IONIC 3

WITH

ANGULAR 2

Región de Murcia

dualiza Bankia

Jonatan Lucas

Jonatan.lucas@centic.es

@Jon_Lucas_

Sumario

1.- Introducción

En el siguiente documento vamos a realizar un estudio del framework de desarrollo de aplicaciones híbridas para móviles llamado *Ionic 3*.

Ionic es un framework de código abierto y libre. Te permite construir aplicaciones fácilmente usando tecnología web. La buena noticia es: si tu sabes crear y gestionar sitios web, entonces sabrás crear aplicaciones móviles; pero si no es así, Ionic 3 es un framework fácil de usar y aprender. Ionic ofrece la mejor composición para el desarrollo de aplicaciones webs y nativas.

Puedes pensar que es como el *front-end* que enlaza todas las apariencias e interacciones que tu app necesita para ser construida, dando soporte a los componentes nativos.

Esta guía te conducirá a través de los pasos para crear una aplicación de Ionic en *TypeScript*. Te guiará siempre paso a paso para completar las características de casa caso: *data binding*, estructura del proyecto, navegación, servicios, inyección de dependencias y acceso a datos remotos, entre otros.

1.1.- Requisitos previos

Para trabajar con Ionic 3 necesitas tener instalado los siguientes paquetes (en la web de Ionic encontrarás más información):

- Node.js y NPM
- Ionic y Cordova CLI
- JDK y Android Studio (Android SDK tools)
- Xcode 7 o superior

En lo que al desarrollo concierne, Ionic 3 se programa bajo el lenguaje de Angular 2. Daremos por hecho en este documento que se tienen los conocimientos básicos del desarrollo de aplicaciones web con Angular 2. De todos modos no está de más pasarse por su web y repasar conceptos.

1.2.- Conceptos básicos

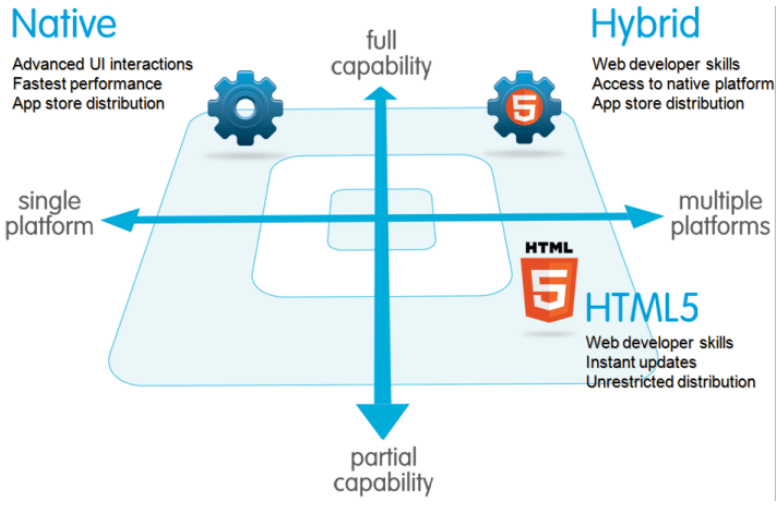
¿Qué son las aplicaciones híbridas?

Las aplicaciones híbridas utilizan una capa de abstracción sobre la plataforma nativa que nos permite encapsular una aplicación HTML5 en una aplicación nativa.

Así, podemos desarrollar un único código fuente para múltiples plataformas reduciendo notablemente los recursos necesarios tanto de desarrollo como de mantenimiento. La tecnología híbrida reduce el *time-to-market* y reduce los costes de desarrollo.

La tecnología híbrida se fundamenta en la siguiente base tecnológica: HTML5, JavaScript y CSS. Como prácticamente todos los sistemas nativos móviles cuentan con un navegador web, basado por norma general en un Webkit, podemos hacer uso de esta base tecnológica de una manera segura y estándar.

Estos sistemas híbridos se encargan, por su parte, de encapsular la aplicación con el webkit de la plataforma nativa. Así pueden ser publicadas sin problemas en los markets de cada plataforma (Google Play Store de Android y App Store de iOS). Además nos facilita la conexión con las API's nativas ofrecidas por cada entorno. De esta manera, accedemos a funcionalidades propias de las aplicaciones nativas como son las notificaciones push, cámara, acceso a las compras de los stores, GPS, sensores, etc.



Este gráfico es un buen resumen de las diferencias entre tecnologías. Sobre un eje cartesiano que representa por un lado las capacidades y por otro la ejecución sobre una o múltiples

plataformas, encontramos que los sistemas híbridos se sitúan en el cuadrante más ventajoso, combinando las virtudes del desarrollo web con el acceso a todas las funcionalidades nativas.

Pero la tecnología híbrida también tiene sus puntos débiles. Respecto a las apps nativas, el rendimiento es menor al ejecutarse sobre una capa webkit y su aspecto poco nativo, lo que puede interferir en la experiencia de usuario.

Para contrarrestar las desventajas de las aplicaciones híbridas, se definen dos técnicas:

- Optimización de rendimiento empleando protocolos ligeros como API Rest o JSON para el acceso a los datos del servidor.
- Mejorar UX de la aplicación aplicando frameworks visuales orientados a dispositivos móviles, como puede ser el caso de este manual con Ionic.

1.3.- Tipos de aplicaciones que se pueden desarrollar

Se puede construir casi cualquier tipo de aplicación, pero conociendo los límites de Ionic 3 podemos desarrollar:

- Apps que se conectan a servidor, como aplicaciones de noticias y aplicaciones de redes sociales.
- Aplicaciones en tiempo real como apps de chat o emisiones en vivo.
- Apps de streaming de música y video.
- Apps de mapas y geolocalización.
- Apps que acceden al hardware del dispositivo.

Como conclusión y siempre desde la necesidad de cada usuario, podríamos decir que si necesitamos una app con claros requerimientos de alto rendimiento como pueden ser juegos, VR, o aplicaciones de realidad aumentada la opción elegida debería ser nativa.

En cambio, si lo que necesitamos es una app de gestión, tratamiento de datos, tienda online, CRM o cualquier otra aplicación para movilizar nuestro backoffice, la tecnología idónea es el desarrollo híbrido, por costes, tiempo de desarrollo y facilidad de mantenimiento.

2.- Diferencia entre aplicaciones híbridas.

Vamos a realizar una tabla comparativa para ver tres de las más usadas:

	Cordova	React Native	NativeScript
Creador	Nitobi; Later	Facebook	Telerik
Interfaz de usuario	HTML	Los componentes de la interfaz de usuario están traducidos a sus contrapartes nativas	Los componentes de la interfaz de usuario están traducidos a sus contrapartes nativas
Puede ser progado en	Navegador, emulador, dispositivo	Emulador, dispositivo	Emulador, dispositivo
Se codifica con	HTML, CSS, JavaScript	Componentes de interfaz de usuario, JavaScript, subconjunto de CSS	Componentes de interfaz de usuario, JavaScript, subconjunto de CSS
Acceso a funcionalidad nativa	A través de plugins	Módulos nativos	Acceso a la API nativa a través de JavaScript
Se despliega en	Androi, iOS, Ubuntu, Windows, OS X, Blackberry 10	Android y iOS. Windows Universal y Samsung Tizen próximamente	Android y iOS
Frameworks y librerías JavaScript	Cualquier librería de front-end o framework (Angular, brasas)	Cualquier librería que no dependa del navegador	Cualquier librería que no dependa del navegador
Patrón de codificación	Cualquier framework front-end se puede utilizar para estructurar el código	El marcado de la interfaz de usuario, JavaScript y CSS están todos agrupados en un solo archivo por defecto	Patrón MVC/MVVM
Cómo es ejecutado JavaScript	WebView	El motor de JavaScriptCore para ejecutar código de aplicaciones en JavaScript y iOS	El motor Webkit JavaScriptCore para ejecutar el código de la aplicación en iOS y el motor de Google V8 en Android

3.- Arquitectura de la app

3.1.- Creando una app

Para crear una app y ejecutarla debemos usar Ionic Cordova CLI, en la terminal ejecutaremos el siguiente comando para crear el proyecto:

ionic start nombreProyecto

Por defecto debemos introducir un tipo de plantilla por lo que por consola nos saldrá una lista para elegir un tipo de plantilla:

- Tabs: Generará un proyecto simple con una interfaz de pestañas (tabs).
- Blank: Generará un proyecto en blanco.
- Sidemenu: Se iniciará un proyecto con un menú lateral con navegación en el área de contenido.
- Super: Generará un proyecto inicial con páginas pre-construidas, proveedores y mejoras prácticas para el desarrollo de Ionic.
- Conference: Se iniciará un proyecto que demuestra una aplicación del mundo real(Alarmas, fecha y hora de una actividad, entre otro).
- Tutorial: Generará un proyecto "tutorial" con ciertos elementos y a preestablecidos que va junto con la documentación en Ionic.
- Aws: Generará un proyecto simple con Amazon Web Service.

3.2.- Ejecución de la aplicación

Desde la raíz de nuestro proyecto podremos lanzar la aplicación y poder visualizarla en nuestro navegador web, para ello ejecutamos el siguiente comando:

ionic serve

Ionic abrirá por defecto el navegador y veremos la aplicación desarrollada con Ionic 3 funcionando. Esta es la forma más rápida y sencilla de probar nuestra aplicación.

Para generar un proyecto en cualquiera de las plataformas Android e iOS debemos usar la consola de Cordova con el siguiente comando:

ionic cordova prepare *plataforma*

Donde *plataforma* podremos elegir entre Android e iOS.

Para ejecutar nuestra aplicación usaremos el siguiente comando:

ionic cordova run *plataforma*

Donde de nuevo *plataforma* será el sistema que queramos ejecutar. Con este comando Ionic compilará y ejecutará la aplicación en el emulador o simulador predeterminado que tengamos asignado o en un dispositivo real si está conectado. Para Android se recomienda usar Genymotion que es más ligero que los propios emuladores de Android Studio. Tan solo tenemos que vincular el emulador a nuestra aplicación.

3.3.- Contenido del proyecto

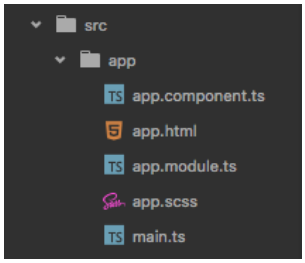
El proyecto, al haber sido creado con la estructura de Angular 2 tiene mucha similitud con ese tipo de estructura.

Al abrir la carpeta del proyecto nos encontramos con una serie de archivos de configuración:

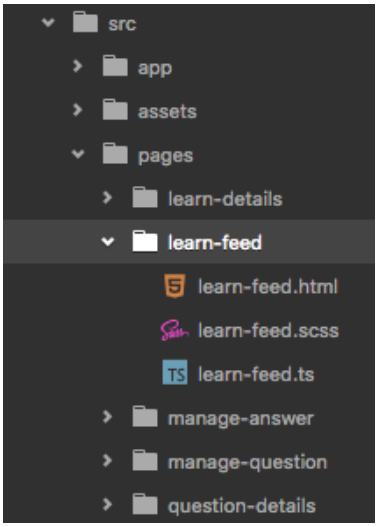
- Package.json: este archivo guarda la información exencial de nuestro proyecto; nombre de la aplicación, dependencias de terceros, versión de la plataforma, etc.
- tsconfig.json: archivo de configuración para la compilación y depuración de nuestra aplicación con el lenguaje de Typescript.
- Config.json: este archivo se usa para compilar la aplicación en la plataforma del dispositivo móvil que queramos.

Dentro de la carpeta *src* es donde encontramos nuestro código. Aquí es donde desarrollaremos la mayor parte de nuestro trabajo. Cuando ejecutamos *ionic serve*, nuestro código dentro de *src* es compilado en el correcto JavaScript que nuestro navegador entiende (normalmente, ES5). Esto significa que podremos trabajar en las últimas versiones de TypeScript, pero compilar bajo la versión de JavaScript que el navegador necesite.

- App.component.ts: define la estructura básica y la navegación inicial de la app.
- App.html: aquí se define el html inicial de nuestra aplicación.
- App.module.ts: este archivo es el punto de entrada de la aplicación. Incluye el módulo principal (NgModule) de nuestra aplicación. Aquí se declaran la mayoría de dependencias (como páginas, componentes, servicios, etc) y enseña al módulo principal a como usarlas.
- App.scss: archivo principal de estilo. Punto de entrada de los diferentes estilos que queramos importar y usar.
- Main.ts: archivo de enlazamiento de la app.



Dentro de src podremos estructurar nuestro proyecto como queramos pero la buena práctica nos recomienda usar un sistema de carpetas según el tipo de archivo que se trate. Por lo que, lo habitual, para trabajar será crear la carpeta *pages* para cada una de las vistas, o páginas, que vayamos a necesitar.

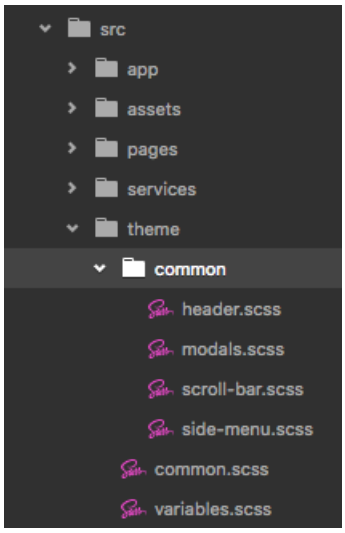


Cada una de estas páginas contiene su propia carpeta. Dentro de ellas encontramos los archivos relacionados con cada página. Estos archivos incluyen el html, sass para los estilos y el javascript para el componente.

- .html: aquí encontraremos las etiquetas de la página, este archivo también es conocido como plantilla.
- .ts: en este archivo encontraremos la funcionalidad e interacción de la página y es donde debería ir la lógica de la vista. Este archivo es conocido como el componente de la página.
- .scss: el estilo especifico de cada página.

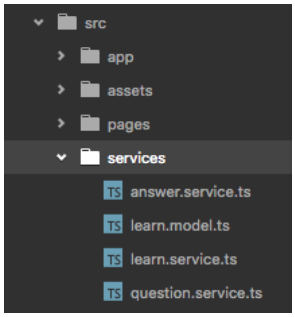
Aquí encontraremos todas las variables, estilos compartidos, etc, que harán que nuestra aplicación sea única.

- Common: dentro de esta carpeta encontraremos (clasificados por componentes/fundionalidad) todos los estilos compartidos.
- Variables.scss: esto es un archivo predefinido por Ionic donde deberíamos incluir todas la variables de sass que usaremos en nuestra aplicación.

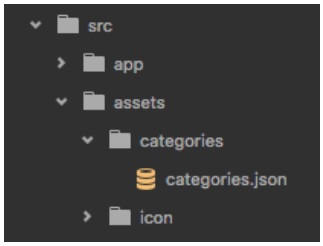


Esta carpeta contiene los servicios de la aplicación. Hay dos tipos de servicios, unos interactúan con una API en un servidor para obtener datos y otros interactúan con información JSON en local.

- .model.ts: acceso a datos en local.
- .service.ts: realiza las peticiones a la API del servidor.



Todas las imágenes y archivos estáticos que vayamos a usar irán en esta carpeta.



3.4.- Módulos

Angular es una aplicación modular. Un módulo es un archivo que contiene un bloque de código dedicado a un solo propósito, exportar un valor que pueda ser usado en otra parte de la aplicación. Por ejemplo:

```
export class AppComponent {}
```

La declaración *export* hace que *AppComponent* pueda ser accesible por otros módulos.

```
import { Component } from '@angular/core';
import { AppComponent } from './app.component';
```

Algunos módulos pueden depender de uno o mas módulos. Los módulos externos deben ser instalados con *npm* y se referencia sin el prefijo de ruta como ‘./’ del componente.

Los módulos externos permiten acceso a características y capacidades de los dispositivos. Todos los dispositivos, plataformas o interfaz de usuario se encuentran en módulos separados. Para poder acceder a las funcionalidades de cada módulo se debe de cargar antes.

En particular, Ionic tiene dos tipos de módulos, ambos usan la implementación de Angular *NgModule*:

- IonicModule: enlaza el *bootstrapping* de una aplicación en Ionic. Nosotros lo importamos en el módulo raíz (app.module.ts). Haciendo esto y pasando un componente raíz (normalmente definido en app.component.ts), IonicModule hará que todos los componentes, directivas, y servicios desde el framework sean importados. De esta manera no necesitaremos importar manualmente todos.
- IonicPageModule: enlaza el *bootstrapping* de las páginas en el orden que se visiten. IonicPages es opcional y es usado para proveer la funionalidad *deep linking* a nuestra aplicación.

3.5.- Componentes

Son el bloque más importante de una aplicación en Angular 2. En ellos se define la UI y la lógica que controla una vista. Cada aplicación tiene un componente raíz que almacena a todos los demás componentes. Cada componente:

- Sabe como interactuar con sus elementos.
- Sabe como renderizarse el mismo.
- Configura las inyecciones de dependencias.
- Tiene bien definido una API de las propiedades de entradas y salidas.

- Tiene bien definido el ciclo de vida.

Vemos el siguiente ejemplo:

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

@Component({
  selector: "my-app",
  template: `
    <StackLayout orientation="vertical">
      <Label [text]="message" (tap)="onTap()"></Label>
    </StackLayout>`
})

export class AppComponent {
  public message: string = "Hello, Angular!";
  public onTap() {
    this.message = "OHAI";
  }
}
```

Cada componente tiene dos partes; la clase que exportamos y la parte de la plantilla:

- Clase: define la lógica del componente, su comportamiento. La parte lógica del componente se comunica con la plantilla a través del *data binding* y los eventos.
- Plantilla: define la vista de un componente.

3.5.1.- Metadatos del componente

La descripción *@Component* contiene metadatos que describen como se crea y presenta el componente. Algunas de sus opciones de configuración:

- Selector: un selector HTML que llama a Angular para crear e insertar una instancia del componente donde se encuentre.

```
<my-app></my-app>
```

- Template: La parte visual del componente.
- TemplateUrl: dirección del archivo .html que contendrá la plantilla del componente.
- Style: directivas CSS para definir el estilo del componente.
- StyleUrls: un array que contiene la ruta los archivos CSS que definen el estilo.
- Providers: un array con la inyección de dependencias de servicios que el componente requiere.

3.5.2.- Ciclo de vida del componente

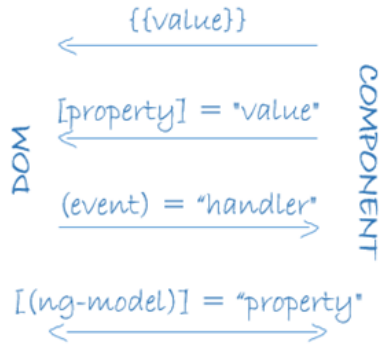
El ciclo de vida del componente está controlado por la parte de Angular. Este ciclo almacena una serie de eventos relacionados. Algunos de los más usados son:

- ngOnInit: llamado después de cargar todos los datos iniciales.
- ngOnChanges: se llama cada vez que se cambia el valor de un punto de dato, es decir algún *input*.
- ngDoCheck: detecta y actua sobre los cambios que Angular puede o no detectar en si mismo. Llamado en cada evento de cambio.
- ngOnDestroy: llamado justo antes de que Angular destruya el componente.

Para conocer la lista completa de eventos ver la página oficial de Angular sobre esta información.

3.5.- Data Binding

Es el mecanismo encargado de conectar las partes de una vista (plantilla) con la lógica del componente. Existen varias maneras de realizar esta conexión:



- *[text]=”message”*: llamado *‘one-way binding’*, la información fluye en una dirección, desde el componente a la vista. Si cambiamos alguna de las propiedades del mensaje en la lógica del componente se reflejará en la vista.
- *(tap)=”onTap()”*: llamado *‘event binding’*. Esta vez conectamos la vista con la lógica del componente a través de una función dada.
- *[(ngModel)]=”message”*: llamado *‘two-way binding’* o doble binding. Cuando el usuario introduce datos en el textfield cambiará las propiedades del mensaje en el componente y viceversa, si se modifica desde el código cambiará en el textfield.

3.7.- Directivas

Las directivas te permiten crear comportamientos en la creación de la vista. Hay tres tipos de directivas:

- Componentes: ya hemos hablado de ellos. A través de código podemos introducir o eliminar alguna etiqueta o elemento de la plantilla.
- Directivas estructurales: alterar la vista añadiendo, eliminando o reemplazando elementos. Los más comúnes son **ngIf* y *ngFor.
- Directivas atributo: cambian el aspecto o el comportamiento de la UI de un elemento. Uno de los más comúnes es la directiva *ngClass*.

3.8.- Dependencia de inyector

Es una herramienta muy potente de Angular y bastante usada en NativeScript. Para saber más sobre ella visitar la web de Angular.

3.9.- Navegación

El módulo Router de Angular es una de sus librerías mas importantes en un aplicación. Sin esta, las aplicaciones serían simples aplicaciones con una sola vista y contesto o no serían capaces de navegar entre vistas.

3.9.1.- Simple Route

Para la mayoría de aplicaciones, a menudo es necesario tener algún tipo de ruta. La configuración más básica se ve un poco así:

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

@NgModule({
  imports: [
  ...
  RouterModule.forRoot([
    { path: '', component: LoginComponent },
    { path: 'detail', component: DetailComponent },
  ])
  ],
})
```

El desglose más simple de lo que tenemos aquí es una búsqueda de ruta / componente. Cuando se carga nuestra aplicación, el enrutador inicia el proceso al leer la URL que el usuario intenta cargar. En el ejemplo, nuestra ruta busca ‘’, que es esencialmente nuestra ruta por defecto. Así que para esto, cargamos el *LoginComponent*. Este patrón de rutas coincidentes con un componente continúa para cada entrada que tenemos en la configuración del enrutador.

3.9.2.- Rediracciones

Los redireccionamientos funcionan de la misma forma que lo hace un objeto de ruta típico, pero solo incluye unas cuantas claves diferentes.

```
[
  { path: '', redirectTo: 'login', pathMatch: 'full' },
  { path: 'login', component: LoginComponent },
  { path: 'detail', component: DetailComponent }
];
```

En nuestra redirección, buscamos la ruta del índice de nuestra aplicación. Luego, si cargamos eso, redirigimos a la ruta ‘login’. El último campo, *pathMatch*, se requiere para indicar al enrutador como debe buscar la ruta.

Como en el ejemplo anterior hemos puesto ‘full’, estamos diciendo al enrutador que debemos comparar la ruta completa, incluso si termina siendo algo así:

```
{ path: '/route1/route2/route3', redirectTo: 'login', pathMatch: 'full' },
{ path: 'login', component: LoginComponent },
```

Si cargamos la ruta de ‘/route1/route2/route3/’ vamos a redirigir. Pero si cargamos otra ruta diferente, por ejemplo ‘/route1/route2/route4/’, no redirigiremos, ya que las rutas no coinciden completamente.

Alternativamente, si utilizamos:

```
{ path: '/route1/route2', redirectTo: 'login', pathMatch: 'prefix' },
{ path: 'login', component: LoginComponent },
```

Luego, al cargar cualquiera de las rutas que hemos propuesto seremos redirigidos para ambas. Esto se debe a *‘pathMatch: ‘prefix’’* que solo coincidirá con parte de la ruta.

3.9.3.- Navegando por diferentes rutas

Hablar de rutas es bueno, pero ¿cómo se navega realmente a dichas rutas? Para ello, podemos utilizar el atributo *routerLink*. Continuando con el ejemplo tendremos el siguiente código HTLM:

```
<ion-header>
  <ion-toolbar>
    <ion-title>Login</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content padding>
  <ion-button [routerLink]="['/detail']">Go to detail</ion-button>
</ion-content>
```

La parte importante aquí es la directiva *ion-button* y *routerLink*. RouterLink funciona con una idea similar a *href*, pero en lugar de construir la URL como una cadena, puede construirse como una matriz, que puede proporcionar rutas más complicadas.

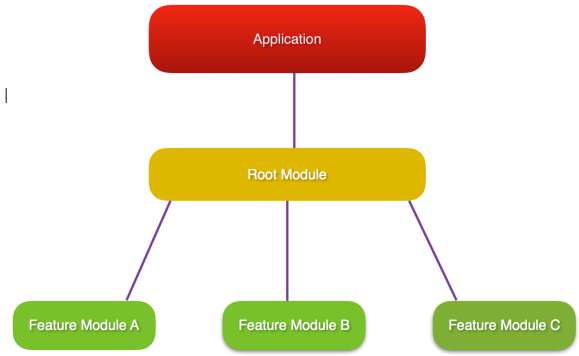
También podemos navegar a través de código en nuestra aplicación utilizando la API del enrutador.

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

@Component({
  ...
})
export class LoginComponent {

  constructor(private router: Router){}

  navigate(){
    this.router.navigate(['/detail'])
  }
}
```

3.9.4.- *Lazy loading* rutas

Cuando desarrollamos aplicaciones debemos estar pendientes de la presentación y optimizar la aplicación. El desarrollo de aplicaciones con Angular puede dar como resultados archivos grandes que afecten al tiempo de inicio de nuestra aplicación. El router de Angular desarrolla una característica para retrasar esos archivos en el tiempo de ejecución inicial si no son necesarios para el arranque de la aplicación.

Con *lazy loading* podemos separar nuestra aplicación en módulos y cargarlos según los necesitemos.



Para mas información acerca de *lazy loading* para Angular visitar este blog.

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

@NgModule({
  imports: [
  ...
  RouterModule.forRoot([
    { path: '', redirectTo: 'login', pathMatch: 'full' },
    { path: 'login', loadChildren: './login/login.module#LoginModule' },
    { path: 'detail', loadChildren: './detail/detail.module#DetailModule' }
  ])
  ],
})
```

Si bien es similar, la *loadChildren* propiedad es una forma de hacer referencia a un módulo por cadena en lugar de un componente directamente. Para hacer esto, sin embargo, necesitamos crear un módulo para cada uno de los componentes.

```
. . .
import { RouterModule } from '@angular/router';
import { LoginComponent } from './login.component';

@NgModule({
  imports: [
  ...
  RouterModule.forChild([
    { path: '', component: LoginComponent },
  ])
  ],
} )
```

5.- Bibliografía

- Documentación oficial de Ionic 3. Recuperado de https://ionicframework.com/docs