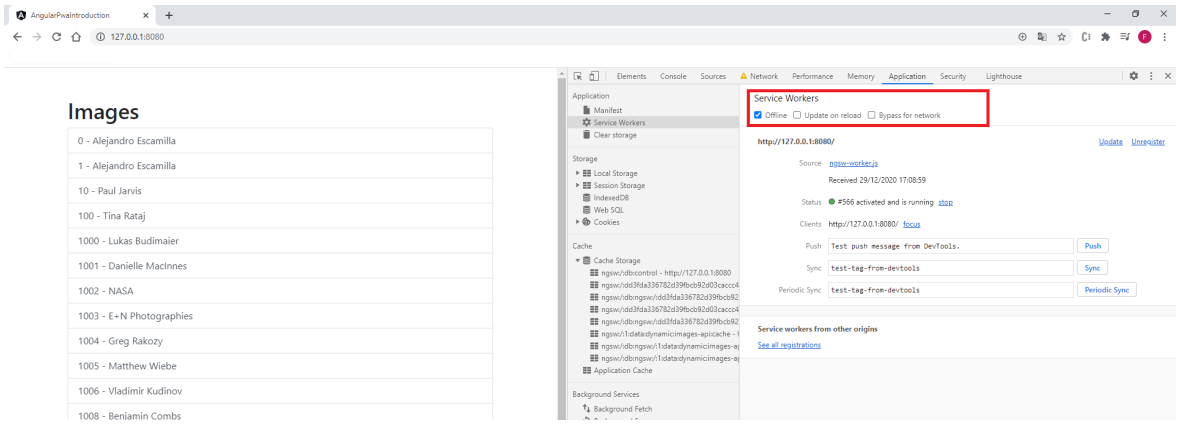
Vamos a probar estos cambios.

Recordemos, hacemos el **build**, volvemos a levantar la aplicación con el servidor **http-server**, sin estar marcado **offline** y estando marcado **update on reload** actualizamos el navegador, y luego marcamos el modo **offline** sin estar marcado **update on reaload.**

Es decir, primero;

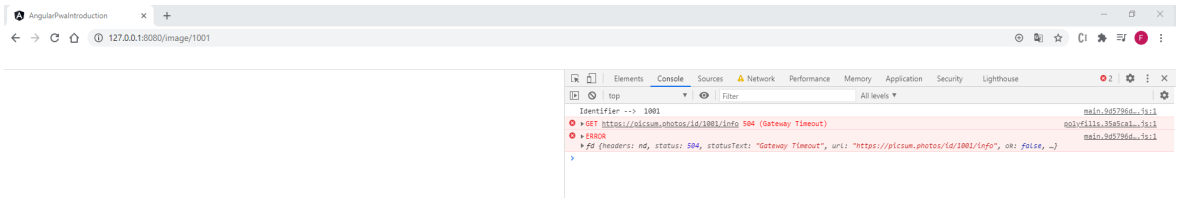


Luego:



En este segundo caso, al refrescar el navegador, podremos ver que igualmente nos carga el listado de imágenes, y esto es debido a que, al no tener internet, lo consulta de la caché. No tendríamos los últimos datos en una app real, pero al menos, tendríamos la app funcional.

¿Vale, pero qué pasa si vamos a una vista detalle estando offline?



Pues que nos fallará, ya que esta **url** no la hemos gestionado.

Aquí lo que podemos hacer es modificar la **url** de esta manera:

```
"dataGroups": [
  {
    "name": "images-api",
    "urls": [
      "https://picsum.photos/**"
    ],
    "cacheConfig": {
      "maxSize": 10,
      "maxAge": "1h",
      "timeout": "1s",
      "strategy": "freshness"
    }
  }
]
```

De esta manera le decimos que trate con la estrategia que nosotros le definimos todo lo que proceda de:

https://picsum.photos/

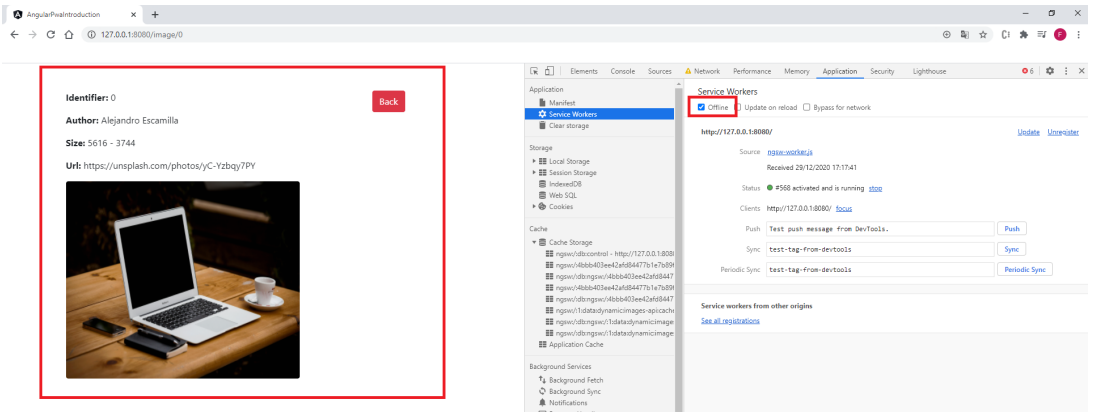
Que en nuestro caso nos vale tanto para la llamada que devuelve el listado de imágenes como la llamada que nos devuelve la información de una imagen concreta y además nos guardaría en caché las propias imágenes.

Compilamos el proyecto y volvamos a ejecutarlo.

Recordar en limpiar caché, ejecutar con **update on reload** marcado**,** luego vamos al detalle por ejemplo de las dos primeras imágenes, y luego marcamos el modo **offline**.

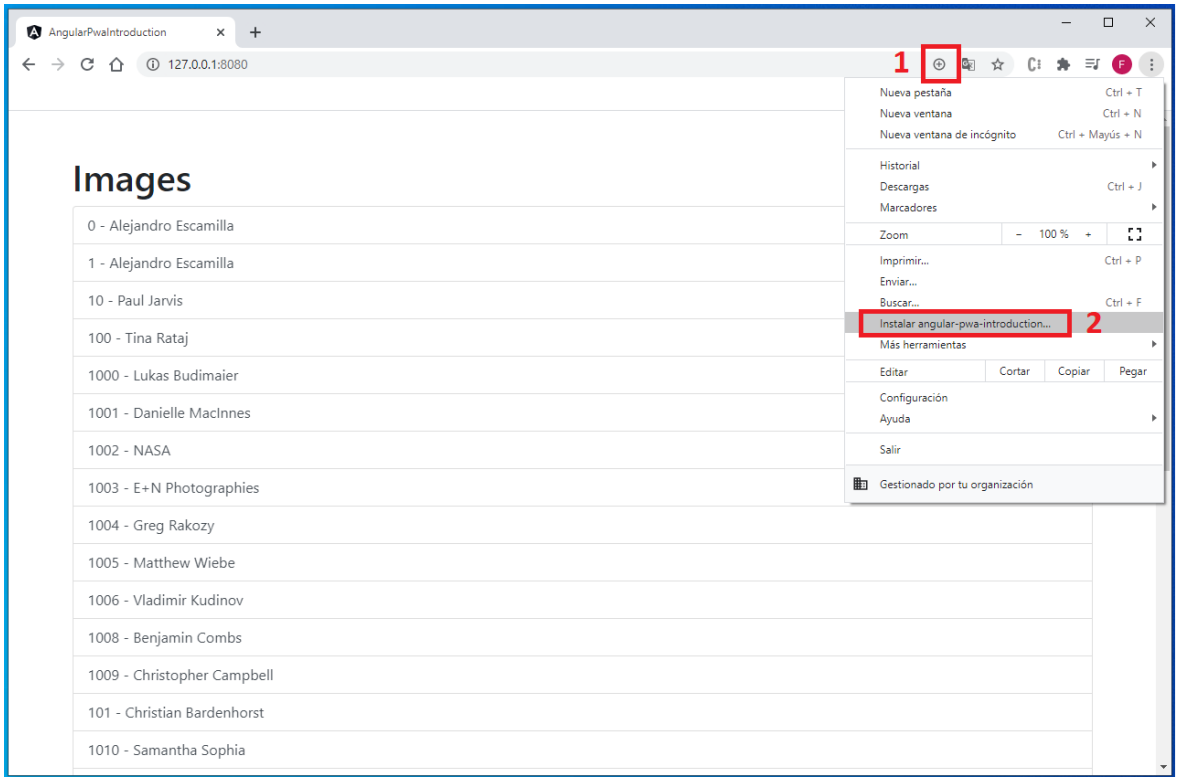
Podremos ver que la página del listado de imágenes funciona correctamente, y que podremos acceder al detalle de las dos primeras imágenes ya que hemos accedido previamente a ellas y aunque ahora no tengamos internet, por nuestra estrategia definida, las recupera de la caché.

Hay que tener en cuenta que el resto de las imágenes no podríamos acceder a su detalle ya que necesitamos al menos haber accedido una vez con internet para guárdalas en caché.

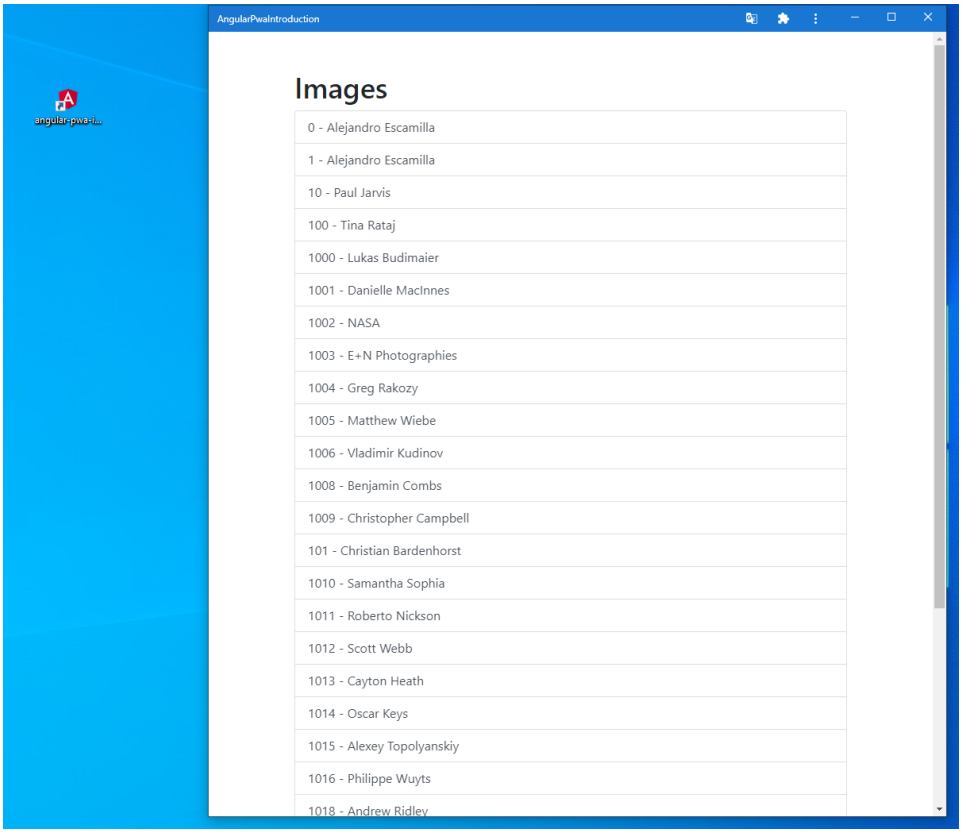


¿Y si queremos instalar nuestra app?

Si estamos en un ordenador, tenemos dos sitios desde donde poder instalar nuestra aplicación:



Si instalamos la app:



Podemos ver el icono en el escritorio, y si lo ejecutamos se nos abrirá nuestra app.

El icono, el nombre del icono, el nombre de la app, el color de la parte superior de la app, … todo esto viene definido del fichero **manifest**.

La gracia de esto es que si accedemos desde un teléfono móvil a una **url** que es una **pwa**, al cabo de unos segundos de acceder nos aparecerá un mensaje que nos dirá si queremos instalar dicha **app**. Si le decimos que sí, se instalará la **app** en un momento y tendremos el icono en el escritorio del móvil como si de otra **app** nativa se tratara.

Esto es muy interesante porque en vez de realizar una **app** nativa para **Android** y una **app** nativa para **iOS**, puedes hacer un único desarrollo web y luego transformarlo a **PWA** con lo que cubres todas las plataformas importantes.

Si que es verdad que en **iOS** hay alguna limitación, como por ejemplo la gestión de las notificaciones **push** que da algunos problemas, pero en principio son cosas que se irán resolviendo.

Una vez estudiada toda esta parte, sería interesante desplegar esta aplicación en algún repositorio publico tipo **github** para que vierais el comportamiento en el móvil.

Server-side rendering

Como hemos visto, una aplicación **Angular** se ejecuta en un navegador donde se renderizan, en tiempo real, las páginas en el **DOM** como respuestas a acciones o eventos del usuario. Así pues, las páginas de una aplicación **Angular** se crean dinámicamente en el cliente cada vez que las pide el usuario, no hay un caché de las páginas de la aplicación porque no son estáticas ni están en el servidor.

Angular Universal es una tecnología incluida en el **core** de **Angular** a partir de su versión 4 que permite correr una aplicación **Angular** en el servidor. Así pues, **Angular Universal** genera páginas estáticas de la aplicación en el servidor gracias a un proceso llamado **server-side rendering (SSR)**. En este sentido, cuando **Angular Universal** es integrada en una aplicación **Angular**, ésta puede generar y servir páginas estáticas en respuesta a las peticiones provenientes desde el navegador. Además, puede pre-generar páginas como HTML antes de que el cliente las solicite.

Las tres principales razones por las que crear una versión **SSR** de tu aplicación son las siguientes:

1. Facilita la indexación en los motores de búsqueda (**SEO**).
2. Mejora el rendimiento en móviles y dispositivos de bajas prestaciones.
3. Muestra la primera página rápidamente.

Para usar **Angular Universal**, se necesita un servidor, conocido con el nombre de **Universal web server**. Éste recibe y responde a las peticiones HTTP de los clientes (normalmente navegadores), sirviendo contenido estático en forma de página HTML (que puede incluir scripts, CSS, imágenes, etc.) usando el **conocido Universal Template Engine**. Normalmente este servidor web es construido utilizando el **framework Express** de **NodeJS**, aunque podría construirse perfectamente con cualquier otro lenguaje (PHP, .NET o JAVA).

Por lo tanto, para conseguir aplicar **Angular Universal** a nuestra aplicación es necesario disponer de dos partes: 1) Servidor (conocido como **Universal web server**); 2) Cliente adaptado a **Angular Universal**.

En la documentación oficial (https://angular.io/guide/universal) se especifica, paso a paso, cómo adaptar una aplicación para poder disponer de la capacidad de **Angular Universal**.

A modo de resumen los pasos que se deben seguir son los siguientes:

1. Instalar dependencias.
2. Preparar la **app**:
    - Añadir soporte para **Angular Universal**.
    - Crear un módulo para ser usado en el **root**.
    - Crear un fichero para exportar el módulo creado previamente.
    - Crear un fichero de configuración para el módulo.
3. Crear un nuevo script de **npm** para construir y empaquetar la aplicación.
4. Configurar el servidor para ejecutar los empaquetados de **Angular Universal**.
5. Empaquetar y ejecutar la aplicación en un servidor.

Schematics

Schematics es una herramienta para el flujo de trabajo enfocada en el desarrollo web creada por el mismo equipo que mantiene **Angular CLI**. Esta herramienta está ideada para realizar transformaciones sobre tu proyecto, permitiendo crear nuevos componentes o actualizar tu código de manera que permita corregir **breaking changes** (i.e. cambios en una parte del código que provocan errores en otras partes que antes funcionaban).

Por lo tanto, esta herramienta es ideal para automatizar tareas repetitivas y mejorar la productividad de los desarrolladores. Los cuatro principales objetivos que trata de cubrir **Schematics** son los siguientes:

1. Fácil de usar y desarrollar (extender). La herramienta debe ser simple para que cualquiera pueda usarlo y extenderla.

2. Extensible y reusable. Al igual que otras herramientas como **gulp** o **grunt**, en **Schematics** se pueden usar otros **Schematics** como entrada y salida.

3. Atomicidad. Todos los cambios son almacenados en memoria (del mismo modo que **gulp**) y sólo son aplicados una vez que se puede confirmar que son válidos.

4. Asincronismo. **Schematics** soporta el asincronismo en sus tareas. De hecho, la entrada de **Schematics** es síncrona, pero la salida puede ser asíncrona.

Si quieres profundizar conociendo la herramienta, te recomendamos la lectura de los siguientes artículos:

1. https://blog.angular.io/schematics-an-introduction-dc1dfbc2a2b2
2. https://medium.com/rocket-fuel/angular-schematics-simple-schematic-76be2aa72850

En esta práctica no se solicitará crear un **Schematic**, sino que se hará uso de uno confeccionado por el equipo de desarrollo.