2.- Arquitectura de Angular

Angular es un framework escrita en Typescript que junto a ficheros HTML crean una aplicación.

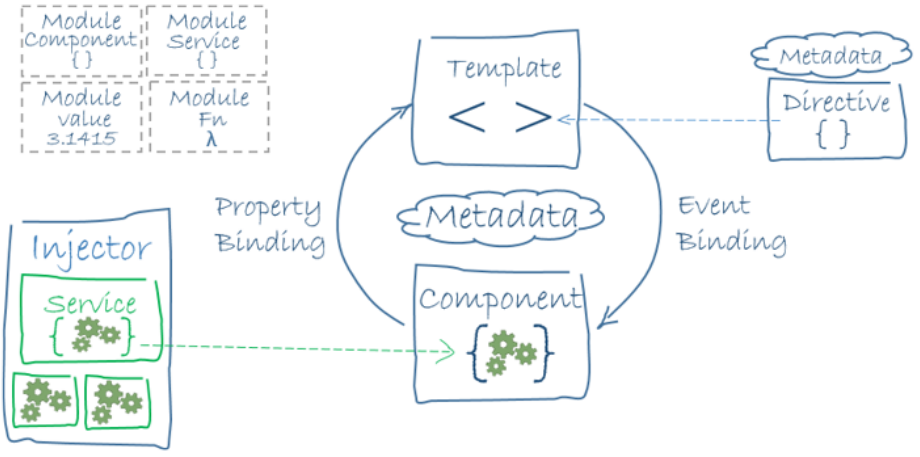
La construcción básica en Angular son los NgModules, los cuales proveen un contexto de compilación para los Components. NgModules recopilan el código en conjuntos funcionales; una app en Angular está definida como un conjunto de NgModules y siempre hay un módulo raíz que capacita el bootstrapping, o enlazamiento, y que contiene más módulos.

- Los Componentes definen vistas, las cuales son un conjunto de elementos visuales que Angular puede elegir y modificar de acuerdo a su lógica de programación y datos.
- Los Componentes usan servicios, services, los cuales proveen funcionalidades específicas no relacionadas directamente con la vista. Los servicios son inyectados en los componentes como dependencias, lo que hace que el código sea más eficiente.

Los componentes y servicios son simplemente clases, que proveen metadata que Angular les dice como usar.

- El metadata para una clase component se asocia con una plantilla, template, que es definida en una vista. Una plantilla combina HTML con directivas de Angular y la funcionalidad de binding que permite a Angular modificar el HTML antes de renderizarlo para mostrar la vista.
- El metadata para una clase service provee la información que Angular necesita para capacitar a los componentes de información a través de las inyecciones de dependencia, dependency injection.

Una app en Angular está compuesta por muchos componentes y vistas, por lo que para poder navegar entre ellas Angular provee un módulo llamado Router que ayuda a la navegación entre las mismas.

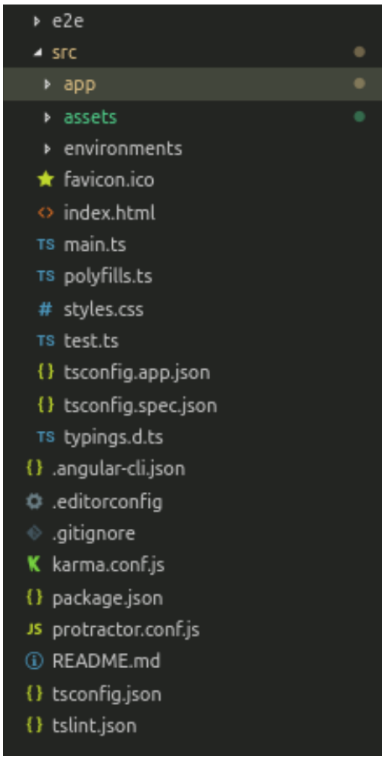


- Juntos, un componente y una plantilla definen una vista en Angular.
    - Un ayudante en la clase del componente que añade el metadata, para asociar la plantilla.

    - Directivas y binding en las plantillas de los componentes modifican las vistas basándose en la lógica de programación e información.
- El inyector de dependencias provee los services a los componentes, como por ejemplo el router service que permite navegar entre vistas.

2.1.- Estructura de un proyecto en Angular

2.1.1.- Estructura inicial



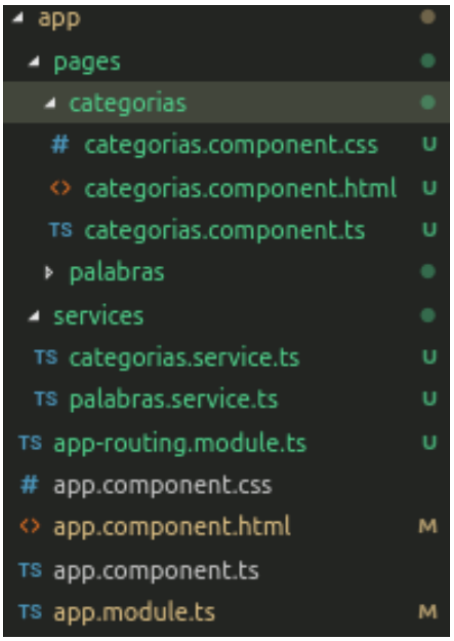
De los siguientes archivos, los más importantes son:

- Index.html: como toda aplicación web, se necesita este archivo para acceder a la aplicación. Pero en un proyecto en Angular no es necesario tocar nada de él salvo para algo de CSS como por ejemplo colocar un footer con imágenes.
- Main.ts: Este archivo rara vez se toca para una aplicación. Su función consiste en lanzar el módulo raíz, el app.module.ts, para poder ejecutar la aplicación.
- Styles.css: archivo principal de CSS de nuestra aplicación.
- Favicon.ico: la imágen de nuestra aplicación en el navegador web.

Y de carpetas las siguientes:

- App: contiene toda la aplicación.
- Assets: esta carpeta es para los archivos estáticos.

2.1.2.- Dentro de la carpeta app



- App.module.ts: módulo raíz de nuestra aplicación. Aquí se cargan los componentes, servicios y módulos de angular que vayamos a usar.
- App.component: es el componente principal, el primer componente que se carga y necesitamos para trabajar.
- App-routing.module.ts: archivo para crear las rutas internas de Angular para trabajar con la aplicación.
- Pages: carpeta creada para almacenar cada una de las vistas creadas para nuestro proyecto de Angular.
- Services: carpeta creada para almacenar nuestros archivos services.

2.2.- Módulos

Las aplicaciones en Angular son modulares y Angular tiene su propio sistema de modulación llamado NgModules. NgModules son contenedores para una cohesión de código dedicado a una app, un flujo de trabajo, o funciones relacionadas de capacidades. Pueden contener componentes, servicios y otros archivos de código necesario para nuestra app.

Cada app de Angular contiene al menos un NgModule, el módulo raíz, normalmente llamado AppModule en el archivo también llamado app.module.ts. La app se lanzará enlazando con este módulo raíz.

Si bien una pequeña aplicación puede tener solo un NgModule, la mayoría de apps tienen muchos más módulos. El módulo raíz se llama así porque puede contener muchos más NgModules secundarios.

2.2.1.- NgModule metadata

Un NgModule se define como una clase enlazada con @NgModule(). El @NgModule() es una función que toma un objeto de metadatos para describir las propiedades de un módulo. Las propiedades más importantes son:

- declarations: Los componentes, directivas y pipes que pertenecen a ese NgModule.
- exports: El subconjunto de declaraciones que deben ser visibles y utilizables en las plantillas de los componentes de otros NgModules.

- imports: Otros módulos cuyas clases exportadas son necesarias para las plantillas de los componentes de este NgModule.
- providers: Colección de los services que se necesitan en este NgModule.
- bootstrap: Solo el módulo raíz deberá almacenar esta propiedad.

Ejemplo de definición de root NgModule :

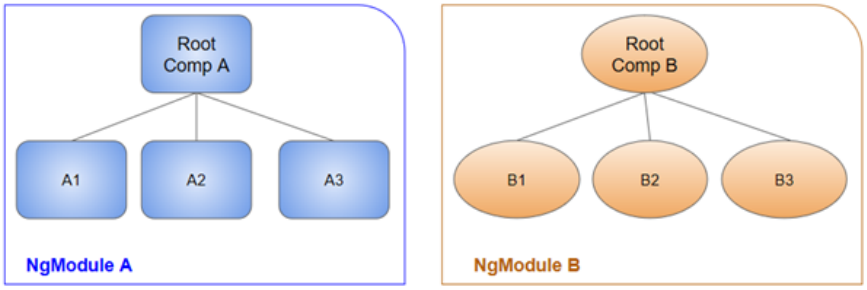
```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})

export class AppModule { }
```

Ejemplo de un subconjunto de NgModule:

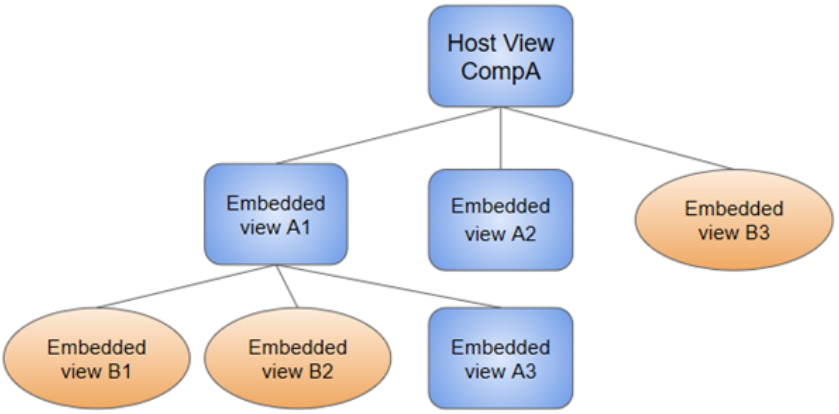
```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ NameService ],
  declarations: [ NameComponent ],
  exports:      [ NameComponent ]
})

export class NameModule { }
```

2.2.2.- NgModule y componentes

NgModule proporciona un contexto de compilación para sus componentes. Un NgModule raíz siempre tiene un componente raíz que se crea durante el arranque, pero cualquier NgModule puede incluir cualquier número de componentes adicionales, que se pueden cargar a través del enrutador o crear a través de la plantilla. Los componentes que pertenecen a un NgModule comparten un contexto de compilación.



Un componente y su plantilla juntos definen una vista de Angular. Un componente puede contener una vista principal, con la cual se permite definir áreas más complejas de una vista que pueden ser creadas, modificadas y destruidas como una unidad. Una vista jerarquizada puede combinar varias vistas definidas en componentes que pertenecen a diferentes NgModules.



Cuando se crea un componente, se asocia directamente con una vista única. Esta vista puede ser raíz de una serie de vistas de otros componentes. Esos componentes pueden estar en el mismo NgModule, o pueden ser importadas desde otros módulos.

2.3.- Componentes

Cada componente controla una vista. Definen la lógica de la aplicación dentro de una clase. Esta clase interactúa con la vista a través de una API de propiedades y métodos.

```
export class HeroListComponent implements OnInit {
  //Variables
  heroes: Hero[];
  selectedHero: Hero;

  constructor(private service: HeroService) { }

  ngOnInit() {
    this.heroes = this.service.getHeroes();
  }

  selectHero(hero: Hero) {
     this.selectedHero = hero;
  }
}
```

Angular crea, actualiza y destruye componentes a medida que el usuario se mueve a través de la aplicación.

Para crear un componente Angular CLI posee un comando que generará una carpeta con los archivos básicos necesarios para una vista en Angular y lo introducirá en el @NgModule principal.

```
ng generate component name_component
```

Tras este comando se generará una carpeta con el nombre para el componente los siguientes archivos:

- Un fichero HTML para la plantilla.
- Un fichero TS para el componente con el código básico de un componente.
- Un fichero CSS para el diseño.

2.3.1.- Metadatos de un componente

```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ],
  styleUrls: [‘./hero-common.css’ ]
})

export class HeroListComponent implements OnInit {
/* . . . */
}
```

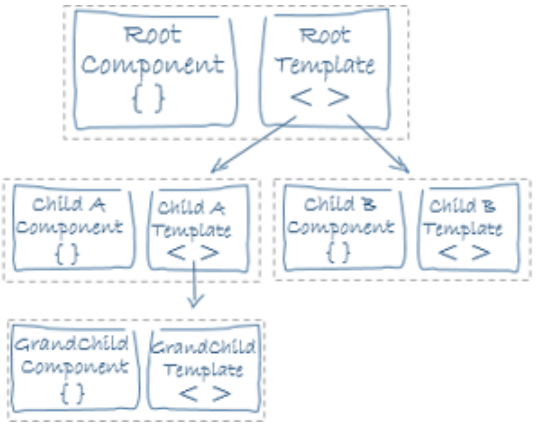
En el código de ejemplo se puede ver como un componente no es más que una clase, sin ningún tipo de anotación o sintaxis especial. Pero no se puede trabajar con él como un componente hasta que no creamos el decorador @Component()

Los metadatos de un componente le indican a Angular dónde obtener los principales bloques de construcción que se necesitan para crear y presentar el componente y su vista. En particular, asocia una plantilla con el componente. Juntos describen y muestran una vista de Angular. Además los metadatos también pueden configurar como se puede hacer referencia al componente en HTML y qué servicios requiere.

Algunas de las opciones más útiles son:

- selector: Un selector CSS que le dice a Angular que cree e inserte una instancia de este componente donde encuentre la etiqueta correspondiente en la plantilla HTML.
- templateUrl: Carga la plantilla que se usará para este componente. Alternativamente se puede generar el CSS directamente a través de la opción template. Pero se recomienda la opción de un archivo diferente para una mejor estructuración.
- providers: Introducimos los servicios necesarios para el componente.
- sytleUrls: Carga los ficheros CSS en nuestra plantilla.

2.3.2.- Plantillas y vistas



Una plantilla es una forma de HTML que le dice a Angular como representar el componente.

Las vistas suelen estar ordenadas jerárquicamente, lo que le permite modificar o mostrar y ocultar secciones o páginas completas como una unidad.

La plantilla inmediatamente asociada a un componente define la vista principal de ese componente. El componente también puede definir una jerarquía de vistas, que vistas anidadas, alojadas por otros componentes.

Una vista puede incluir vistas de componentes en el mismo NgModule, pero también puede incluir vistas de componentes que están definidos en diferentes NgModules.

2.3.3.- Sintaxis de una plantilla

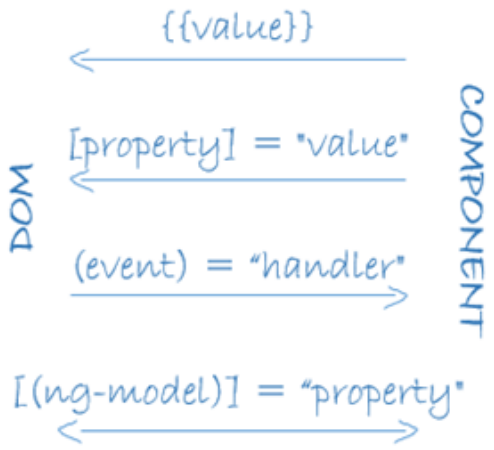
Una plantilla es como un archivo de HTML, a excepción que este contiene sintaxis de Angular, con el cual se podrá alterar el HTML a través de la lógica de la app. La plantilla puede usar data binding para coordinar la app y el DOM, los pipes transforman la información antes de ser mostrada y las directivas aplican la lógica a lo que se muestra.

```
<h2>Hero List</h2>

<p><i>Pick a hero from the list</i></p>
<ul>
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
    {{hero.name}}
  </li>
</ul>

<app-hero-detail *ngIf="selectedHero" [hero]="selectedHero"></app-hero-detail>
```

Esta plantilla usa elementos comunes de HTML como <h2> y <p> combinados con elementos de Angular como *ngFor, {{hero.name}}, (click), [hero] y <app-hero-detail>. La sintaxis de Angular dirá como mostrar los elementos de HTML, usando la lógica de la app y los datos.

- *ngFor dice a Angular como interactuar con una lista de elementos.
- {{hero.name}}, (click) y [hero] enlaza los datos a mostrar con el DOM.
- <app-hero-detail> es un elemento que representa una plantilla diferente con su componente correspondiente.

2.3.4.- Data binding



Sin un framework, seríamos responsables de enlazar la información en el HTML y los componentes y viceversa, lo que supondría una tarea pesada.

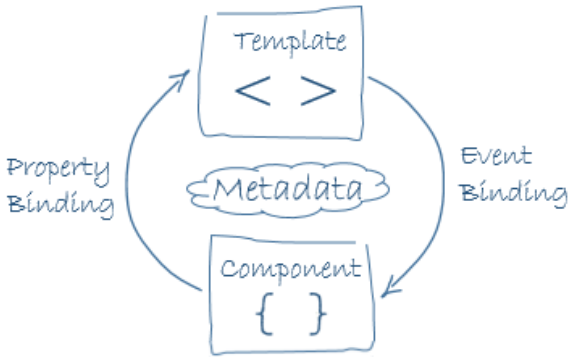
Angular soporta two-way data binding, un mecanismo que coordina las partes de una plantilla con la lógica de un componente y viceversa para trabajar con los datos de una forma más sencilla y ágil.

```
<li>{{hero.name}}</li>

**getVal() devuelve el valor de 4**
<p>The sum of 1 + 1 is not {{1 + 1 + getVal()}}.</p>

<app-hero-detail [hero]="selectedHero"></app-hero-detail>

<img [src]="itemImageUrl2">

<button [class.special]="isSpecial" [style.color]="isSpecial ? 'red': 'green'" >Special</button>

<li (click)="selectHero(hero)"></li>
```

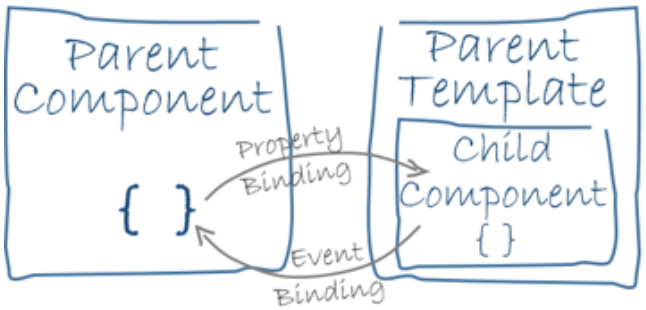
- {{hero.name}} muestra la información del componente relacionada con el objeto hero.name.
- [hero] enlaza el valor de la propiedad selectedHero del componente HeroListComponent al objeto hero del componente HeroDetailComponent.
- (click) enlace de evento. Llama al método del componente cuando el usuario hace clic en el nombre de un elemento.

```
<input [(ngModel)]="hero.name">
```

- Two-way data binding, enlaza los datos asociados a ese elemento de tal manera que si se actualizan en el componente también lo hacen en la plantillas.



El enlace de datos juega un papel importante en la comunicación entre una plantilla y su componente, y también es importante para la comunicación entre los componentes principal y secundario.



Bindeo de propiedad o interpolación

Normalmente podemos elegir entre ambas. Por ejemplo:

```
<p><img src="{{heroImageUrl}}"> is the <i>interpolated</i> image.</p>
<p><img [src]="heroImageUrl"> is the <i>property bound</i> image.</p>

<p><span>"{{title}}" is the <i>interpolated</i> title.</span></p>
<p>"<span [innerHTML]="title"></span>" is the <i>property bound</i> title.</p>
```

Interpolar {{}} es una alternativa al bindeo de propiedad []. No hay una regla que defina cuándo usar uno u otro con valores string. Pero sí que debemos usar el bindeo de propiedad para enlazar con valores no-string.

2.3.5.- Pipes

Las “tuberías” de Angular permiten transformar el valor de visualización de un elemento en la plantilla de HTML.

Angular define varias pipes, las mas conocidas son las de fecha y de moneda. Para una lista completa visitar la lista de API de pipes. También se pueden definir nuevas tuberías.

Para especificar dicha transformación se usará el operador de pipe:

```
{{ interpolated_value | pipe_name }}
```

Ejemplos de uso de pipe:

```
<!-- Default format: output 'Jun 15, 2015'-->
 <p>Today is {{today | date}}</p>

<!-- fullDate format: output 'Monday, June 15, 2015'-->
<p>The date is {{today | date:'fullDate'}}</p>

 <!-- shortTime format: output '9:43 AM'-->
 <p>The time is {{today | date:'shortTime'}}</p>
```

2.3.6.- Directivas

Las plantillas de Angular son dinámicas. Cuando Angular las procesa, transforma el DOM de acuerdo con las instrucciones dadas por las directivas. Una directiva es una clase con un decorador @Directive().

Un componente es técnicamente una directiva. Además existen otros dos tipos de directivas: estructurales y de atributo.

Al igual que para los componentes, los metadatos de una directiva asocian la clase con un selector que se usa para insertarlo en HTML. En las plantillas las directivas suelen aparecer dentro de una etiqueta como atributo de un elemento.

Directivas estructurales

Las directivas estructurales modifican el diseño al agregar, eliminar y reemplazar elementos en el DOM. La plantilla del ejemplo utiliza dos directivas integradas para agregar la lógica de la aplicación a cómo se representa la vista.

```
<li *ngFor="let hero of heroes"></li>

<app-hero-detail *ngIf="selectedHero"></app-hero-detail>
```

- *ngFor es un iterativo; le dice a Angular que muestre e un <li> por cada héroe de la lista.
- *ngIf es un condicional: incluye el componente HeroDetail solo si exite un héroe seleccionado.

Directivas de atributos

Las directivas de atributos alteran la apariencia o el comportamiento de un elemento existente. En las plantillas, se ven como atributo HTML normales.

La directiva ngModel que implementa enlace de datos bidireccional, es un claro ejemplo. NgModel modifica el comportamiento de un elemento existente configurando su propiedad de valor de visualización y respondiendo a los eventos que generen algún cambio.

```
<input [(ngModel)]="hero.name">
```

Otras directivas de atributo pueden ser:

- ngSwitch: altera la estructura del diseño.
- ngStyle o ngClass: modifican los elementos y componentes del DOM.

2.4.- Servicios e inyector de dependencias

Un servicio, service, es una clase amplia que abarca cualquier valor, función o característica que necesita una app. Un servicio está bien definido y con un propósito claro. Debería hacer algo específico y hacerlo bien.

Angular distingue los componentes de los servicios para aumentar la modularidad y la reutilización. Al separar la funcionalidad relacionada con la vista de un componente de otros tipos de procesamiento, puede hacer que sus clases de componentes sean ágiles y eficientes.

Un componente delega ciertas tareas a los servicios, como recuperar datos del servidor, validar la entrada del usuario o iniciar sesión directamente en la consola. Al definir dichas tareas de procesamiento en una clase de servicio inyectable, las pone a disposición de cualquier componente.

Angular ayuda a los componentes a facilitar la incorporación de la lógica de los servicios a través de la inyección de dependencia.

Como en el caso de los componente, Angular CLI tiene un comando por el que se generará un archivo básico de service con el nombre del parámetro que le pasemos:

```
ng generate service name_service
```

2.4.1.- Ejemplos de servicios

El siguiente ejemplo es un servicio que muestra los mensajes de la consola del navegador.

```
export class Logger {

  log(msg: any)   { console.log(msg); }
  error(msg: any) { console.error(msg); }
  warn(msg: any)  { console.warn(msg); }

}
```

Los servicios pueden depender de otros servicios.

```
export class HeroService {
  private heroes: Hero[] = [];

  constructor(
    private backend: BackendService,
    private logger: Logger) { }

  getHeroes() {
    this.backend.getAll(Hero).then( (heroes: Hero[]) => {
      this.logger.log(`Fetched ${heroes.length} heroes.`);
      this.heroes.push(...heroes); // fill cache
    });
    return this.heroes; }}
```

2.4.2.- Inyección de dependencias (ID)

ID está conectado al marco de Angular y se usa en todas partes para proporcionar nuevos enlaces de componentes con sus servicios y otras cosas que necesitan.

Para definir una clase como un servicio en Angular, se usa el decorador @Injectable() para proporcionar los metadatos que permiten a Angular inyectarlo en un componente como una dependencia.

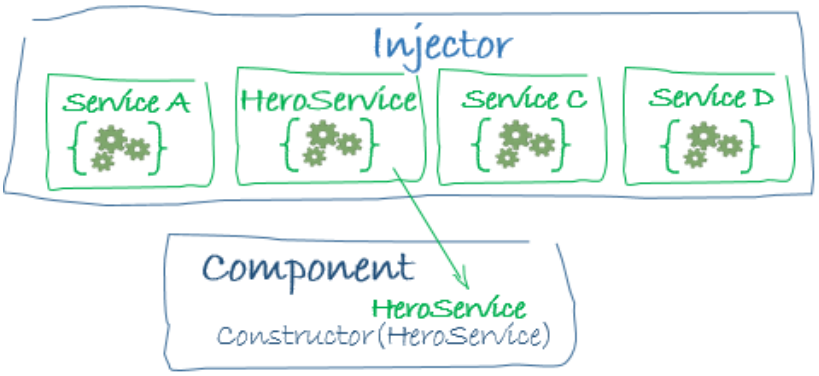
- El inyector es el mecanismo principal. Angular crea un inyector para toda la app durante el proceso de arranque, e inyectores adicionales según sea necesario. No se tienen que crear a mano en ningún momento.
- Un inyector crea dependencias y mantiene un contenedor de instancias de dependencia que reutiliza si es posible.
- Un provider es un objeto que le dice a un inyector cómo obtener o crear una dependencia.

Cuando Angular crea una instancia de una clase de componente, determina qué servicios u otras dependencias necesita ese componente al observar los tipos de parámetros del constructor.

Cuando Angular descubre que un componente depende de un servicio, primero verifica si el inyector tiene alguna instancia existente de ese servicio. Si una instancia de servicio solicitada aún no existe, el inyector hace que el provider agregue al inyector el servicio antes de ser devuelto.

Cuando todos los servicios solicitados se han resuelto y devuelto, Angular puede llamar al constructor del componente con esos servicios como argumentos.

El ejemplo de inyección se ve algo así:



2.4.3.- Proporcionando servicios

Debe registrar al menos un provider de cualquier servicio que se vaya a usar. El proveedor puede ser parte de los propios metadatos del servicio, haciendo que ese servicio esté disponible en todas partes, o puede registrar proveedores con módulos o componentes específicos. Se registran proveedores en los metadatos de los servicios (en el decorador @Injectable()), o en los metadatos de @NgModule() o @Component().

- Por defecto, el comando de Angular CLI ng generate service registra un proveedor con el inyector raíz para su servicio al incluir metadatos del proveedor en el decorador @Injectable().

```
@Injectable({
 providedIn: 'root',
})
```

Cuando se proporciona el servicio en el nivel del proveedor raíz, Angular crea una instancia única y compartida de dicho servicio y la inyecta en cualquier clase que lo solicite. El registro del proveedor en los metadatos también le permite a Angular optimizar una app eliminando el servicio de la aplicación compilada si no se utiliza.

- Cuando se registra un proveedor con un NgModule específico, la misma instancia de un servicio está disponible para todos los componentes en ese NgModule.

```
@NgModule({
  providers: [
  BackendService,
  Logger
 ],
 ...
})
```

- Cuando se registra un proveedor a nivel de componente, obtiene una nueva instancia del servicio con cada nueva instancia de ese componente.

```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
```

2.5.- HTTP

La mayoría de las aplicaciones creadas con Angular 2 se comunican a través de los services con un back-end que les proporciona la información necesaria a través del protocolo HTTP. Los navegadores modernos admiten dos APIs para realizar las peticiones HTTP: la interfaz XMLHttpRequest y fetch() API.

En Angular encontramos un objeto para simplificar las peticiones llamado HttpClient en el paquete de @angular/common/http que se basa en la interfaz XMLHttpRequest expuesta por los navegadores. Pero este objeto HttpClient incluye características de capacidad de prueba, objetos de solicitud y respuesta, interpretación de solicitud y respuesta, Observable api y manejo de errores.

2.5.1.- Setup

Lo primero que se debe hacer para poder utilizar HttpClient es importar el modulo de HttpClientModule. Como se trata de un objeto que se usará bastante se colocará en el AppModule raíz.

```
import { NgModule }         from '@angular/core';
import { BrowserModule }    from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
  ],
  declarations: [
    AppComponent,
  ],
  bootstrap: [ AppComponent ]
})

export class AppModule {}
```

Ahora ya podremos utilizarlo en cualquier service que lo necesitemos.

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

@Injectable()
export class ConfigService {

  constructor(private http: HttpClient) { }

}
```

2.5.2.- JSON data

La mejor forma de recuperar información de un servidor será en formato JSON, por lo que el servidor al que se estén realizando las peticiones deberá estar configurado para que devuelva los datos en este formato.

Un ejemplo de petición sería:

Config.service.ts

```
configUrl = 'url_server';

getConfig() {
  return this.http.get(this.configUrl);
}
```

Config.component.ts

```
showConfig() {
  this.configService.getConfig()
    .subscribe((data: Config) => this.config = {
        heroesUrl: data['heroesUrl'],
        textfile:  data['textfile']
    },
    (error) => console.log(error)
    );
}
```

Un método de HttpClient no comienza su solicitud HTTP hasta que llama al método .subscribe().

2.5.3.- Tipo de peticiones

Además de obtener datos del servidor, *HttpClient* admite solicitudes de PUT, POST y DELETE.

2.5.3.1.- Añadir Headers

Muchos servidores requieren encabezados adicionales para operaciones de guardado. Un ejemplo sería como el siguiente donde enviamos un token de autorización y el tipo de contenido.

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

const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type':  'application/json',
    'Authorization': 'my-auth-token'
  })
};
```

2.5.3.2.- Peticiones POST

```
/** POST: add a new hero to the database */
addHero (hero: Hero): Observable<Hero> {
  return this.http.post<Hero>(this.heroesUrl, hero, httpOptions)
    .pipe(
      catchError(this.handleError('addHero', hero))
    );
}
```

El método de *.post()* es similar al de *.get()* pero necesita dos parámetros más:

- Los datos a enviar en la petición POST en el cuerpo de la solicitud.
- El objeto *httpOptions,* opciones de método que se almacenan en el *headers*.

2.5.3.3.- Peticiones DELETE

Este método elimina un objeto del servidor cuando le pasamos el id del mismo.

```
/** DELETE: delete the hero from the server */
deleteHero (id: number): Observable<{}> {
  const url = `${this.heroesUrl}/${id}`; // DELETE api/heroes/42
  return this.http.delete(url, httpOptions)
    .pipe(
      catchError(this.handleError('deleteHero'))
    );
}
```

2.5.3.4.- Peticiones PUT

Una aplicación enviará una solicitud PUT para actualizar completamente un recurso.

```
/** PUT: update the hero on the server. Returns the updated hero upon success. */
updateHero (hero: Hero): Observable<Hero> {
  return this.http.put<Hero>(this.heroesUrl, hero, httpOptions)
    .pipe(
      catchError(this.handleError('updateHero', hero))
    );
}
```

2.6.- Routing

El router de Angular permite la navegación de una vista a la siguiente a medida que los usuarios realizan tareas de la aplicación. Puede interpretar una URL del navegador como una instrucción para navegar a una vista generada por el cliente. Puede pasar parámetros opcionales. El router registra la actividad en el diario de historial del navegador para que los botones de avance y retroceso también funcionen.

Antes de comenzar a introducir rutas en nuestra aplicación debemos configurar la app para que el router de Angular pueda trabajar y crear un fichero donde almacenar dichas rutas.

En el archivo index.html, justo debajo del comienzo de la etiqueta <head> se debe colocar:

```
<base href=”/”>
```

Esta etiqueta le indicará al router cómo redactar las URLs de navegación.

En Angular, la mejor forma de cargar y configurar las rutas es en un archivo aparte, a nivel del app.module.ts que se dedique a esta función solamente y sea cargado por el AppModule. Para crear este archivo usaremos el siguiente comando de Angular CLI:

```
ng generate module app-routing --flat --module=app
```

Este comando genera un archivo app-routing.module.ts con la clase AppRoutingModule. El archivo generado será algo parecido al siguiente código:

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

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})

export class AppRoutingModule { }
```

Pero este no es el código que se necesita verdaderamente para generar las rutas de nuestra app. Sustituiremos parte del código y quedará como sigue:

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

@NgModule({
  exports: [ RouterModule ]
})

export class AppRoutingModule {}
```

2.6.1.- Añadiendo rutas

Las rutas le indican al router qué vistas mostrar cuando un usuario hace clic en un enlace. Cada dirección tiene dos propiedades mínimo:

- path: una cadena que representa la url a visitar.
- component: el componente que el router debe cargar al navegar a esa ruta.

Siguiendo con el ejemplo anterior ahora nuestro archivo quedaría así:

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

import { HeroesComponent } from ‘./heroes/heroes.component’;
import { HeroDetailComponent  } from ‘./heroDetail/heroDetail.component’;
import { PageNotFoundComponet } from ‘./page/page.component’;

const   routes:   Routes =  [
   {   path: ‘heroes’, component:  HeroesComponent  },
   {   path: ‘heroes/:id’, component:  HeroDetailComponent   },
   {   path: ‘’, redirecTo:  ‘/heroes’, pathMatch:  ‘full’  },
   {   path: ‘**’, component:  PageNotFoundComponent  }
];

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

El método .forRoot() se llama así porque configura el router en la raíz de la app. Este método proporciona los providers y las directivas necesarias para el enrutamiento y realiza la navegación inicial según la url actual.

2.6.2.- Añadir RouterOutlet

En la vista de app.component.html debemos introducir la etiqueta router-outlet que se encarga de mostrar las vistas de los componentes cargados en el router. Aunque lo normal es dejar solo este archivo HTML para esta etiqueta se puede trabajar en este archivo como se crea conveniente para el proyecto.

```
<h1>{{title}}</h1>
<router-outlet></router-outlet>
<app-messages></app-messages>
```

2.6.3.- Navegación entre vistas

Para navegar entre diferentes vistas Angular posee varias formas de realizar esta tarea:

- Con el atributo routerLink en una etiqueta en HTML.
- Con la clase router desde el componente a través de código.

2.6.4.- Extraer parámetros de la url

Si una de las url creadas en el archivo de rutas posee algún parámetro como la siguiente:

```
{ path:  ‘heroes/:id’, component:  HeroDetailComponent  }
```

La forma de recuperar ese valor en la vista de HeroDetailComponent será:

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

…

constructor(…, private activatedRoute: ActivatedRoute ) {
  activatedRoute.params.subscribe( params => {
      this.id = +params['id'];
  });
  …
}
…
```

Los parámetros de router son siempre string. Para convertirlos a número se coloca el operador + delante para hacer la conversión.