Tutorial **Angular 6**:

El tour de los Héroes

Esta es la **documentación archivada para Angular v6.** Visite angular.io para ver la documentación de la versión actual de Angular.

inicio

Las buenas herramientas hacen que el desarrollo de aplicaciones sea más rápido y más fácil de mantener que si lo hiciera todo a mano.

La **CLI angular** es una *interfaz de línea de comandos* herramienta que puede crear un proyecto, añadir archivos, y realizar una variedad de tareas de desarrollo en curso, como las pruebas, empaquetamiento y despliegue.

El objetivo de esta guía es crear y ejecutar una aplicación Angular simple en TypeScript, utilizando la CLI angular y respetando las recomendaciones de la Guía de estilo que benefician a *todos los* proyectos Angular.

Al final del capítulo, tendrá una comprensión básica del desarrollo con la CLI y una base para estos ejemplos de documentación y para las aplicaciones del mundo real.

Y también puedes descargar el ejemplo.

Paso 1. Configurar el Entorno de Desarrollo

Debe configurar su entorno de desarrollo antes de poder hacer algo.

Instale Node.js® y npm si aún no están en su máquina.

> **Verifique que esté ejecutando al menos la versión Node.js `8.x`o superior y la versión npm `5.x`o superior** ejecutando `node -v`y `npm -v`en una ventana de terminal / consola. Las versiones más antiguas producen errores, pero las versiones más nuevas están bien.

Luego instale el CLI angular a nivel mundial.

```
npm install -g @angular/cli
```

Paso 2. Crear un nuevo proyecto

Abra una ventana de terminal.

Genere un nuevo proyecto y una aplicación predeterminada ejecutando el siguiente comando:

```
ng new my-app
```

Angular CLI instala los paquetes npm necesarios, crea los archivos del proyecto y completa el proyecto con una aplicación predeterminada simple. Esto puede llevar algún tiempo.

> Puede agregar funcionalidad preempaquetada a un nuevo proyecto usando el `ng add`comando. El `ng add`comando transforma un proyecto aplicando los esquemas en el paquete especificado. Para obtener más información, consulte la documentación de CLI angular.
>
> El material angular proporciona esquemas para diseños típicos de aplicaciones. Consulte la documentación del material angular para más detalles.

Paso 3: Servir la aplicación

Ir al directorio del proyecto e iniciar el servidor.

```
cd my-app
ng serve --open
```

El `ng serve`comando inicia el servidor, observa sus archivos y reconstruye la aplicación a medida que realiza cambios en esos archivos.

El uso de la opción `--open`(o solo `-o`) abrirá automáticamente su navegador `http://localhost:4200/`.

Tu aplicación te saluda con un mensaje:

Paso 4: Edita tu primer componente angular

El CLI creó el primer componente angular para usted. Este es el *componente raíz* y se llama `app-root`. Puedes encontrarlo en `./src/app/app.component.ts`.

Abra el archivo componente y cambie la `title`propiedad de `'app'`a `'My First Angular App!'`.

src / app / app.component.ts

```
export class AppComponent {
  title = 'My First Angular App!';
}
```

El navegador se vuelve a cargar automáticamente con el título revisado. Eso está bien, pero podría verse mejor.

Abre `src/app/app.component.css`y dale un poco de estilo al componente.

src / app / app.component.css

```
h1 {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 250%;
}
```

¡Se ve bien!

¿Que sigue?

Eso es todo lo que esperas hacer en una aplicación "Hello, World".

Estás listo para tomar el Tutorial de Tour of Heroes y crear una pequeña aplicación que demuestre las grandes cosas que puedes construir con Angular.

O puede quedarse un poco más para aprender sobre los archivos en su nuevo proyecto.

Revisión de los archivos del proyecto

Un proyecto Angular CLI es la base para experimentos rápidos y soluciones empresariales.

El primer archivo que debes revisar es `README.md`. Tiene información básica sobre cómo usar los comandos CLI. Cuando quiera saber más sobre cómo funciona Angular CLI, asegúrese de visitar el repositorio de Angular CLI y Wiki .

Algunos de los archivos generados pueden ser desconocidos para usted.

La carpeta `src`

Tu aplicación vive en la `src`carpeta. Aquí encontrará todos los componentes, plantillas, estilos, imágenes y cualquier otra cosa que necesite su aplicación. Todos los archivos que se encuentran fuera de esta carpeta están diseñados para admitir la creación de su aplicación.

```
src
  app
    app.component.css
    app.component.html
    app.component.spec.ts
    app.component.ts
    app.module.ts
  assets
    .gitkeep
  environments
    environment.prod.ts
    environment.ts
  browserslist
  favicon.ico
  index.html
  karma.conf.js
  main.ts
  polyfills.ts
  styles.css
  test.ts
  tsconfig.app.json
  tsconfig.spec.json
  tslint.json
```

Archivo	Propósito
`app/app.component.{ts,html,css,spec.ts}`	Define `AppComponent` junto con una plantilla HTML, una hoja de estilo CSS y una prueba de unidad. Es el componente **raíz** de lo que se convertirá en un árbol de componentes anidados a medida que la aplicación evolucione.
`app/app.module.ts`	Define `AppModule`, el módulo raíz que le dice a Angular cómo ensamblar la aplicación. En este momento se declara sólo el `AppComponent`. Pronto habrá más componentes para declarar.
`assets/*`	Una carpeta donde puede colocar imágenes y cualquier otra cosa para copiar al por mayor cuando compile su aplicación.
`environments/*`	Esta carpeta contiene un archivo para cada uno de sus entornos de destino, cada uno de los cuales exporta variables de configuración simples para usar en su aplicación. Los archivos se reemplazan sobre la marcha cuando crea su aplicación. Puede usar un punto final de API diferente para el desarrollo que para la producción o tal vez diferentes tokens de análisis. Incluso podría utilizar algunos servicios simulados. De cualquier manera, el CLI lo tiene cubierto.
`browserslist`	Un archivo de configuración para compartir navegadores de destino entre diferentes herramientas de front-end.
`favicon.ico`	Cada sitio quiere verse bien en la barra de marcadores. Comienza con tu propio ícono angular.
`index.html`	La página HTML principal que se sirve cuando alguien visita su sitio. La mayoría de las veces nunca necesitarás editarlo. La CLI agrega automáticamente todos `js`y `css`archivos cuando la construcción de su aplicación por lo que no será necesario añadir ningún `<script>`o `<link>`etiquetas aquí manualmente.
`karma.conf.js`	Configuración de prueba unitaria para el corredor de prueba Karma , utilizada al ejecutar `ng test`.

`main.ts`	El principal punto de entrada para tu aplicación. Compila la aplicación con el compilador JIT y arranca el módulo raíz de la aplicación (`AppModule`) para que se ejecute en el navegador. También puede usar el compilador AOT sin cambiar ningún código agregando la `--aot`bandera a los comandos `ng build`y `ng serve`.
`polyfills.ts`	Los diferentes navegadores tienen diferentes niveles de soporte de los estándares web. Los polyfills ayudan a normalizar esas diferencias. Debería estar bastante seguro con `core-js`y `zone.js`, pero asegúrese de consultar la Guía de soporte del navegador para obtener más información.
`styles.css`	Tus estilos globales van aquí. La mayoría de las veces, querrá tener estilos locales en sus componentes para facilitar el mantenimiento, pero los estilos que afectan a todas sus aplicaciones deben estar en un lugar central.
`test.ts`	Este es el punto de entrada principal para tus pruebas unitarias. Tiene alguna configuración personalizada que puede ser desconocida, pero no es algo que deba editar.
`tsconfig.{app|spec}.json`	Configuración del compilador de TypeScript para la aplicación Angular (`tsconfig.app.json`) y para las pruebas unitarias (`tsconfig.spec.json`).
`tslint.json`	Configuración adicional de Linting para TSLint junto con Codelyzer , utilizada cuando se ejecuta `ng lint`. Linting ayuda a mantener su estilo de código consistente.

La carpeta raíz

La `src/`carpeta es solo uno de los elementos dentro de la carpeta raíz del proyecto. Otros archivos lo ayudan a construir, probar, mantener, documentar e implementar la aplicación. Estos archivos van en la carpeta raíz al lado de `src/`.

```
my_app
  e2e
    src
```

```
        app.e2e-spec.ts
        app.po.ts
    tsconfig.e2e.json
    protractor.conf.js
node_modules / ...
src / ...
    karma.conf.js
.editorconfig
.gitignore
angular.json
paquete.json
README.md
tsconfig.json
tslint.json
```

Archivo	Propósito
`e2e/`	En `e2e/` el interior viven las pruebas de punta a punta. No deberían estar dentro `src/` porque las pruebas e2e son en realidad una aplicación separada que simplemente prueba tu aplicación principal. Es por eso que también tienen el suyo propio `tsconfig.e2e.json`.
`node_modules/`	`Node.js` crea esta carpeta y coloca todos los módulos de terceros listados en su `package.json` interior.
`.editorconfig`	Configuración simple para su editor para asegurar que todos los que usan su proyecto tengan la misma configuración básica. La mayoría de los editores admiten un `.editorconfig` archivo. Ver http://editorconfig.org para más información.
`.gitignore`	Configuración de Git para asegurarse de que los archivos generados automáticamente no estén comprometidos con el control de fuente.
`angular.json`	Configuración para CLI angular. En este archivo puede establecer varios valores predeterminados y también configurar qué archivos se incluyen cuando se construye su proyecto. Revisa la documentación oficial si quieres saber más.
`package.json`	`npm` configuración que enumera los paquetes de terceros que utiliza su proyecto. También puede agregar sus propios scripts personalizados aquí.
`protractor.conf.js`	Configuración de prueba de extremo a extremo para el transportador , que se utiliza cuando se ejecuta `ng e2e`.
`README.md`	Documentación básica para su proyecto, precargada con información de comandos de CLI. ¡Asegúrate de mejorarlo con la documentación del proyecto para que cualquiera que visite el repositorio pueda construir tu aplicación!
`tsconfig.json`	La configuración del compilador de TypeScript para su IDE para recoger y darle herramientas útiles.

`tslint.json`	Configuración de linting para TSLint junto con Codelyzer , utilizada cuando se ejecuta `ng lint`. Linting ayuda a mantener su estilo de código consistente.

Siguiente paso

Si eres nuevo en Angular, continúa con el tutorial . Puede omitir el paso de "Configuración" ya que ya está utilizando la configuración de CLI angular.

Esta es la **documentación archivada para Angular v6.** Visite angular.io para ver la documentación de la versión actual de Angular.

Tutorial: Tour de Heroes

El tutorial *de Tour of Heroes* cubre los fundamentos de Angular.
En este tutorial, construirá una aplicación que ayude a una agencia de personal a gestionar sus establos de héroes.

Esta aplicación básica tiene muchas de las características que esperaría encontrar en una aplicación basada en datos. Adquiere y muestra una lista de héroes, edita los detalles de un héroe seleccionado y navega entre diferentes vistas de datos heroicos.

Al final del tutorial podrás hacer lo siguiente:

- Use directivas angulares incorporadas para mostrar y ocultar elementos y mostrar listas de datos de héroe.
- Crea componentes angulares para mostrar detalles de héroes y mostrar una gran variedad de héroes.
- Utilice el enlace de datos de una sola vía para datos de solo lectura.
- Agregue campos editables para actualizar un modelo con enlace de datos bidireccional.
- Enlazar métodos de componentes a eventos de usuario, como pulsaciones de teclas y clics.
- Permite a los usuarios seleccionar un héroe de una lista maestra y editar ese héroe en la vista de detalles.
- Formato de datos con las tuberías.
- Crea un servicio compartido para reunir a los héroes.
- Utilice el enrutamiento para navegar entre diferentes vistas y sus componentes.

Aprenderá suficiente Angular para comenzar y ganará la confianza de que Angular puede hacer lo que necesite.

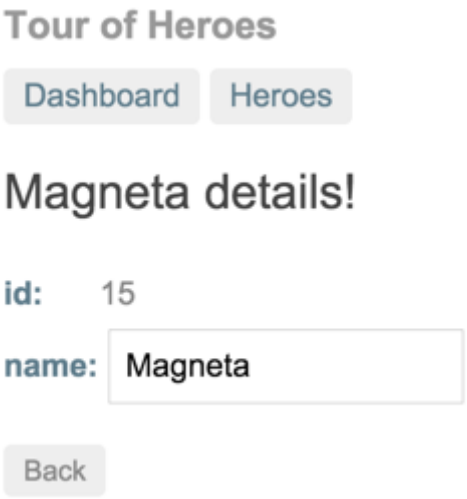
Después de completar todos los pasos del tutorial, la aplicación final se verá así ejemplo vivo / ejemplo de descarga.

Lo que vas a construir

Aquí hay una idea visual de a dónde conduce este tutorial, comenzando con la vista "Panel de control" y los héroes más heroicos:

Puede hacer clic en los dos enlaces que se encuentran sobre el tablero ("Cuadro de mando" y "Héroes") para navegar entre esta vista del cuadro de mandos y la vista de Héroes.

Si haces clic en el héroe del panel "Magneta", el enrutador abre una vista de "Detalles del héroe" donde puedes cambiar el nombre del héroe.

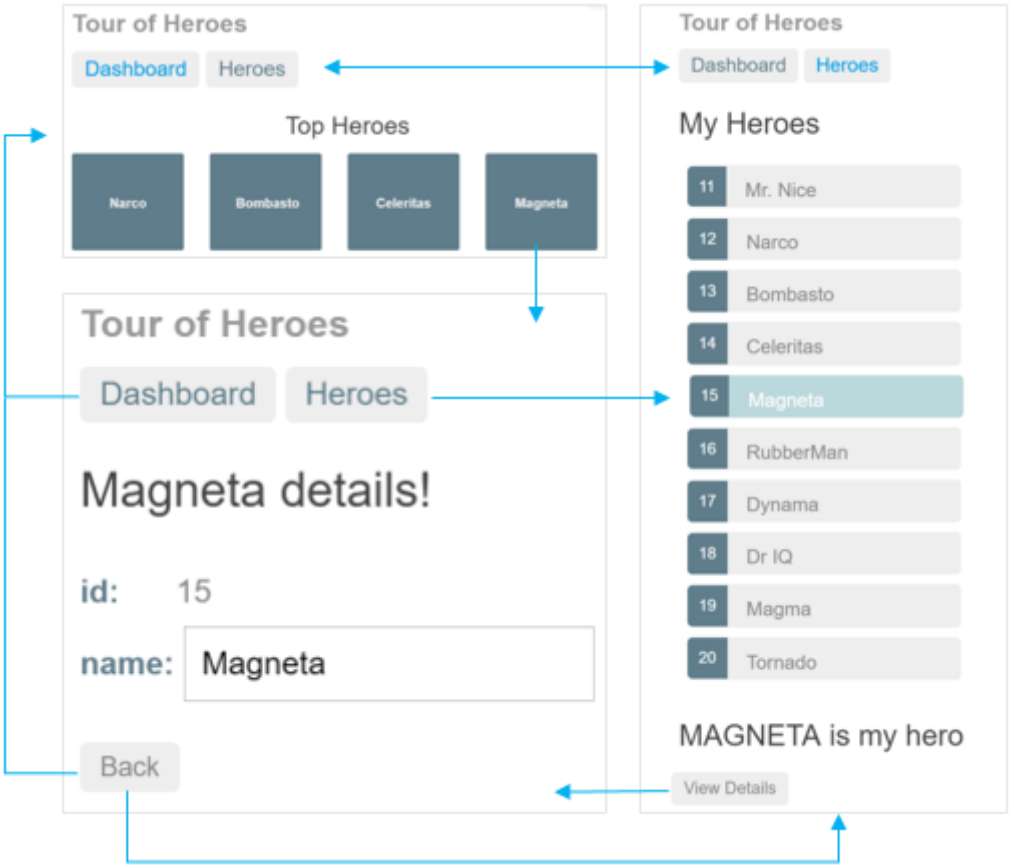


Al hacer clic en el botón "Atrás", volverá al panel. Los enlaces en la parte superior lo llevan a cualquiera de las vistas principales. Si hace clic en "Héroes", la aplicación muestra la vista de la lista maestra "Héroes".

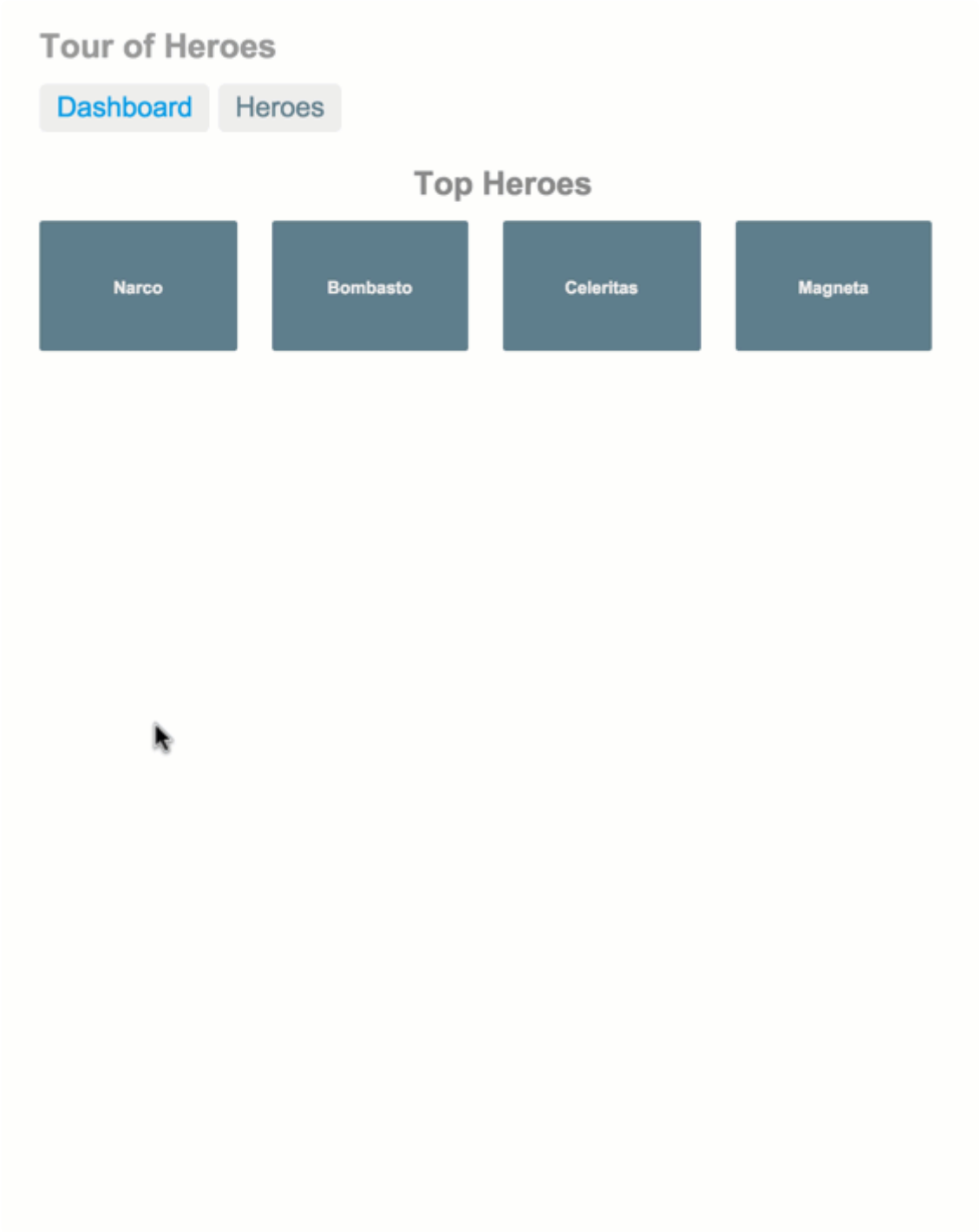
Cuando haces clic en un nombre de héroe diferente, el mini detalle de solo lectura debajo de la lista refleja la nueva opción.

Puede hacer clic en el botón "Ver detalles" para profundizar en los detalles editables del héroe seleccionado.

El siguiente diagrama captura todas las opciones de navegación.



Aquí está la aplicación en acción:


Tour of Heroes
Dashboard
Heroes
Top Heroes
Narco
Bombasto
Celeritas
Magneta

Esta es la **documentación archivada para Angular v6.** Visite angular.io para ver la documentación de la versión actual de Angular.

El Shell de aplicación

Instale el CLI angular

Instale la CLI angular , si aún no lo ha hecho.

```
npm install -g @angular/cli
```

Crear un nuevo aplicación

Cree un nuevo proyecto nombrado `angular-tour-of-heroes` con este comando CLI.

```
ng new angular-tour-of-heroes
```

Angular CLI generó un nuevo proyecto con una aplicación predeterminada y archivos de soporte.

> Puede agregar funcionalidad preempaquetada a un nuevo proyecto usando el `ng add`comando. El `ng add`comando transforma un proyecto aplicando los esquemas en el paquete especificado. Para obtener más información, consulte la documentación de CLI angular.
>
> El material angular proporciona esquemas para diseños típicos de aplicaciones. Consulte la documentación del material angular para más detalles.

Servir la aplicación

Ir al directorio del proyecto e iniciar la aplicación.

```
cd angular-tour-of-heroes
ng serve --open
```

> El comando `ng serve` crea la aplicación, inicia el servidor de desarrollo, observa los archivos de origen y reconstruye la aplicación a medida que realiza cambios en esos archivos.
>
> La `--open` bandera abre un navegador para `http://localhost:4200/`.

Debería ver la aplicación ejecutándose en su navegador.

Componentes de Angular

La página que ves es la *aplicación shell* . La cáscara está controlada por un **componente** Angular llamado `AppComponent`.

Los componentes son los *componentes* fundamentales de las aplicaciones angulares. Muestran datos en la pantalla, escuchan las opiniones del usuario y toman medidas en función de esa entrada.

Cambiar el título de la aplicación

Abra el proyecto en su editor favorito o IDE y navegue hasta la `src/app`carpeta.

Encontrarás la implementación del shell `AppComponent`distribuido en tres archivos:

1. `app.component.ts`- el código de clase del componente, escrito en TypeScript.
2. `app.component.html`- La plantilla del componente, escrita en HTML.
3. `app.component.css`- Los estilos CSS privados del componente.

Abra el archivo de clase de componente (`app.component.ts`) y cambie el valor de la `title`propiedad a 'Tour of Heroes'.

app.component.ts (propiedad de título de clase)

```
title = 'Tour of Heroes';
```

Abra el archivo de plantilla de componente (`app.component.html`) y elimine la plantilla predeterminada generada por la CLI angular. Reemplácelo con la siguiente línea de HTML.

app.component.html (plantilla)

```
<h1>{{title}}</h1>
```

Las llaves dobles son la sintaxis de *enlace de interpolación* de Angular . Este enlace de interpolación presenta el `title` valor de propiedad del componente dentro de la etiqueta de encabezado HTML.

El navegador actualiza y muestra el nuevo título de la aplicación.

Añadir estilos de aplicación

La mayoría de las aplicaciones se esfuerzan por lograr una apariencia coherente en toda la aplicación. El CLI generó un vacío `styles.css` para este propósito. Ponga sus estilos de toda la aplicación allí.

He aquí un extracto de la `styles.css` para el *tour de los Héroes* aplicación de ejemplo.

src / styles.css (extracto)

```
/* Application-wide Styles */
h1 {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 250%;
}
h2, h3 {
  color: #444;
  font-family: Arial, Helvetica, sans-serif;
  font-weight: lighter;
}
body {
  margin: 2em;
}
body, input[text], button {
  color: #888;
  font-family: Cambria, Georgia;
}
/* everywhere else */
* {
  font-family: Arial, Helvetica, sans-serif;
}
```

Resumen

- Usted creó la estructura inicial de la aplicación utilizando la CLI angular.
- Aprendiste que los componentes angulares muestran datos.
- Usaste las llaves dobles de interpolación para mostrar el título de la aplicación.

Esta es la **documentación archivada para Angular v6.** Visite angular.io para ver la documentación de la versión actual de Angular.

El editor del Héroe

La aplicación ahora tiene un título básico. A continuación, creará un nuevo componente para mostrar información de héroe y colocar ese componente en el shell de la aplicación.

Crea el componente de héroes.

Usando la CLI angular, genere un nuevo componente llamado `heroes` .

```
ng generate component heroes
```

La CLI crea una nueva carpeta `src/app/heroes/` y genera los tres archivos de `HeroesComponent` .

El `HeroesComponent` archivo de clase es el siguiente:

app / heroes / heroes.component.ts (versión inicial)

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

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

  constructor() { }

  ngOnInit() {
  }

}
```

Siempre importa el `Component` símbolo de la biblioteca principal Angular y anote la clase de componente con .`@Component`

`@Component` es una función decoradora que especifica los metadatos angulares para el componente.

El CLI generó tres propiedades de metadatos:

1. `selector` - el selector de elementos CSS del componente
2. `templateUrl` - la ubicación del archivo de plantilla del componente.
3. `styleUrls` - la ubicación de los estilos CSS privados del componente.

El selector de elementos CSS , `'app-heroes'` , coincide con el nombre del elemento HTML que identifica este componente dentro de la plantilla de un componente de los padres.

El `ngOnInit` es un gancho de ciclo de vida . Llamadas angulares `ngOnInit` poco después de crear un componente. Es un buen lugar para poner la lógica de inicialización.

Siempre `export` la clase de componentes para que pueda `import` en otro lugar ... como en el `AppModule` .

Añadir un propiedad de *héroe*

Agrega una `hero` propiedad al `HeroesComponent` héroe llamado "Tormenta de viento".

heroes.component.ts (propiedad de héroe)

```
hero = 'Windstorm';
```

Mostrar el héroe

Abra el `heroes.component.html` archivo de plantilla. Elimine el texto predeterminado generado por el CLI angular y reemplácelo con un enlace de datos a la nueva `hero` propiedad.

heroes.component.html

```
{{hero}}
```

Mostrar la vista *HeroesComponent*

Para mostrar el `HeroesComponent` , debe agregarlo a la plantilla del shell `AppComponent` .

Recuerda que `app-heroes` es el elemento selector para el `HeroesComponent` . Así que agregue un `<app-heroes>` elemento al `AppComponent` archivo de plantilla, justo debajo del título.

src / app / app.component.html

```
<h1>{{title}}</h1>
<app-heroes></app-heroes>
```

Suponiendo que el `ng serve` comando CLI aún se está ejecutando, el navegador debería actualizar y mostrar tanto el título de la aplicación como el nombre del héroe.

Crear una clase de héroe

Un verdadero héroe es más que un nombre.

Crear una `Hero` clase en su propio archivo en la `src/app` carpeta. Dale `id` y `name` propiedades.

src / app / hero.ts

```
export class Hero {
  id: number;
  name: string;
}
```

Regresa a la `HeroesComponent` clase e importa la `Hero` clase.

Refactoriza la `hero` propiedad del componente para que sea de tipo `Hero` . Inicialízalo con una `id` de `1` y el nombre `Windstorm` .

El `HeroesComponent` archivo de clase revisado debería verse así:

src / app / heroes / heroes.component.ts

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

@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css']
})
export class HeroesComponent implements OnInit {
  hero: Hero = {
    id: 1,
    name: 'Windstorm'
  };

  constructor() { }

  ngOnInit() {
  }

}
```

La página ya no se muestra correctamente porque cambiaste el héroe de una cadena a un objeto.

Mostrar el objeto héroe.

Actualice el enlace en la plantilla para anunciar el nombre del héroe y muestre ambos `id` y `name` en un diseño de detalles como este:

heroes.component.html (plantilla de HeroesComponent)

```
<h2>{{hero.name}} Details</h2>
<div><span>id: </span>{{hero.id}}</div>
<div><span>name: </span>{{hero.name}}</div>
```

El navegador actualiza y muestra la información del héroe.

Formato con el *UppercasePipe*

Modificar el `hero.name` enlace de esta manera.

```
<h2>{{hero.name | uppercase}} Details</h2>
```

El navegador se actualiza y ahora el nombre del héroe se muestra en mayúsculas.

La palabra `uppercase` en el enlace de interpolación, justo después del operador de tubería (|), activa el incorporado `UppercasePipe` .

Las canalizaciones son una buena forma de formatear cadenas, montos de moneda, fechas y otros datos de visualización. Naves angulares con varios tubos incorporados y puedes crear el tuyo propio.

Edita el heroe

Los usuarios deben poder editar el nombre del héroe en un `<input>` cuadro de texto.

El cuadro de texto debe *mostrar* la `name` propiedad del héroe y *actualizar* esa propiedad a medida que el usuario escribe. Eso significa que el flujo de datos de la clase de componente *sale a la pantalla* y de la pantalla *a la clase* .

Para automatizar ese flujo de datos, configure un enlace de datos bidireccional entre el `<input>` elemento de formulario y la `hero.name` propiedad.

enlace bidireccional

Refactoriza el área de detalles en la `HeroesComponent` plantilla para que se vea así:

src / app / heroes / heroes.component.html (plantilla de HeroesComponent)

```
<div>
  <label>name:
    <input [(ngModel)]="hero.name" placeholder="name">
  </label>
</div>
```

[(ngModel)] es la sintaxis de enlace de datos bidireccional de Angular.

Aquí vincula la `hero.name` propiedad con el cuadro de texto HTML para que los datos puedan fluir *en ambas direcciones:* desde la `hero.name` propiedad al cuadro de texto y desde el cuadro de texto a la `hero.name` .

El *FormsModule que* falta

Observe que la aplicación dejó de funcionar cuando agregó .`[(ngModel)]`

Para ver el error, abra las herramientas de desarrollo del navegador y busque en la consola un mensaje como

```
Template parse errors:
Can't bind to 'ngModel' since it isn't a known property of 'input'.
```

Aunque `ngModel` es una directiva angular válida, no está disponible de forma predeterminada.

Pertenece a lo opcional `FormsModule` y debe *optar por* usarlo.

AppModule

Angular necesita saber cómo encajan las piezas de su aplicación y qué otros archivos y bibliotecas requiere la aplicación. Esta información se llama *metadata.*

Algunos de los metadatos se encuentran en los decoradores que agregó a sus clases de componentes. Otros metadatos críticos están en los decoradores.`@Component` `@NgModule`

El decorador más importante anota la clase **AppModule de** nivel **superior** .`@NgModule`

El CLI angular generó una `AppModule` clase en `src/app/app.module.ts` cuando creó el proyecto. Aquí es donde *optas por* el `FormsModule` .

Importar *FormsModule*

Abra `AppModule` (`app.module.ts`) e importe el `FormsModule` símbolo de la `@angular/forms` biblioteca.

app.module.ts (Importación de símbolos de FormsModule)

```
import { FormsModule } from '@angular/forms'; // <-- NgModel lives here
```

Luego, agregue `FormsModule` a la matriz de metadatos , que contiene una lista de módulos externos que la aplicación necesita.`@NgModule` `imports`

app.module.ts (@NgModule importaciones)

```
imports: [
  BrowserModule,
  FormsModule
],
```

Cuando el navegador se actualice, la aplicación debería funcionar de nuevo. Puede editar el nombre del héroe y ver los cambios reflejados inmediatamente en `<h2>` el cuadro de texto anterior.

Declare *HeroesComponent*

Cada componente debe ser declarado en *exactamente un* NgModule .

No declaraste el `HeroesComponent` . Entonces, ¿por qué funcionó la aplicación?

Funcionó porque la CLI angular se declaró `HeroesComponent` en el `AppModule` momento en que generó ese componente.

Abrir `src/app/app.module.ts` y encontrar `HeroesComponent` importados cerca de la parte superior.

```
import { HeroesComponent } from './heroes/heroes.component';
```

Se `HeroesComponent` declara en la matriz. `@NgModule.declarations`

```
declarations: [
  AppComponent,
  HeroesComponent
],
```

Tenga en cuenta que `AppModule` declara ambos componentes de la aplicación, `AppComponent` y `HeroesComponent` .

Resumen

- Usaste el CLI para crear un segundo `HeroesComponent` .
- Lo mostraste `HeroesComponent` agregándolo al `AppComponent` shell.
- Se aplicó el `UppercasePipe` para formatear el nombre.
- Se utilizó enlace de datos bidireccional con la `ngModel` directiva.
- Usted aprendió sobre el `AppModule` .
- Usted importó el código `FormsModule` en el `AppModule` que Angular reconocería y aplicaría la `ngModel` directiva.
- Aprendió la importancia de declarar componentes en `AppModule` y agradeció que CLI lo declarara por usted.

Esta es la **documentación archivada para Angular v6.** Visite angular.io para ver la documentación de la versión actual de Angular.

Mostrar una lista de héroes

En esta página, expandirás la aplicación Tour of Heroes para mostrar una lista de héroes y permitir a los usuarios seleccionar un héroe y mostrar los detalles del héroe.

Crear bosquejos de héroes

Necesitarás algunos héroes para mostrar.

Eventualmente los obtendrá de un servidor de datos remoto. Por ahora, crearás algunos *héroes simulados* y fingirás que provienen del servidor.

Crea un archivo llamado `mock-heroes.ts` en la `src/app/` carpeta. Define una `HEROES` constante como una matriz de diez héroes y expórtala. El archivo debería verse así.

src / app / mock-heroes.ts

```
import { Hero } from './hero';

export const HEROES: Hero[] = [
  { id: 11, name: 'Mr. Nice' },
  { id: 12, name: 'Narco' },
  { id: 13, name: 'Bombasto' },
  { id: 14, name: 'Celeritas' },
  { id: 15, name: 'Magneta' },
  { id: 16, name: 'RubberMan' },
  { id: 17, name: 'Dynama' },
  { id: 18, name: 'Dr IQ' },
  { id: 19, name: 'Magma' },
  { id: 20, name: 'Tornado' }
];
```

Viendo héroes

Estás a punto de mostrar la lista de héroes en la parte superior de la `HeroesComponent` .

Abre el `HeroesComponent` archivo de clase e importa el simulacro `HEROES` .

src / app / heroes / heroes.component.ts (import HEROES)

```
import { HEROES } from '../mock-heroes';
```

En el mismo archivo (`HeroesComponent` clase), defina una propiedad de componente llamada `heroes` para exponer la `HEROES` matriz para el enlace.

```
export class HeroesComponent implements OnInit {

  heroes = HEROES;
```

Lista de héroes con * *ngFor*

Abra el `HeroesComponent` archivo de plantilla y realice los siguientes cambios:

- Añadir un `<h2>` en la parte superior,
- Debajo de esto, agregue una lista HTML desordenada (`<ul>`)
- Insertar un `<li>` dentro del `<ul>` que muestra las propiedades de un `hero` .
- Espolvoree algunas clases de CSS para el estilo (agregará los estilos CSS en breve).

Haz que se vea así:

heroes.component.html (plantilla de héroes)

```
<h2>My Heroes</h2>
<ul class="heroes">
  <li>
    <span class="badge">{{hero.id}}</span> {{hero.name}}
  </li>
</ul>
```

Ahora cambia el `<li>` a esto:

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

El `*ngFor` es de angular *repetidor* Directiva. Repite el elemento host para cada elemento en una lista.

En este ejemplo

- `<li>` es el elemento host
- `heroes` Es la lista de la `HeroesComponent` clase.
- `hero` contiene el objeto de héroe actual para cada iteración a través de la lista.

> No olvides el asterisco (*) delante de `ngFor` . Es una parte crítica de la sintaxis.

Después de que el navegador se actualiza, aparece la lista de héroes.

El estilo de los heroes

La lista de héroes debe ser atractiva y debe responder visualmente cuando los usuarios se desplazan y seleccionan un héroe de la lista.

En el primer tutorial , establece los estilos básicos para toda la aplicación en `styles.css` . Esa hoja de estilo no incluía estilos para esta lista de héroes.

Puede agregar más estilos `styles.css` y seguir creciendo esa hoja de estilos a medida que agrega componentes.

Puede preferir, en cambio, definir estilos privados para un componente específico y mantener todo lo que necesita un componente: el código, el HTML y el CSS, todo en un solo lugar.

Este enfoque hace que sea más fácil reutilizar el componente en otro lugar y entregar el aspecto deseado del componente incluso si los estilos globales son diferentes.

Los estilos privados se definen en línea en la matriz o como archivos de hojas de estilo identificados en la matriz.`@Component.styles` `@Component.styleUrls`

Cuando el CLI generó el `HeroesComponent` , creó una `heroes.component.css` hoja de estilo vacía para el `HeroesComponent` y así lo señaló .`@Component.styleUrls`

src / app / heroes / heroes.component.ts (@Component)

```
@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css']
})
```

Abra el `heroes.component.css` archivo y pegue los estilos CSS privados para el archivo `HeroesComponent` . Los encontrará en la revisión final del código al final de esta guía.

> Los estilos y las hojas de estilo identificados en los metadatos están orientados a ese componente específico. Los estilos se aplican solo a y no afectan el HTML externo o el HTML en ningún otro componente.`@Component` `heroes.component.css` `HeroesComponent`

Maestro / Detalle

Cuando el usuario hace clic en un héroe en la lista **maestra** , el componente debe mostrar los **detalles** del héroe seleccionado en la parte inferior de la página.

En esta sección, escucharás el evento de clic del elemento héroe y actualizarás el detalle del héroe.

Añadir un enlace de evento de clic

Agrega un enlace de evento de clic a `<li>` este como:

heroes.component.html (extracto de la plantilla)

```
<li *ngFor="let hero of heroes" (click)="onSelect(hero)">
```

Este es un ejemplo de la sintaxis de enlace de eventos de Angular .

Los paréntesis alrededor le `click` dicen a Angular que escuche el evento `<li>` del elemento `click` . Cuando el usuario hace clic en `<li>` , Angular ejecuta la `onSelect(hero)` expresión.

`onSelect()` Es un `HeroesComponent` método que estás a punto de escribir. Angular lo llama con el `hero` objeto mostrado en el clic `<li>` , el mismo `hero` definido previamente en la expresión.`*ngFor`

Añadir el controlador de eventos clic

Cambie el nombre de la `hero` propiedad del componente a `selectedHero` pero no la asigne. No hay *héroe seleccionado* cuando se inicia la aplicación.

Agregue el siguiente `onSelect()` método, que asigna el héroe seleccionado de la plantilla a los componentes `selectedHero` .

src / app / heroes / heroes.component.ts (onSelect)

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

Actualizar la plantilla de detalles

La plantilla aún se refiere a la antigua `hero` propiedad del componente que ya no existe. Renombrar `hero` a `selectedHero` .

heroes.component.html (detalles del héroe seleccionado)

```
<h2>{{selectedHero.name | uppercase}} Details</h2>
<div><span>id: </span>{{selectedHero.id}}</div>
<div>
  <label>name:
    <input [(ngModel)]="selectedHero.name" placeholder="name">
  </label>
</div>
```

Ocultar detalles vacíos con *ngIf*

Después de que el navegador se actualiza, la aplicación se rompe.

Abra las herramientas de desarrollo del navegador y busque en la consola un mensaje de error como este:

```
HeroesComponent.html:3 ERROR TypeError: Cannot read property 'name' of undefined
```

Ahora haga clic en uno de los elementos de la lista. La aplicación parece estar funcionando de nuevo. Los héroes aparecen en una lista y los detalles sobre el héroe seleccionado aparecen en la parte inferior de la página.

¿Que pasó?

Cuando se inicia la aplicación, `selectedHero` es `undefined` *por diseño* .

Encuadernación expresiones en la plantilla que se refieren a propiedades de `selectedHero` - expresiones como `{{selectedHero.name}}` - *debe fallar* porque no hay ningún héroe seleccionado.

La corrección

El componente solo debería mostrar los detalles del héroe seleccionado si `selectedHero` existe.

Envuelve el detalle del héroe HTML en un archivo `<div>` . Agrega la directiva de Angular a y configúralo en .`*ngIf` `<div>` `selectedHero`

> No olvides el asterisco (*) delante de `ngIf` . Es una parte crítica de la sintaxis.

src / app / heroes / heroes.component.html (* nglf)

```
<div *ngIf="selectedHero">

  <h2>{{selectedHero.name | uppercase}} Details</h2>
  <div><span>id: </span>{{selectedHero.id}}</div>
  <div>
    <label>name:
      <input [(ngModel)]="selectedHero.name" placeholder="name">
    </label>
  </div>

</div>
```

Después de que el navegador se actualiza, la lista de nombres vuelve a aparecer. El área de detalles está en blanco. Haga clic en un héroe y sus detalles aparecen.

Por qué funciona

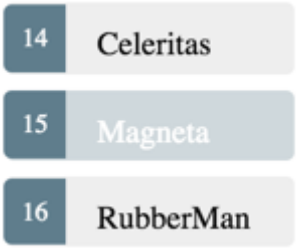
Cuando `selectedHero` no está definido, `ngIf` elimina el detalle del héroe del DOM. No hay `selectedHero` ataduras de las que preocuparse.

Cuando el usuario elige un héroe, `selectedHero` tiene un valor y `ngIf` coloca el detalle del héroe en el DOM.

Estilo del héroe seleccionado

Es difícil identificar al *héroe seleccionado* en la lista cuando todos los `<li>` elementos se parecen.

Si el usuario hace clic en "Magneta", ese héroe debe renderizar con un color de fondo distintivo pero sutil como este:



El coloreado del *héroe seleccionado* es el trabajo de la `.selected` clase CSS en los estilos que agregaste anteriormente . Solo tiene que aplicar la `.selected` clase a `<li>` cuando el usuario haga clic en ella.

El enlace de clase Angular facilita agregar y eliminar una clase CSS condicionalmente. Solo agrégalo `[class.some-css-class]="some-condition"` al elemento que quieras estilizar.

Agregue el siguiente `[class.selected]` enlace a la `<li>` en la `HeroesComponent` plantilla:

heroes.component.html (para alternar la clase CSS 'seleccionada')

```
[class.selected]="hero === selectedHero"
```

Cuando el héroe de la fila actual es el mismo que el `selectedHero` , Angular agrega la `selected` clase CSS. Cuando los dos héroes son diferentes, Angular elimina la clase.

El acabado se `<li>` ve así:

heroes.component.html (héroe de elemento de lista)

```
<li *ngFor="let hero of heroes"
  [class.selected]="hero === selectedHero"
  (click)="onSelect(hero)">
  <span class="badge">{{hero.id}}</span> {{hero.name}}
</li>
```

Resumen

- La aplicación Tour of Heroes muestra una lista de héroes en una vista Maestro / Detalle.
- El usuario puede seleccionar un héroe y ver los detalles de ese héroe.
- Solías mostrar una lista.`*ngFor`
- Que utilizó para incluir o excluir condicionalmente un bloque de HTML.`*ngIf`
- Puede alternar una clase de estilo CSS con un `class` enlace.

Componentes Master/Detail

En este momento, `HeroesComponent` muestra la lista de héroes y los detalles del héroe seleccionado.

Mantener todas las funciones en un componente a medida que la aplicación crezca no se podrá mantener. Querrá dividir los componentes grandes en subcomponentes más pequeños, cada uno centrado en una tarea o flujo de trabajo específico.

En esta página, dará el primer paso en esa dirección moviendo los detalles del héroe a un lugar separado y reutilizable `HeroDetailComponent` .

La `HeroesComponent` única presentará la lista de héroes. El `HeroDetailComponent` presentará los detalles de un héroe seleccionado.

Hacer el `HeroDetailComponent`

Utilice la CLI angular para generar un nuevo componente llamado `hero-detail` .

```
ng generate component hero-detail
```

El comando de andamios lo siguiente:

- Crea un directorio `src/app/hero-detail` .

Dentro de ese directorio se generan cuatro archivos:

- Un archivo CSS para los estilos de componentes.
- Un archivo HTML para la plantilla del componente.
- Un archivo de TypeScript con una clase de componente llamada `HeroDetailComponent` .
- Un archivo de prueba para la `HeroDetailComponent` clase.

El comando también agrega el `HeroDetailComponent` como una declaración en el decorador del archivo.`@NgModule` `src/app/app.module.ts`

Escribe la plantilla

Corte el HTML para el detalle del héroe desde la parte inferior de la `HeroesComponent` plantilla y péguelo en la `HeroDetailComponent` plantilla generada en la plantilla.

El HTML pegado se refiere a a `selectedHero` . Lo nuevo `HeroDetailComponent` puede presentar *cualquier* héroe, no solo un héroe seleccionado. Así que reemplaza "selectedHero" con "hero" en todas partes de la plantilla.

Cuando hayas terminado, la `HeroDetailComponent` plantilla debería verse así:

src / app / hero-detail / hero-detail.component.html

```
<div *ngIf="hero">

  <h2>{{hero.name | uppercase}} Details</h2>
  <div><span>id: </span>{{hero.id}}</div>
  <div>
    <label>name:
      <input [(ngModel)]="hero.name" placeholder="name"/>
    </label>
  </div>

</div>
```

Añade la propiedad del héroe.`@()`

La `HeroDetailComponent` plantilla se enlaza a la `hero` propiedad del componente que es de tipo `Hero` .

Abra el `HeroDetailComponent` archivo de clase e importe el `Hero` símbolo.

src / app / hero-detail / hero-detail.component.ts (import Hero)

```
import { Hero } from '../hero';
```

La `hero` propiedad debe ser una propiedad de *entrada* , anotada con el decorador, porque la *externa* se enlazará de esta manera.`@Input()` `HeroesComponent`

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

Modifique la `@angular/core` declaración de importación para incluir el `Input` símbolo.

src / app / hero-detail / hero-detail.component.ts (entrada de importación)

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

Añade una `hero` propiedad, precedida por el decorador.`@Input()`

```
@Input() hero: Hero;
```

Ese es el único cambio que debes hacer en la `HeroDetailComponent` clase. No hay más propiedades. No hay lógica de presentación. Este componente simplemente recibe un objeto héroe a través de su `hero` propiedad y lo muestra.

Mostrar la `HeroDetailComponent`

El `HeroesComponent` sigue siendo un punto de vista de maestro / detalle.

Solía mostrar los detalles del héroe por sí solo, antes de cortar esa parte de la plantilla. Ahora delegará en el `HeroDetailComponent` .

Los dos componentes tendrán una relación padre / hijo. El padre `HeroesComponent` controlará al niño `HeroDetailComponent` enviándole un nuevo héroe para que lo muestre cada vez que el usuario seleccione un héroe de la lista.

No cambiarás la `HeroesComponent` *clase* pero cambiarás su *plantilla* .

Actualizar la plantilla `HeroesComponent`

El `HeroDetailComponent` selector es `'app-hero-detail'` . Agrega un `<app-hero-detail>` elemento cerca de la parte inferior de la `HeroesComponent` plantilla, donde solía estar la vista detallada del héroe.

Enlazar la propiedad `HeroesComponent.selectedHero` del elemento `hero` como esta.

heroes.component.html (enlace HeroDetail)

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

`[hero]="selectedHero"` Es un enlace de propiedad angular .

Es un enlace de datos *unidireccional* de la `selectedHero` propiedad de la `HeroesComponent` a la `hero` propiedad del elemento de destino, que se asigna a la `hero` propiedad de la `HeroDetailComponent` .

Ahora cuando el usuario hace clic en un héroe en la lista, los `selectedHero` cambios. Cuando los `selectedHero` cambios, el *enlace de propiedades se* actualiza `hero` y se `HeroDetailComponent` muestra el nuevo héroe.

La `HeroesComponent` plantilla revisada debe tener este aspecto:

heroes.component.html

```
<h2>My Heroes</h2>

<ul class="heroes">
  <li *ngFor="let hero of heroes"
    [class.selected]="hero === selectedHero"
    (click)="onSelect(hero)">
    <span class="badge">{{hero.id}}</span> {{hero.name}}
  </li>
</ul>

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

El navegador se actualiza y la aplicación vuelve a funcionar como antes.

¿Qué cambió?

Como antes , cada vez que un usuario hace clic en el nombre de un héroe, el detalle del héroe aparece debajo de la lista de héroes. Ahora el `HeroDetailComponent` está presentando esos detalles en lugar del `HeroesComponent` .

Refactorizar el original `HeroesComponent` en dos componentes produce beneficios, tanto ahora como en el futuro:

1. Usted simplificó la `HeroesComponent` reducción de sus responsabilidades.
2. Puedes convertirlo `HeroDetailComponent` en un editor de héroe rico sin tocar al padre `HeroesComponent` .
3. Puedes evolucionar `HeroesComponent` sin tocar la vista detallada del héroe.
4. Puede reutilizar el `HeroDetailComponent` en la plantilla de algún componente futuro.

Resumen

- Usted creó un separado, reutilizable `HeroDetailComponent` .

- Usaste un enlace de propiedad para darle al padre `HeroesComponent` control sobre el hijo `HeroDetailComponent` .
- `@Input` Usaste el decorador para hacer que la `hero` propiedad esté disponible para ser atada por el externo `HeroesComponent` .

Servicios

El Tour de los Héroes `HeroesComponent` actualmente está obteniendo y mostrando datos falsos.

Después de la refactorización en este tutorial, `HeroesComponent` será magro y se centrará en apoyar la vista. También será más fácil realizar una prueba unitaria con un servicio simulado.

¿Por qué servicios?

Los componentes no deben recuperar o guardar datos directamente y, ciertamente, no deben presentar datos falsos a sabiendas. Deben centrarse en presentar datos y delegar el acceso de datos a un servicio.

En este tutorial, creará una `HeroService` que todas las clases de aplicaciones pueden usar para obtener héroes. En lugar de crear ese servicio con `new` , confiará en la *inyección de dependencia* angular para inyectarlo en el `HeroesComponent` constructor.

Los servicios son una excelente manera de compartir información entre clases que *no se conocen entre sí* . Crearás una `MessageService` y la inyectarás en dos lugares:

1. en el `HeroService` que utiliza el servicio para enviar un mensaje.
2. en el `MessagesComponent` que se muestra ese mensaje.

Crea el *HeroService*

Usando el CLI angular, cree un servicio llamado `hero` .

```
ng generate service hero
```

El comando genera una `HeroService` clase de esqueleto en `src/app/hero.service.ts` La `HeroService` clase debe verse como el siguiente ejemplo.

src / app / hero.service.ts (nuevo servicio)

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

@Injectable({
  providedIn: 'root',
})
export class HeroService {

  constructor() { }

}
```

@Injectable() servicios

Observe que el nuevo servicio importa el `Injectable` símbolo Angular y anota la clase con el decorador. Esto marca la clase como una que participa en el *sistema de inyección de dependencia* . La clase va a proporcionar un servicio inyectable y también puede tener sus propias dependencias inyectadas. Aún no tiene dependencias, pero las tendrá pronto .`@Injectable()` `HeroService`

El decorador acepta un objeto de metadatos para el servicio, de la misma forma que lo hizo el decorador para sus clases de componentes.`@Injectable()` `@Component()`

Obtener datos de héroe

El `HeroService` podían obtener los datos desde cualquier lugar héroe-un servicio web, almacenamiento local, o una fuente de datos simulada.

Eliminar el acceso a los datos de los componentes significa que puede cambiar de opinión acerca de la implementación en cualquier momento, sin tocar ningún componente. No saben cómo funciona el servicio.

La implementación en *este* tutorial continuará entregando *héroes simulados* .

Importar el `Hero` y `HEROES` .

```
import { Hero } from './hero';
import { HEROES } from './mock-heroes';
```

Agrega un `getHeroes` método para devolver los *héroes simulados* .

```
getHeroes(): Hero[] {
  return HEROES;
}
```

Proporcionar la `HeroService`

Debe ponerlo a `HeroService` disposición del sistema de inyección de dependencia antes de que Angular pueda *inyectarlo* en el `HeroesComponent` , como lo hará a continuación . Usted hace esto mediante el registro de un *proveedor* . Un proveedor es algo que puede crear o entregar un servicio; en este caso, crea una instancia de la `HeroService` clase para proporcionar el servicio.

Ahora, debe asegurarse de que `HeroService` esté registrado como el proveedor de este servicio. Lo está registrando con un *inyector* , que es el objeto responsable de elegir e inyectar al proveedor donde sea necesario.

De forma predeterminada, el comando Angular CLI `ng generate service` registra un proveedor con el *inyector raíz* para su servicio al incluir metadatos del proveedor en el decorador.`@Injectable`

Si observa la declaración justo antes de la definición de la clase, puede ver que el valor de los metadatos es 'raíz': `@Injectable()` `HeroService` `providedIn`

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

Cuando proporciona el servicio en el nivel raíz, Angular crea una instancia única y compartida de `HeroService` e inserta en cualquier clase que lo solicite. El registro del proveedor en los metadatos también le permite a Angular optimizar una aplicación eliminando el servicio si resulta que no se utiliza después de todo.`@Injectable`

> Para obtener más información sobre los proveedores, consulte la sección Proveedores . Para obtener más información sobre los inyectores, consulte la guía de inyección de dependencia .

El `HeroService` está ahora listo para conectar a la `HeroesComponent` .

> Este es un ejemplo de código interino que le permitirá proporcionar y utilizar el `HeroService` . En este punto, el código será diferente al `HeroService` de la "revisión final del código" .

Actualizar `HeroesComponent`

Abra el `HeroesComponent` archivo de clase.

Elimina la `HEROES` importación, porque ya no la necesitarás. Importar el `HeroService` lugar.

src / app / heroes / heroes.component.ts (importar HeroService)

```
import { HeroService } from '../hero.service';
```

Reemplace la definición de la `heroes` propiedad con una declaración simple.

```
heroes: Hero[];
```

Inyectar el `HeroService`

Agregue un `heroService` parámetro privado de tipo `HeroService` al constructor.

```
constructor(private heroService: HeroService) { }
```

El parámetro define simultáneamente una `heroService` propiedad privada y la identifica como un `HeroService` sitio de inyección.

Cuando Angular crea a `HeroesComponent` , el sistema de inyección de dependencias establece el `heroService` parámetro en la instancia de singleton de `HeroService` .

Añadir *getHeroes ()*

Crea una función para recuperar los héroes del servicio.

```
getHeroes(): void {
  this.heroes = this.heroService.getHeroes();
}
```

Llamalo en `ngOnInit`

Si bien puede llamar `getHeroes()` al constructor, esa no es la mejor práctica.

Reserve el constructor para una inicialización simple, como los parámetros del constructor de cableado a las propiedades. El constructor no debe *hacer nada* . Ciertamente, no debería llamar a una función que realiza solicitudes HTTP a un servidor remoto como lo haría un servicio de datos *real* .

En su lugar, llame `getHeroes()` dentro del *gancho del ciclo de vida de ngOnInit* y deje que Angular *realice la* llamada `ngOnInit` en el momento adecuado *después de crear* una `HeroesComponent` instancia.

```
ngOnInit() {
  this.getHeroes();
}
```

Verlo correr

Después de que el navegador se actualice, la aplicación debería ejecutarse como antes, mostrando una lista de héroes y una vista detallada del héroe al hacer clic en el nombre de un héroe.

Datos observables

El `HeroService.getHeroes()` método tiene una *firma síncrona* , lo que implica que `HeroService` puede obtener héroes de forma síncrona. El `HeroesComponent` consume el `getHeroes()` resultado como si los héroes pudieran ser buscados sincrónicamente.

```
this.heroes = this.heroService.getHeroes();
```

Esto no funcionará en una aplicación real. Te estás saliendo con la tuya ahora porque el servicio actualmente devuelve *simulacros de héroes* . Pero pronto la aplicación buscará héroes desde un servidor remoto, que es una operación inherentemente *asíncrona* .

El `HeroService` debe esperar a que el servidor responda, `getHeroes()` no se puede volver inmediatamente con los datos de héroe, y el navegador no bloqueará mientras que los espera el servicio.

`HeroService.getHeroes()` Debe tener una *firma asíncrona* de algún tipo.

Puede tomar una devolución de llamada. Podría volver a `Promise` . Podría devolver un `Observable` .

En este tutorial, `HeroService.getHeroes()` devolverá una `Observable` parte porque eventualmente usará el `HttpClient.get` método Angular para buscar a los héroes y `HttpClient.get()` devolverá una`Observable` .

Observable *HeroService*

`Observable` Es una de las clases clave en la biblioteca RxJS .

En un tutorial posterior sobre HTTP , aprenderá que los `HttpClient` métodos de Angular devuelven RxJS `Observable` s. En este tutorial, simulará obtener datos del servidor con la `of()` función RxJS .

Abra el `HeroService` archivo e importe los símbolos `Observable` y `of` desde RxJS.

src / app / hero.service.ts (Importaciones observables)

```
import { Observable, of } from 'rxjs';
```

Reemplace el `getHeroes` método con este.

```
getHeroes(): Observable<Hero[]> {
  return of(HEROES);
}
```

`of(HEROES)` devuelve una `Observable<Hero[]>` que emite *un solo valor* , la matriz de héroes simulados.

> En el tutorial de HTTP , llamarás, `HttpClient.get<Hero[]>()` que también devuelve un valor `Observable<Hero[]>` que emite *un solo valor* , una serie de héroes del cuerpo de la respuesta HTTP.

Suscríbete en *HeroesComponent*

El `HeroService.getHeroes` método utilizado para devolver un `Hero[]` . Ahora vuelve un `Observable<Hero[]>` .

Tendrás que adaptarte a esa diferencia en `HeroesComponent` .

Encuentre el `getHeroes` método y reemplácelo con el siguiente código (se muestra lado a lado con la versión anterior para comparación)

heroes.component.ts (observable) *heroes.component.ts (Original)*

```
getHeroes(): void {
  this.heroService.getHeroes()
```

```
        .subscribe(heroes => this.heroes = heroes);
  }
```

`Observable.subscribe()` Es la diferencia crítica.

La versión anterior asigna una serie de héroes a la `heroes` propiedad del componente . La asignación se realiza de forma *sincrónica* , como si el servidor pudiera devolver héroes al instante o el navegador pudiera congelar la interfaz de usuario mientras esperaba la respuesta del servidor.

Eso *no funcionará* cuando en `HeroService` realidad está realizando solicitudes de un servidor remoto.

La nueva versión espera a `Observable` que emita la serie de héroes, lo que podría suceder ahora o dentro de unos minutos. Luego `subscribe` pasa la matriz emitida a la devolución de llamada, que establece la `heroes` propiedad del componente .

Este enfoque asíncrono *funcionará* cuando los `HeroService` héroes solicitados desde el servidor.

Mostrar mensajes

En esta sección podrás

- agregue una `MessagesComponent` que muestre los mensajes de la aplicación en la parte inferior de la pantalla.
- crear un inyectable, en toda la aplicación `MessageService` para enviar mensajes a mostrar
- inyectar `MessageService` en el`HeroService`
- mostrar un mensaje cuando se `HeroService` obtiene héroes con éxito.

Crear *MessagesComponent*

Utilice el CLI para crear el `MessagesComponent` .

```
ng generate component messages
```

La CLI crea los archivos de componentes en la carpeta y declarar en .`src/app/messages` `MessagesComponent` `AppModule`

Modificar la `AppComponent` plantilla para visualizar el generado.`MessagesComponent`

/src/app/app.component.html

```
<h1>{{title}}</h1>
<app-heroes></app-heroes>
<app-messages></app-messages>
```

Debería ver el párrafo predeterminado `MessagesComponent` en la parte inferior de la página.

Crear el servicio de *mensajes*

Use el CLI para crear el `MessageService` in `src/app` .

```
ng generate service message
```

Abra `MessageService` y reemplace su contenido con lo siguiente.

/src/app/message.service.ts

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

@Injectable({
  providedIn: 'root',
})
export class MessageService {
  messages: string[] = [];

  add(message: string) {
    this.messages.push(message);
  }

  clear() {
    this.messages = [];
  }
}
```

El servicio expone su caché de `messages` y dos métodos: uno a `add()` un mensaje al caché y otro al `clear()` caché.

Inyectarlo en el `HeroService`

Vuelva a abrir el `HeroService` e importar el `MessageService` .

/src/app/hero.service.ts (importar MessageService)

```
import { MessageService } from './message.service';
```

Modificar el constructor con un parámetro que declara una `messageService` propiedad privada . Angular inyectará el singleton `MessageService` en esa propiedad cuando cree el `HeroService` .

```
constructor(private messageService: MessageService) { }
```

> Este es un escenario típico de " *servicio en servicio* ": se inyecta `MessageService` en el `HeroService` que se inyecta en el `HeroesComponent` .

Enviar un mensaje desde `HeroService`

Modifica el `getHeroes` método para enviar un mensaje cuando los héroes son buscados.

```
getHeroes(): Observable<Hero[]> {
  // TODO: send the message _after_ fetching the heroes
  this.messageService.add('HeroService: fetched heroes');
  return of(HEROES);
}
```

Mostrar el mensaje de `HeroService`

El `MessagesComponent` debe mostrar todos los mensajes, incluyendo el mensaje enviado por el `HeroService` cuando se obtiene héroes.

Abra `MessagesComponent` e importe el `MessageService` .

/src/app/messages/messages.component.ts (importar MessageService)

```
import { MessageService } from '../message.service';
```

Modificar el constructor con un parámetro que declara una propiedad **pública** `messageService` . Angular inyectará el singleton `MessageService` en esa propiedad cuando cree el `MessagesComponent` .

```
constructor(public messageService: MessageService) {}
```

La `messageService` propiedad **debe ser pública** porque está a punto de unirse a ella en la plantilla.

> Angular solo se enlaza a *las* propiedades de los componentes *públicos* .

Enlace al servicio de *mensajes*

Reemplace la `MessagesComponent` plantilla generada por CLI con lo siguiente.

src / app / messages / messages.component.html

```
<div *ngIf="messageService.messages.length">

  <h2>Messages</h2>
  <button class="clear"
          (click)="messageService.clear()">clear</button>
  <div *ngFor='let message of messageService.messages'> {{message}} </div>

</div>
```

Esta plantilla se enlaza directamente a los componentes `messageService` .

- La única muestra el área de mensajes si hay mensajes para mostrar.`*ngIf`
- An presenta la lista de mensajes en elementos repetidos .`*ngFor` `<div>`
- Un enlace de evento Angular vincula el evento de clic del botón a `MessageService.clear()` .

Los mensajes se verán mejor cuando agregue los estilos de CSS privados `messages.component.css` que figuran en una de las pestañas de "revisión final del código" a continuación.

El navegador se actualiza y la página muestra la lista de héroes. Desplácese hasta la parte inferior para ver el mensaje `HeroService` en el área de mensajes. Haga clic en el botón "Borrar" y el área de mensaje desaparecerá.

Resumen

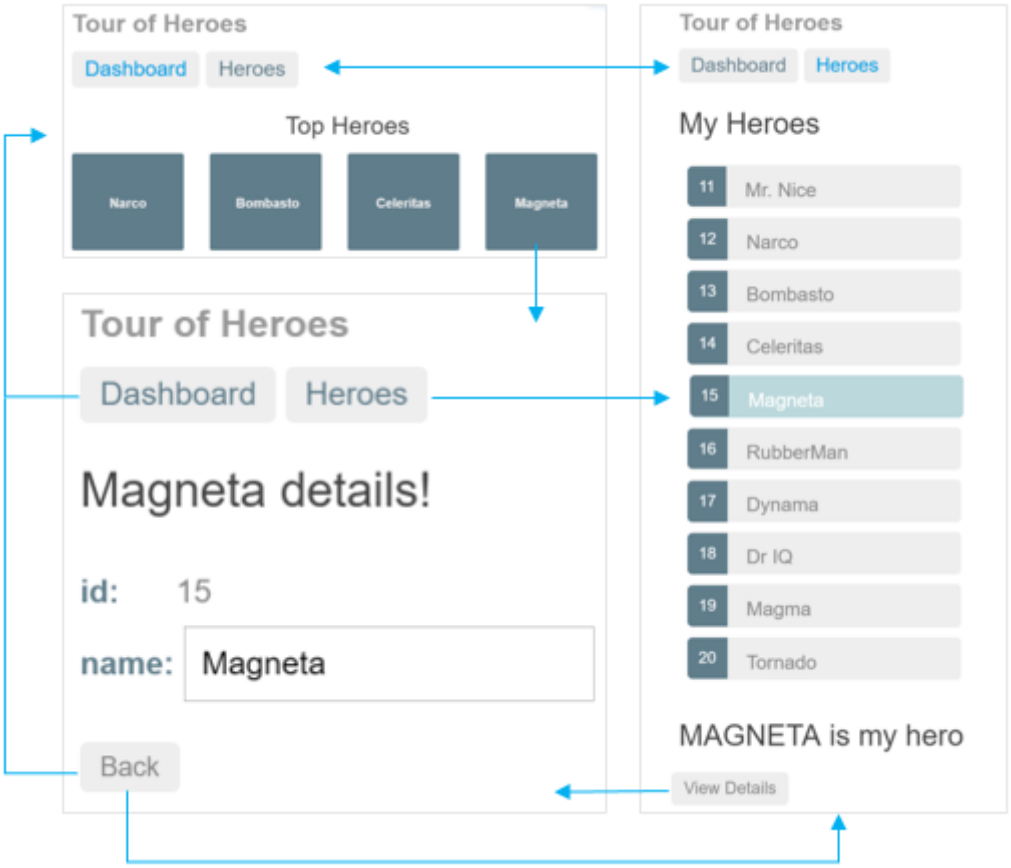
- Usted refactorizó el acceso de datos a la `HeroService` clase.
- Usted registró `HeroService` como *proveedor* de su servicio en el nivel raíz para que pueda ser inyectado en cualquier parte de la aplicación.
- Usó la inyección de dependencia angular para inyectarla en un componente.
- Usted le dio al método de `HeroService` *obtención de datos* una firma asíncrona.
- Usted descubrió `Observable` y la biblioteca RxJS *Observable* .
- Usaste RxJS `of()` para devolver un observable de simulacros de héroes (`Observable<Hero[]>`).
- El `ngOnInit` gancho del ciclo de vida del componente llama al `HeroService` método, no al constructor.
- Usted creó una `MessageService` comunicación entre clases de forma holgada.
- El `HeroService` inyectado en un componente se crea con otro servicio inyectado, `MessageService` .

Enrutamiento

Hay nuevos requisitos para la aplicación Tour of Heroes:

- Añadir una vista de *panel de* control.
- Añade la capacidad de navegar entre las vistas *Héroes* y *Tablero* .
- Cuando los usuarios hagan clic en el nombre de un héroe en cualquiera de las vistas, navegue a una vista detallada del héroe seleccionado.
- Cuando los usuarios hacen clic en un *enlace profundo* en un correo electrónico, abra la vista detallada para un héroe en particular.

Cuando hayas terminado, los usuarios podrán navegar por la aplicación de esta manera:



Añade el `AppRoutingModule`

Una de las mejores prácticas de Angular es cargar y configurar el enrutador en un módulo separado de nivel superior dedicado al enrutamiento e importado por la raíz `AppModule` .

Por convención, el nombre de la clase del módulo es `AppRoutingModule` y pertenece `app-routing.module.ts` a la `src/app` carpeta.

Usa el CLI para generarlo.

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

> `--flat` pone el archivo en `src/app` lugar de su propia carpeta.
> `--module=app` le dice a la CLI que lo registre en la `imports` matriz de `AppModule` .

El archivo generado se ve así:

src / app / app-routing.module.ts (generado)

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

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class AppRoutingModule { }
```

Por lo general, no declara los componentes en un módulo de enrutamiento, por lo que también puede eliminar la matriz y eliminar las referencias.`@NgModule.declarations` `CommonModule`

Configurará el enrutador `Routes` en el `RouterModule` para importar esos dos símbolos de la `@angular/router` biblioteca.

Añadir una matriz con en ella. La exportación hace que las directivas de enrutador estén disponibles para su uso en los componentes que las necesitarán.`@NgModule.exports` `RouterModule` `RouterModule` `AppModule`

`AppRoutingModule` se ve así ahora

src / app / app-routing.module.ts (v1)

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

@NgModule({
  exports: [ RouterModule ]
})
export class AppRoutingModule {}
```

Agregar rutas

Las rutas le indican al enrutador qué vista mostrar cuando un usuario hace clic en un enlace o pega una URL en la barra de direcciones del navegador.

Un Angular típico `Route` tiene dos propiedades:

1. `path` : una cadena que coincide con la URL en la barra de direcciones del navegador.
2. `component` : el componente que el enrutador debe crear al navegar por esta ruta.

Tiene la intención de navegar a `HeroesComponent` cuando la URL es algo así `localhost:4200/heroes` .

Importa el `HeroesComponent` para que puedas referenciarlo en un archivo `Route` . Luego, defina una matriz de rutas con un solo `route` componente para ese componente.

```
import { HeroesComponent }      from './heroes/heroes.component';

const routes: Routes = [
  { path: 'heroes', component: HeroesComponent }
];
```

Una vez que hayas terminado de configurar, el enrutador hará coincidir esa URL `path: 'heroes'` y mostrará el `HeroesComponent` .

RouterModule.forRoot ()

Primero debe inicializar el enrutador y comenzar a escuchar los cambios de ubicación del navegador.

Agregue `RouterModule` a la matriz y configúrela con el en un solo paso llamando *dentro de* la matriz, como esto:`@NgModule.imports` `routes` `RouterModule.forRoot()` `imports`

```
imports: [ RouterModule.forRoot(routes) ],
```

> Se llama al método `forRoot()` porque configura el enrutador en el nivel raíz de la aplicación. El `forRoot()` método proporciona los proveedores de servicios y las directivas necesarias para el enrutamiento y realiza la navegación inicial según la URL del navegador actual.

Añadir *RouterOutlet*

Abra la `AppComponent` plantilla reemplace el `<app-heroes>` elemento con un elemento.`<router-outlet>`

src / app / app.component.html (enrutador-salida)

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

Se eliminó `<app-heroes>` porque solo se mostrará `HeroesComponent` cuando el usuario navegue hacia él.

Le indica al enrutador dónde mostrar las vistas enrutadas.`<router-outlet>`

> El `RouterOutlet` es una de las directivas de router que ahora están disponibles para las `AppComponent` PORQUE `AppModule` importaciones `AppRoutingModule` que exportaron `RouterModule` .

Intentalo

Todavía debe estar ejecutando con este comando CLI.

```
ng serve
```

El navegador debe actualizar y mostrar el título de la aplicación, pero no la lista de héroes.

Mira la barra de direcciones del navegador. La URL termina en `/` . El camino a la ruta `HeroesComponent` es `/heroes` .

Adjuntar `/heroes` a la URL en la barra de direcciones del navegador. Deberías ver la vista de maestro / detalle de los héroes familiares.

Añadir un enlace de navegación ( )

Los usuarios no deberían tener que pegar una URL de ruta en la barra de direcciones. Deben poder hacer clic en un enlace para navegar.

Agregue un `<nav>` elemento y, dentro de eso, un elemento de anclaje que, al hacer clic, active la navegación hacia `HeroesComponent` . La `AppComponent` plantilla revisada se ve así:

src / app / app.component.html (heroes RouterLink)

```
<h1>{{title}}</h1>
<nav>
  <a routerLink="/heroes">Heroes</a>
</nav>
<router-outlet></router-outlet>
<app-messages></app-messages>
```

Un `routerLink` atributo se establece en `"/heroes"` , la cadena con la que el enrutador coincide con la ruta `HeroesComponent` . El `routerLink` es el selector para la `RouterLink` directiva que convierte los clics del usuario en navegaciones del enrutador. Es otra de las directivas publicas en el `RouterModule` .

El navegador actualiza y muestra el título de la aplicación y el enlace de los héroes, pero no la lista de héroes.

Haga clic en el enlace. La barra de direcciones se actualiza `/heroes` y aparece la lista de héroes.

> Haga que este y los futuros enlaces de navegación se vean mejor agregando estilos de CSS privados `app.component.css` como se indica en la revisión final del código a continuación.

Añadir una vista de panel

El enrutamiento tiene más sentido cuando hay varias vistas. Hasta el momento sólo hay vista de los héroes.

Agregue un `DashboardComponent` usando el CLI:

```
ng generate component dashboard
```

El CLI genera los archivos para `DashboardComponent` y lo declara en `AppModule` .

Reemplace el contenido del archivo predeterminado en estos tres archivos de la siguiente manera y luego vuelva para una pequeña discusión:

src / app / dashboard / dashboard.component.html src / app / dashboard / dashboard.c

```
<h3>Top Heroes</h3>
<div class="grid grid-pad">
  <a *ngFor="let hero of heroes" class="col-1-4">
    <div class="module hero">
      <h4>{{hero.name}}</h4>
    </div>
  </a>
</div>
```

La *plantilla* presenta una cuadrícula de enlaces de nombre de héroe.

- El repetidor crea tantos enlaces como están en la matriz del componente .`*ngFor` `heroes`
- Los enlaces están diseñados como bloques de colores por el `dashboard.component.css` .
- Los enlaces no van a ninguna parte todavía, pero lo harán en breve .

La *clase* es similar a la `HeroesComponent` clase.

- Define una `heroes` propiedad de matriz.
- El constructor espera que Angular inyecte el `HeroService` en una `heroService` propiedad privada .
- El `ngOnInit()` gancho del ciclo de vida llama `getHeroes` .

Esto `getHeroes` devuelve la lista dividida de héroes en las posiciones 1 y 5, devolviendo solo cuatro de los mejores héroes (2º, 3º, 4º y 5º).

```
getHeroes(): void {
  this.heroService.getHeroes()
    .subscribe(heroes => this.heroes = heroes.slice(1, 5));
}
```

Agregar la ruta del tablero

Para navegar hacia el panel de control, el enrutador necesita una ruta adecuada.

Importar el `DashboardComponent` en el `AppRoutingModule` .

src / app / app-routing.module.ts (importar DashboardComponent)

```
import { DashboardComponent }   from './dashboard/dashboard.component';
```

Agregue una ruta a la `AppRoutingModule.routes` matriz que coincida con una ruta al `DashboardComponent` .

```
{ path: 'dashboard', component: DashboardComponent },
```

Añadir una ruta por defecto

Cuando se inicia la aplicación, la barra de direcciones del navegador apunta a la raíz del sitio web. Eso no coincide con ninguna ruta existente, por lo que el enrutador no navega a ningún lado. El espacio debajo de está en blanco.`<router-outlet>`

Para hacer que la aplicación navegue automáticamente al panel de control, agregue la siguiente ruta a la `AppRoutingModule.Routes` matriz.

```
{ path: '', redirectTo: '/dashboard', pathMatch: 'full' },
```

Esta ruta redirige una URL que coincide completamente con la ruta vacía a la ruta cuya ruta es `'/dashboard'` .

Una vez que se actualiza el navegador, el enrutador carga `DashboardComponent` y la barra de direcciones del navegador muestra la `/dashboard` URL.

Añadir enlace del panel de control a la shell

El usuario debe poder navegar hacia adelante y hacia atrás entre `DashboardComponent` y `HeroesComponent` haciendo clic en los enlaces en el área de navegación cerca de la parte superior de la página.

Agrega un enlace de navegación del panel a la `AppComponent` plantilla de shell, justo encima del enlace *Héroes* .

src / app / app.component.html

```
<h1>{{title}}</h1>
<nav>
  <a routerLink="/dashboard">Dashboard</a>
  <a routerLink="/heroes">Heroes</a>
</nav>
<router-outlet></router-outlet>
<app-messages></app-messages>
```

Después de que el navegador se actualice, puede navegar libremente entre las dos vistas haciendo clic en los enlaces.

Navegando a los detalles del héroe

La `HeroDetailsComponent` muestra los detalles de un héroe seleccionado. Por el momento `HeroDetailsComponent` solo es visible en la parte inferior de la`HeroesComponent`

El usuario debe poder acceder a estos detalles de tres maneras.

1. Al hacer clic en un héroe en el tablero de instrumentos.
2. Al hacer clic en un héroe en la lista de héroes.
3. Al pegar una URL de "vínculo profundo" en la barra de direcciones del navegador que identifica al héroe que se mostrará.

En esta sección, habilitará la navegación hacia `HeroDetailsComponent` y la liberará de `HeroesComponent` .

Eliminar *detalles de héroe* de`HeroesComponent`

Cuando el usuario hace clic en un elemento de héroe en el `HeroesComponent` , la aplicación debe navegar a `HeroDetailComponent` , reemplazando la vista de la lista de héroes con la vista de detalles del héroe. La vista de lista de héroes ya no debería mostrar detalles de héroe como lo hace ahora.

Abra la `HeroesComponent` plantilla (`heroes/heroes.component.html`) y elimine el `<app-hero-detail>` elemento de la parte inferior.

Hacer clic en un objeto de héroe ahora no hace nada. Lo arreglará poco después de habilitar el enrutamiento a `HeroDetailComponent` .

Añadir una ruta de *detalle de héroe*

Una URL como `~/detail/11` sería una buena URL para navegar a la vista de *Detalle* del héroe del héroe cuyo `id` es `11` .

Abrir `AppRoutingModule` e importar `HeroDetailComponent` .

src / app / app-routing.module.ts (importar HeroDetailComponent)

```
import { HeroDetailComponent }  from './hero-detail/hero-detail.component';
```

Luego, agregue una ruta *parametrizada* a la `AppRoutingModule.routes` matriz que coincida con el patrón de ruta con la vista de *detalles* del *héroe* .

```
{ path: 'detail/:id', component: HeroDetailComponent },
```

Los dos puntos (:) en el `path` indica que `:id` es un marcador de posición para un héroe específico `id` .

En este punto, todas las rutas de aplicación están en su lugar.

src / app / app-routing.module.ts (todas las rutas)

```
const routes: Routes = [
  { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
  { path: 'dashboard', component: DashboardComponent },
  { path: 'detail/:id', component: HeroDetailComponent },
  { path: 'heroes', component: HeroesComponent }
];
```

`DashboardComponent` enlaces de héroe

Los `DashboardComponent` enlaces de héroe no hacen nada por el momento.

Ahora que el enrutador tiene una ruta a `HeroDetailComponent` , corrija los enlaces del héroe del tablero para navegar a través de la ruta del tablero *parametrizado* .

src / app / dashboard / dashboard.component.html (enlaces de héroe)

```
<a *ngFor="let hero of heroes" class="col-1-4"
    routerLink="/detail/{{hero.id}}">
  <div class="module hero">
    <h4>{{hero.name}}</h4>
  </div>
</a>
```

Está utilizando un enlace de interpolación angular dentro del repetidor para insertar la iteración actual en cada uno .`*ngFor` `hero.id` `routerLink`

`HeroesComponent` enlaces de héroe

Los elementos de héroe en `HeroesComponent` son `<li>` elementos cuyos eventos de clic están vinculados al `onSelect()` método del componente .

src / app / heroes / heroes.component.html (lista con onSelect)

```
<ul class="heroes">
  <li *ngFor="let hero of heroes"
    [class.selected]="hero === selectedHero"
    (click)="onSelect(hero)">
    <span class="badge">{{hero.id}}</span> {{hero.name}}
  </li>
</ul>
```

Desplace la `<li>` parte de atrás a solo su , envuelva la insignia y el nombre en un elemento de ancla ( ) y agregue un atributo al ancla que sea el mismo que en la plantilla del panel de control`*ngFor` `<a>` `routerLink`

src / app / heroes / heroes.component.html (lista con enlaces)

```
<ul class="heroes">
  <li *ngFor="let hero of heroes">
    <a routerLink="/detail/{{hero.id}}">
      <span class="badge">{{hero.id}}</span> {{hero.name}}
    </a>
  </li>
</ul>
```

Tendrá que arreglar la hoja de estilo privada (`heroes.component.css`) para que la lista se vea como lo hizo antes. Los estilos revisados se encuentran en la revisión final del código al final de esta guía.

Eliminar código muerto (opcional)

Mientras la `HeroesComponent` clase todavía funciona, el `onSelect()` método y la `selectedHero` propiedad ya no se utilizan.

Es bueno ordenarlo y te lo agradecerás más tarde. Aquí está la clase después de podar el código muerto.

src / app / heroes / heroes.component.ts (limpiado)

```
export class HeroesComponent implements OnInit {
  heroes: Hero[];

  constructor(private heroService: HeroService) { }

  ngOnInit() {
    this.getHeroes();
  }

  getHeroes(): void {
    this.heroService.getHeroes()
    .subscribe(heroes => this.heroes = heroes);
  }
}
```

HeroDetailComponent *enrutable*

Anteriormente, el padre `HeroesComponent` establecía la `HeroDetailComponent.hero` propiedad y `HeroDetailComponent` mostraba al héroe.

`HeroesComponent` ya no hace eso Ahora el enrutador crea el `HeroDetailComponent` en respuesta a una URL como `~/detail/11` .

El `HeroDetailComponent` necesita una nueva forma de obtener el *héroe para mostrar* .

- Consigue la ruta que lo creó,
- Extraer `id` de la ruta.
- Adquirir el héroe con eso `id` desde el servidor a través de la`HeroService`

Agregue las siguientes importaciones:

src / app / hero-detail / hero-detail.component.ts

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

import { HeroService }  from '../hero.service';
```

Inyecte el `ActivatedRoute` , `HeroService` y los `Location` servicios en el constructor, guardando sus valores en campos privados:

```
constructor(
  private route: ActivatedRoute,
  private heroService: HeroService,
  private location: Location
) {}
```

El `ActivatedRoute` guarda información sobre la ruta a esta instancia del `HeroDetailComponent` . Este componente está interesado en la bolsa de parámetros de la ruta extraída de la URL. El parámetro "*id*" es el `id` del héroe a mostrar.

La `HeroService` obtiene datos héroe desde el servidor remoto y este componente se utilizan para obtener el *héroe-a-pantalla* .

El `location` es un servicio angular para interactuar con el navegador. Lo usarás más tarde para volver a la vista que navegó aquí.

Extraer el parámetro de ruta de *ID*

En el `ngOnInit()` gancho del ciclo de vida llame `getHero()` y defínalo como sigue.

```
ngOnInit(): void {
  this.getHero();
}

getHero(): void {
  const id = +this.route.snapshot.paramMap.get('id');
  this.heroService.getHero(id)
    .subscribe(hero => this.hero = hero);
}
```

El `route.snapshot` es una imagen estática de la información de ruta poco después de que el componente fue creado.

El `paramMap` es un diccionario de valores de parámetros de ruta extraídos de la URL. La `"id"` clave devuelve la `id` del héroe a buscar.

Los parámetros de ruta son siempre cadenas. El operador de JavaScript (+) convierte la cadena en un número, que es lo que `id` debería ser un héroe .

El navegador se actualiza y la aplicación se bloquea con un error del compilador. `HeroService` no tiene un `getHero()` metodo Agrégalo ahora.

Añadir `HeroService.getHero()`

Abre `HeroService` y añade este `getHero()` método.

src / app / hero.service.ts (getHero)

```
getHero(id: number): Observable<Hero> {
  // TODO: send the message _after_ fetching the hero
  this.messageService.add(`HeroService: fetched hero id=${id}`);
  return of(HEROES.find(hero => hero.id === id));
}
```

> Tenga en cuenta los comillas invertidas (`) que definen un *literal de plantilla de* JavaScript para incrustar `id` .

Como `getHeroes()` , `getHero()` tiene una firma asíncrona. Devuelve un *héroe simulado* como `Observable` , usando la `of()` función RxJS .

Podrás volver a implementar `getHero()` como una `Http` solicitud real sin tener que cambiar el `HeroDetailComponent` que la llama.

Intentalo

El navegador se actualiza y la aplicación está funcionando de nuevo. Puedes hacer clic en un héroe en el tablero o en la lista de héroes y navegar a la vista detallada de ese héroe.

Si pega `localhost:4200/detail/11` en la barra de direcciones del navegador, el enrutador navega a la vista de detalles del héroe con `id: 11` "Mr. Nice".

Encuentra el camino de regreso

Al hacer clic en el botón de retroceso del navegador, puede volver a la lista de héroes o la vista del tablero de mandos, según lo que le haya enviado a la vista detallada.

Sería bueno tener un botón en la `HeroDetail` vista que pueda hacer eso.

Agregue un botón de *retroceso* en la parte inferior de la plantilla del componente y conéctelo al `goBack()` método del componente .

src / app / hero-detail / hero-detail.component.html (botón atrás)

```
<button (click)="goBack()">go back</button>
```

Agregue un `goBack()` *método* a la clase de componente que navegue hacia atrás un paso en la pila de historial del navegador usando el `Location` servicio que inyectó anteriormente .

src / app / hero-detail / hero-detail.component.ts (goBack)

```
goBack(): void {
  this.location.back();
}
```

Actualiza el navegador y comienza a hacer clic. Los usuarios pueden navegar por la aplicación, desde el panel de control hasta los detalles del héroe y viceversa, desde la lista de héroes hasta el mini detalle hasta los detalles del héroe y de nuevo a los héroes.

Has cumplido con todos los requisitos de navegación que impulsaron esta página.

Resumen

- Agregaste el enrutador angular para navegar entre los diferentes componentes.
- Se convirtió `AppComponent` en un shell de navegación con enlaces y a .`<a>` `<router-outlet>`
- Configuró el enrutador en un `AppRoutingModule`
- Definió rutas simples, una ruta de redirección y una ruta parametrizada.
- Se utilizó la `routerLink` directiva en elementos de anclaje.
- Usted reformuló una vista maestra / detalle estrechamente acoplada en una vista de detalles enrutada.
- Usaste los parámetros del enlace del enrutador para navegar a la vista detallada de un héroe seleccionado por el usuario.
- Compartiste los `HeroService` múltiples componentes.

HTTP

En este tutorial, agregará las siguientes funciones de persistencia de datos con la ayuda de Angular `HttpClient` .

- El `HeroService` obtiene datos de héroe con peticiones HTTP.
- Los usuarios pueden agregar, editar y eliminar héroes y guardar estos cambios a través de HTTP.
- Los usuarios pueden buscar héroes por nombre.

Cuando hayas terminado con esta página, la aplicación debería tener este aspecto ejemplo vivo / ejemplo de descarga.

Habilitar servicios HTTP

`HttpClient` Es el mecanismo de Angular para comunicarse con un servidor remoto a través de HTTP.

Para que esté `HttpClient` disponible en todas partes en la aplicación:

- abre la raiz `AppModule`
- importar el `HttpClientModule` símbolo desde`@angular/common/http`

src / app / app.module.ts (importación de cliente HTTP)

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

- agregarlo a la matriz`@NgModule.imports`

Simular un servidor de datos

Este ejemplo de tutorial *imita la* comunicación con un servidor de datos remoto utilizando el módulo de *API web en memoria* .

Después de instalar el módulo, la aplicación hará solicitudes y recibirá respuestas de las mismas `HttpClient` sin saber que la *API web en memoria* está interceptando esas solicitudes, aplicándolas a un almacén de datos en memoria y devolviendo respuestas simuladas.

Esta instalación es una gran conveniencia para el tutorial. No tendrás que configurar un servidor para conocer `HttpClient` .

También puede ser conveniente en las primeras etapas de su propio desarrollo de aplicaciones cuando la API web del servidor no está bien definida o aún no está implementada.

> **Importante:** el módulo de *API web en memoria* no tiene nada que ver con HTTP en Angular.
>
> Si solo estás *leyendo* este tutorial para aprender `HttpClient` , puedes omitir este paso. Si está programando *junto* con este tutorial, quédese aquí y agregue la *API web en memoria* ahora.

Instale el paquete de *API web en memoria* desde *npm*

```
npm install angular-in-memory-web-api --save
```

Importa el `HttpClientInMemoryWebApiModule` y la `InMemoryDataService` clase, que crearás en un momento.

src / app / app.module.ts (importaciones de la API web en memoria)

```
import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';
import { InMemoryDataService }  from './in-memory-data.service';
```

Agregue el `HttpClientInMemoryWebApiModule` a la matriz, *después de importar la* , mientras la configura con la .`@NgModule.imports` *`HttpClientModule`* `InMemoryDataService`

```
HttpClientModule,

// The HttpClientInMemoryWebApiModule module intercepts HTTP requests
// and returns simulated server responses.
// Remove it when a real server is ready to receive requests.
HttpClientInMemoryWebApiModule.forRoot(
  InMemoryDataService, { dataEncapsulation: false }
)
```

El `forRoot()` método de configuración toma una `InMemoryDataService` clase que prepara la base de datos en memoria.

La muestra *de Tour of Heroes* crea una clase de este tipo `src/app/in-memory-data.service.ts` que tiene el siguiente contenido:

src / app / in-memory-data.service.ts

```
import { InMemoryDbService } from 'angular-in-memory-web-api';
import { Hero } from './hero';

export class InMemoryDataService implements InMemoryDbService {
  createDb() {
    const heroes = [
      { id: 11, name: 'Mr. Nice' },
      { id: 12, name: 'Narco' },
      { id: 13, name: 'Bombasto' },
      { id: 14, name: 'Celeritas' },
      { id: 15, name: 'Magneta' },
      { id: 16, name: 'RubberMan' },
      { id: 17, name: 'Dynama' },
      { id: 18, name: 'Dr IQ' },
      { id: 19, name: 'Magma' },
      { id: 20, name: 'Tornado' }
    ];
    return {heroes};
  }

  // Overrides the genId method to ensure that a hero always has an id.
  // If the heroes array is empty,
  // the method below returns the initial number (11).
  // if the heroes array is not empty, the method below returns the highest
  // hero id + 1.
  genId(heroes: Hero[]): number {
    return heroes.length > 0 ? Math.max(...heroes.map(hero => hero.id)) + 1 : 11;
  }
}
```

Este archivo reemplaza `mock-heroes.ts` , que ahora es seguro eliminar.

Cuando su servidor esté listo, desconecte la *API web en memoria* y las solicitudes de la aplicación pasarán al servidor.

Ahora volvamos a la `HttpClient` historia.

Héroes y HTTP

Importe algunos símbolos HTTP que necesitará:

src / app / hero.service.ts (importar símbolos HTTP)

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

Inyectar `HttpClient` en el constructor en una propiedad privada llamada `http` .

```
constructor(
  private http: HttpClient,
  private messageService: MessageService) { }
```

Sigue inyectando el `MessageService` . Lo llamarás con tanta frecuencia que lo envolverás en un `log` método privado .

```
/** Log a HeroService message with the MessageService */
private log(message: string) {
  this.messageService.add(`HeroService: ${message}`);
}
```

Defina el `heroesUrl` formulario `:base/:collectionName` con la dirección del recurso de héroes en el servidor. Aquí `base` está el recurso al que se hacen las solicitudes, y `collectionName` es el objeto de datos de los héroes en `in-memory-data-service.ts` .

```
private heroesUrl = 'api/heroes';  // URL to web api
```

Consigue héroes con *HttpClient*

La corriente `HeroService.getHeroes()` usa la `of()` función RxJS para devolver una serie de héroes simulados como un `Observable<Hero[]>` .

src / app / hero.service.ts (getHeroes con RxJs 'de ()')

```
getHeroes(): Observable<Hero[]> {
  return of(HEROES);
}
```

Convierte ese método para usar `HttpClient`

```
/** GET heroes from the server */
getHeroes (): Observable<Hero[]> {
  return this.http.get<Hero[]>(this.heroesUrl)
}
```

Actualizar el navegador. Los datos del héroe deben cargarse correctamente desde el servidor simulado.

Ha cambiado `of` por `http.get` y la aplicación sigue funcionando sin ningún otro cambio porque ambas funciones devuelven un `Observable<Hero[]>` .

Los métodos HTTP devuelven un valor

Todos los `HttpClient` métodos devuelven un RxJS `Observable` de algo.

HTTP es un protocolo de solicitud / respuesta. Usted hace una solicitud, devuelve una sola respuesta.

En general, un observable *puede* devolver múltiples valores a lo largo del tiempo. Un observable desde `HttpClient` siempre emite un solo valor y luego se completa, nunca más emite.

Esta `HttpClient.get` llamada en particular devuelve un `Observable<Hero[]>` , literalmente, " *un observable de matrices de héroes* ". En la práctica, solo devolverá una única matriz de héroe.

HttpClient.get devuelve datos de respuesta

`HttpClient.get` devuelve el *cuerpo* de la respuesta como un objeto JSON sin tipo por defecto. La aplicación del especificador de tipo opcional,, `<Hero[]>` le da un objeto de resultado escrito.

La forma de los datos JSON está determinada por la API de datos del servidor. La API de datos *de Tour of Heroes* devuelve los datos de héroe como una matriz.

> Otras API pueden enterrar los datos que desee dentro de un objeto. Es posible que tenga que extraer esos datos procesando el `Observable` resultado con el `map` operador RxJS .
>
> Aunque no se explica aquí, hay un ejemplo `map` en el `getHeroNo404()` método incluido en el código fuente de ejemplo.

Manejo de errores

Las cosas van mal, especialmente cuando obtiene datos de un servidor remoto. El `HeroService.getHeroes()` método debería detectar errores y hacer algo apropiado.

Para detectar errores, **"canaliza" el** resultado **observable** a `http.get()` través de un `catchError()` operador RxJS .

Importe el `catchError` símbolo desde `rxjs/operators` , junto con algunos otros operadores que necesitará más adelante.

```
import { catchError, map, tap } from 'rxjs/operators';
```

Ahora extiende el resultado observable con el `.pipe()` método y dale un `catchError()` operador.

```
getHeroes (): Observable<Hero[]> {
  return this.http.get<Hero[]>(this.heroesUrl)
    .pipe(
      catchError(this.handleError('getHeroes', []))
    );
}
```

El `catchError()` operador intercepta un `Observable` **que falló** . Pasa el error un *controlador de errores* que puede hacer lo que quiera con el error.

El siguiente `handleError()` método informa el error y luego devuelve un resultado inocuo para que la aplicación siga funcionando.

manejarError

Lo siguiente `errorHandler()` será compartido por muchos `HeroService` métodos, por lo que se generaliza para satisfacer sus diferentes necesidades.

En lugar de manejar el error directamente, devuelve una función de *manejador de errores* `catchError` que se ha configurado con el nombre de la operación que falló y un valor de retorno seguro.

```
/**
 * Handle Http operation that failed.
 * Let the app continue.
 * @param operation - name of the operation that failed
 * @param result - optional value to return as the observable result
 */
private handleError<T> (operation = 'operation', result?: T) {
  return (error: any): Observable<T> => {

    // TODO: send the error to remote logging infrastructure
    console.error(error); // log to console instead

    // TODO: better job of transforming error for user consumption
    this.log(`${operation} failed: ${error.message}`);

    // Let the app keep running by returning an empty result.
    return of(result as T);
  };
}
```

Después de informar el error a la consola, el controlador crea un mensaje fácil de usar y devuelve un valor seguro a la aplicación para que pueda seguir trabajando.

Debido a que cada método de servicio devuelve un tipo diferente de `Observable` resultado, `errorHandler()` toma un parámetro de tipo para que pueda devolver el valor seguro como el tipo que la aplicación espera.

Toque en el *observable*

Los `HeroService` métodos **aprovecharán** el flujo de valores observables y enviarán un mensaje (vía `log()`) al área de mensajes en la parte inferior de la página.

Harán eso con el `tap` operador RxJS , que *observa* los valores observables, hace *algo* con esos valores y los pasa. La `tap` llamada de vuelta no toca los valores en sí mismos.

Aquí está la versión final de `getHeroes` la `tap` que registra la operación.

```
/** GET heroes from the server */
getHeroes (): Observable<Hero[]> {
  return this.http.get<Hero[]>(this.heroesUrl)
    .pipe(
      tap(heroes => this.log('fetched heroes')),
      catchError(this.handleError('getHeroes', []))
    );
}
```

Obtener héroe por id

La mayoría de las API web admiten una solicitud de *obtención por ID* en el formulario `:baseURL/:id` .

Aquí, la *URL base* es la `heroesURL` definida en la sección Héroes y HTTP (`api/heroes`) e *id* es el número del héroe que desea recuperar. Por ejemplo `api/heroes/11` ,.

Agrega un `HeroService.getHero()` método para hacer esa solicitud:

src / app / hero.service.ts

```
/** GET hero by id. Will 404 if id not found */
getHero(id: number): Observable<Hero> {
  const url = `${this.heroesUrl}/${id}`;
  return this.http.get<Hero>(url).pipe(
    tap(_ => this.log(`fetched hero id=${id}`)),
    catchError(this.handleError<Hero>(`getHero id=${id}`))
  );
}
```

Hay tres diferencias significativas respecto a `getHeroes()` .

- construye una URL de solicitud con el id del héroe deseado.
- el servidor debe responder con un solo héroe en lugar de una serie de héroes.
- por lo tanto, `getHero` devuelve un `Observable<Hero>` (" *un objeto observable de héroes* ") en lugar de un observable de *matrices* de héroes .

Héroes de actualización

Editar el nombre de un héroe en la vista de *detalles* del *héroe* . A medida que escribes, el nombre del héroe actualiza el encabezado en la parte superior de la página. Pero cuando haces clic en el botón "volver", los

cambios se pierden.

Si desea que los cambios persistan, debe escribirlos de nuevo en el servidor.

Al final de la plantilla de detalles del héroe, agregue un botón de guardar con un `click` enlace de evento que invoque un nuevo método de componente llamado `save()` .

src / app / hero-detail / hero-detail.component.html (guardar)

```
<button (click)="save()">save</button>
```

Agregue el siguiente `save()` método, que persiste los cambios de nombre de héroe utilizando el `updateHero()` método de servicio de héroe y luego regresa a la vista anterior.

src / app / hero-detail / hero-detail.component.ts (guardar)

```
save(): void {
   this.heroService.updateHero(this.hero)
     .subscribe(() => this.goBack());
 }
```

Añadir *HeroService.updateHero ()*

La estructura general del `updateHero()` método es similar a la de `getHeroes()` , pero se utiliza `http.put()` para conservar el héroe cambiado en el servidor.

src / app / hero.service.ts (actualización)

```
/** PUT: update the hero on the server */
updateHero (hero: Hero): Observable<any> {
  return this.http.put(this.heroesUrl, hero, httpOptions).pipe(
    tap(_ => this.log(`updated hero id=${hero.id}`)),
    catchError(this.handleError<any>('updateHero'))
  );
}
```

El `HttpClient.put()` método toma tres parámetros.

- la URL
- Los datos a actualizar (el héroe modificado en este caso).
- opciones

La URL no ha cambiado. La API web de los héroes sabe qué héroe actualizar al mirar los héroes `id` .

La API web de heroes espera un encabezado especial en las solicitudes de guardado HTTP. Ese encabezado está en la `httpOptions` constante definida en el `HeroService` .

src / app / hero.service.ts

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

Actualice el navegador, cambie el nombre de un héroe y guarde su cambio. La navegación a la vista anterior se implementa en el `save()` método definido en `HeroDetailComponent` . El héroe aparece ahora en la lista con el nombre cambiado.

Añadir un nuevo héroe

Para agregar un héroe, esta aplicación solo necesita el nombre del héroe. Puedes usar un `input` elemento emparejado con un botón de agregar.

Inserte lo siguiente en la `HeroesComponent` plantilla, justo después del encabezado:

src / app / heroes / heroes.component.html (agregar)

```
<div>
  <label>Hero name:
    <input #heroName />
  </label>
  <!-- (click) passes input value to add() and then clears the input -->
  <button (click)="add(heroName.value); heroName.value=''">
    add
  </button>
</div>
```

En respuesta a un evento de clic, llame al controlador de clic del componente y luego borre el campo de entrada para que esté listo para otro nombre.

src / app / heroes / heroes.component.ts (agregar)

```
add(name: string): void {
  name = name.trim();
  if (!name) { return; }
  this.heroService.addHero({ name } as Hero)
    .subscribe(hero => {
      this.heroes.push(hero);
    });
}
```

Cuando el nombre dado no está en blanco, el controlador crea un `Hero` objeto similar al nombre (solo falta el `id`) y lo pasa al `addHero()` método de servicios .

Cuando se `addHero` guarda correctamente, la `subscribe` devolución de llamada recibe al nuevo héroe y lo inserta en la `heroes` lista para su visualización.

Escribirás `HeroService.addHero` en la siguiente sección.

Añadir *HeroService.addHero ()*

Agregue el siguiente `addHero()` método a la `HeroService` clase.

src / app / hero.service.ts (addHero)

```
/** POST: add a new hero to the server */
addHero (hero: Hero): Observable<Hero> {
  return this.http.post<Hero>(this.heroesUrl, hero, httpOptions).pipe(
    tap((hero: Hero) => this.log(`added hero w/ id=${hero.id}`)),
    catchError(this.handleError<Hero>('addHero'))
  );
}
```

`HeroService.addHero()` Se diferencia de `updateHero` dos maneras.

- llama en `HttpClient.post()` lugar de `put()` .
- espera que el servidor genere una identificación para el nuevo héroe, que devuelve al `Observable<Hero>` que llama.

Actualiza el navegador y añade algunos héroes.

Eliminar un heroe

Cada héroe en la lista de héroes debe tener un botón de eliminar.

Agregue el siguiente elemento de botón a la `HeroesComponent` plantilla, después del nombre del héroe en el `<li>` elemento repetido .

```
<button class="delete" title="delete hero"
  (click)="delete(hero)">x</button>
```

El HTML para la lista de héroes debería verse así:

src / app / heroes / heroes.component.html (lista de héroes)

```
<ul class="heroes">
  <li *ngFor="let hero of heroes">
    <a routerLink="/detail/{{hero.id}}">
      <span class="badge">{{hero.id}}</span> {{hero.name}}
    </a>
    <button class="delete" title="delete hero"
      (click)="delete(hero)">x</button>
  </li>
</ul>
```

Para colocar el botón de eliminar en el extremo derecho de la entrada del héroe, agregue algo de CSS a `heroes.component.css` . Encontrarás ese CSS en el código de revisión final a continuación.

Agregue el `delete()` controlador al componente.

src / app / heroes / heroes.component.ts (eliminar)

```
delete(hero: Hero): void {
  this.heroes = this.heroes.filter(h => h !== hero);
  this.heroService.deleteHero(hero).subscribe();
}
```

Aunque el componente delega la eliminación del héroe al `HeroService` , sigue siendo responsable de actualizar su propia lista de héroes. El `delete()` método del componente elimina inmediatamente el *héroe a eliminar* de esa lista, anticipando que `HeroService` tendrá éxito en el servidor.

Realmente no hay nada que hacer con el componente `Observable` devuelto por el componente `heroService.delete()` . **Debe suscribirse de todos modos** .

> Si no lo hace `subscribe()` , el servicio no enviará la solicitud de eliminación al servidor. Como regla general, una `Observable` *no hace nada* hasta que algo se suscribe!
>
> Confirma esto por ti mismo eliminando temporalmente `subscribe()` , haciendo clic en "Panel" y luego en "Héroes". Verás la lista completa de héroes de nuevo.

Añadir *HeroService.deleteHero ()*

Añade un `deleteHero()` método para que te `HeroService` guste esto.

src / app / hero.service.ts (eliminar)

```
/** DELETE: delete the hero from the server */
deleteHero (hero: Hero | number): Observable<Hero> {
  const id = typeof hero === 'number' ? hero : hero.id;
  const url = `${this.heroesUrl}/${id}`;

  return this.http.delete<Hero>(url, httpOptions).pipe(
    tap(_ => this.log(`deleted hero id=${id}`)),
    catchError(this.handleError<Hero>('deleteHero'))
  );
}
```

Tenga en cuenta que

- que llama `HttpClient.delete` .
- La URL es la URL del recurso de héroes más la `id` del héroe que se eliminará.
- No envías datos como lo hiciste con `put` y `post` .
- usted todavía envía el `httpOptions` .

Actualice el navegador y pruebe la nueva funcionalidad de eliminación.

Buscar por nombre

En este último ejercicio, aprenderá a encadenar `Observable` operadores para que pueda minimizar la cantidad de solicitudes HTTP similares y consumir el ancho de banda de la red de manera económica.

Agregará una función de *búsqueda de héroes* al *Panel de control* . A medida que el usuario escribe un nombre en un cuadro de búsqueda, realizarás solicitudes HTTP repetidas de héroes filtrados por ese nombre. Tu objetivo es emitir solo tantas solicitudes como sea necesario.

HeroService.searchHeroes

Comience agregando un `searchHeroes` método al archivo `HeroService` .

src / app / hero.service.ts

```
/* GET heroes whose name contains search term */
searchHeroes(term: string): Observable<Hero[]> {
  if (!term.trim()) {
    // if not search term, return empty hero array.
    return of([]);
  }
  return this.http.get<Hero[]>(`${this.heroesUrl}/?name=${term}`).pipe(
    tap(_ => this.log(`found heroes matching "${term}"`)),
    catchError(this.handleError<Hero[]>('searchHeroes', []))
  );
}
```

El método regresa inmediatamente con una matriz vacía si no hay un término de búsqueda. El resto se asemeja mucho `getHeroes()` . La única diferencia significativa es la URL, que incluye una cadena de consulta con el término de búsqueda.

Añadir búsqueda al Dashboard

Abra la `DashboardComponent` *plantilla* y agregue el elemento de búsqueda de héroe `<app-hero-search>` , en la parte inferior de la `DashboardComponent` plantilla.

src / app / dashboard / dashboard.component.html

```
<h3>Top Heroes</h3>
<div class="grid grid-pad">
  <a *ngFor="let hero of heroes" class="col-1-4"
      routerLink="/detail/{{hero.id}}">
    <div class="module hero">
      <h4>{{hero.name}}</h4>
    </div>
  </a>
</div>

<app-hero-search></app-hero-search>
```

Esta plantilla se parece mucho al repetidor de la plantilla.`*ngFor` `HeroesComponent`

Desafortunadamente, agregar este elemento rompe la aplicación. Angular no puede encontrar un componente con un selector que coincida `<app-hero-search>` .

El `HeroSearchComponent` no existe todavía. Arregla eso.

Crear *HeroSearchComponent*

Crear un `HeroSearchComponent` con el CLI.

```
ng generate component hero-search
```

El CLI genera los tres `HeroSearchComponent` archivos y agrega el componente a las `AppModule` declaraciones.

Reemplace la `HeroSearchComponent` *plantilla* generada con un cuadro de texto y una lista de resultados de búsqueda coincidentes como este.

src / app / hero-search / hero-search.component.html

```
<div id="search-component">
  <h4>Hero Search</h4>

  <input #searchBox id="search-box" (keyup)="search(searchBox.value)" />

  <ul class="search-result">
    <li *ngFor="let hero of heroes$ | async" >
      <a routerLink="/detail/{{hero.id}}">
        {{hero.name}}
      </a>
    </li>
  </ul>
</div>
```

Agregue estilos de CSS privados `hero-search.component.css` como se indica en la revisión final del código a continuación.

A medida que el usuario escribe en el cuadro de búsqueda, un *enlace de* evento *keyup* llama al `search()` método del componente con el nuevo valor del cuadro de búsqueda.

AsyncPipe

Como era de esperar, el héroe repite objetos.`*ngFor`

Mire de cerca y verá que la iteración sobre una lista llamada , no .`*ngFor` `heroes$` `heroes`

```
<li *ngFor="let hero of heroes$ | async" >
```

El `$` es una convención que indica `heroes$` es una `Observable` , no una matriz.

El no puede hacer nada con un . Pero también hay un carácter de canalización ( ) seguido de , que identifica los de Angular .`*ngFor` `Observable` `|` `async` `AsyncPipe`

El se `AsyncPipe` suscribe a `Observable` automáticamente para que no tenga que hacerlo en la clase de componente.

Arregla la clase *HeroSearchComponent*

Reemplace la `HeroSearchComponent` clase y los metadatos generados de la siguiente manera.

src / app / hero-search / hero-search.component.ts

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

import { Observable, Subject } from 'rxjs';

import {
   debounceTime, distinctUntilChanged, switchMap
 } from 'rxjs/operators';

import { Hero } from '../hero';
import { HeroService } from '../hero.service';

@Component({
  selector: 'app-hero-search',
  templateUrl: './hero-search.component.html',
  styleUrls: [ './hero-search.component.css' ]
})
export class HeroSearchComponent implements OnInit {
  heroes$: Observable<Hero[]>;
  private searchTerms = new Subject<string>();

  constructor(private heroService: HeroService) {}

  // Push a search term into the observable stream.
  search(term: string): void {
    this.searchTerms.next(term);
  }

  ngOnInit(): void {
    this.heroes$ = this.searchTerms.pipe(
      // wait 300ms after each keystroke before considering the term
      debounceTime(300),

      // ignore new term if same as previous term
      distinctUntilChanged(),

      // switch to new search observable each time the term changes
      switchMap((term: string) => this.heroService.searchHeroes(term)),
    );
```

```
39.   }
40. }
```

Observe la declaración de `heroes$` como`Observable`

```
heroes$: Observable<Hero[]>;
```

Lo pondrás en `ngOnInit()` . Antes de hacerlo, concéntrate en la definición de `searchTerms` .

El tema *searchTerms* RxJS

La `searchTerms` propiedad está declarada como RxJS `Subject` .

```
private searchTerms = new Subject<string>();

// Push a search term into the observable stream.
search(term: string): void {
  this.searchTerms.next(term);
}
```

A `Subject` es tanto una fuente de valores *observables* como un `Observable` sí mismo. Puede suscribirse a una `Subject` como lo haría cualquiera `Observable` .

También puede insertar valores en eso `Observable` llamando a su `next(value)` método como lo hace el `search()` método.

El `search()` método se llama mediante un *enlace de evento* al *evento* del cuadro de texto `keystroke` .

```
<input #searchBox id="search-box" (keyup)="search(searchBox.value)" />
```

Cada vez que el usuario escribe en el cuadro de texto, el enlace llama `search()` con el valor del cuadro de texto, un "término de búsqueda". El se `searchTerms` convierte en un `Observable` flujo constante de términos de búsqueda.

Encadenamiento de operadores RxJS.

Pasar un nuevo término de búsqueda directamente a la `searchHeroes()` pulsación de tecla después de cada usuario crearía una cantidad excesiva de solicitudes HTTP, gravar los recursos del servidor y grabar a través del plan de datos de la red celular.

En su lugar, el `ngOnInit()` método canaliza lo `searchTerms` observable a través de una secuencia de operadores RxJS que reducen el número de llamadas a la `searchHeroes()` , devolviendo finalmente un observable de resultados de búsqueda de héroe oportunos (cada uno `Hero[]`).

Aquí está el código.

```
this.heroes$ = this.searchTerms.pipe(
  // wait 300ms after each keystroke before considering the term
  debounceTime(300),

  // ignore new term if same as previous term
  distinctUntilChanged(),

  // switch to new search observable each time the term changes
  switchMap((term: string) => this.heroService.searchHeroes(term)),
);
```

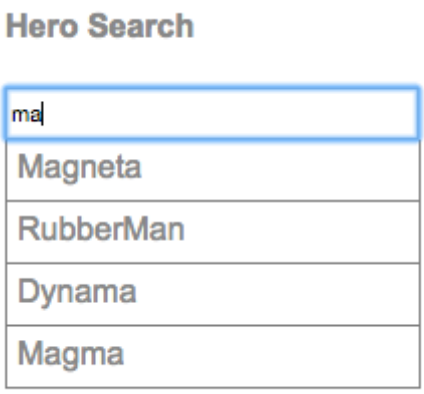
- `debounceTime(300)` espera hasta que el flujo de nuevos eventos de cadena se detenga durante 300 milisegundos antes de pasar por la última cadena. Nunca harás solicitudes con más frecuencia de 300ms.
- `distinctUntilChanged()` asegura que una solicitud se envíe solo si el texto del filtro ha cambiado.
- `switchMap()` llama al servicio de búsqueda para cada término de búsqueda que lo hace a través de `debounce` y `distinctUntilChanged` . Cancela y descarta los observables de búsqueda anteriores, devolviendo solo el servicio de búsqueda más reciente observable.

> Con el operador switchMap , cada evento clave que califica puede desencadenar una `HttpClient.get()` llamada de método. Incluso con una pausa de 300 ms entre las solicitudes, puede tener varias solicitudes HTTP en vuelo y es posible que no vuelvan en el orden enviado.
>
> `switchMap()` conserva la orden de solicitud original y devuelve solo lo observable de la llamada de método HTTP más reciente. Los resultados de llamadas anteriores se cancelan y se descartan.
>
> Tenga en cuenta que *cancelar* un `searchHeroes()` *Observable* anterior no cancela una solicitud HTTP pendiente. Los resultados no deseados simplemente se descartan antes de que alcancen el código de su aplicación.

Recuerde que la *clase de* componente no se suscribe a lo `heroes$` *observable* . Ese es el trabajo de `AsyncPipe` la plantilla.

Intentalo

Ejecutar la aplicación de nuevo. En el *Panel* , ingrese un poco de texto en el cuadro de búsqueda. Si ingresas caracteres que coincidan con los nombres de héroes existentes, verás algo como esto.



Resumen

Estás al final de tu viaje y has logrado mucho.

- Agregaste las dependencias necesarias para usar HTTP en la aplicación.
- Has refaccionado `HeroService` para cargar héroes desde una API web.
- Ampliada `HeroService` para apoyar `post()` , `put()` y `delete()` métodos.
- Actualizó los componentes para permitir agregar, editar y eliminar héroes.
- Has configurado una API web en memoria.
- Aprendiste a usar los observables.

Con esto concluye el tutorial "Tour of Heroes". Está listo para aprender más sobre el desarrollo angular en la sección de fundamentos, comenzando con la guía de Arquitectura .