The Hero Editor

The application now has a basic title. Next you will create a new component to display hero information and place that component in the application shell.

Create the heroes component

Using the Angular CLI, generate a new component named `heroes`.

```
ng generate component heroes
```

The CLI creates a new folder, `src/app/heroes/` and generates the three files of the `HeroesComponent`.

The `HeroesComponent` class file is as follows:

app/heroes/heroes.component.ts (initial version)

```
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() {
  }

}
```

You always import the `Component` symbol from the Angular core library and annotate the component class with `@Component`.

`@Component` is a decorator function that specifies the Angular metadata for the component.

The CLI generated three metadata properties:

1. `selector` — the component's CSS element selector
2. `templateUrl` — the location of the component's template file.
3. `styleUrls` — the location of the component's private CSS styles.

The CSS element selector, `'app-heroes'`, matches the name of the HTML element that identifies this component within a parent component's template.

The `ngOnInit` is a lifecycle hook Angular calls `ngOnInit` shortly after creating a component. It's a good place to put initialization logic.

Always `export` the component class so you can `import` it elsewhere ... like in the `AppModule`.

Add a `hero` property

Add a `hero` property to the `HeroesComponent` for a hero named "Windstorm."

heroes.component.ts (hero property)

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

Show the hero

Open the `heroes.component.html` template file. Delete the default text generated by the Angular CLI and replace it with a data binding to the new `hero` property.

heroes.component.html

```
{{hero}}
```

Show the `HeroesComponent` view

To display the `HeroesComponent`, you must add it to the template of the shell `AppComponent`.

Remember that `app-heroes` is the element selector for the `HeroesComponent`. So add an `<app-heroes>` element to the `AppComponent` template file, just below the title.

src/app/app.component.html

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

Assuming that the CLI `ng serve` command is still running, the browser should refresh and display both the application title and the hero name.

Create a Hero class

A real hero is more than a name.

Create a `Hero` class in its own file in the `src/app` folder. Give it `id` and `name` properties.

src/app/hero.ts

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

Return to the `HeroesComponent` class and import the `Hero` class.

Refactor the component's `hero` property to be of type `Hero`. Initialize it with an `id` of `1` and the name `Windstorm`.

The revised `HeroesComponent` class file should look like this:

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() {
  }

}
```

The page no longer displays properly because you changed the hero from a string to an object.

Show the hero object

Update the binding in the template to announce the hero's name and show both `id` and `name` in a details layout like this:

heroes.component.html (HeroesComponent's template)

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

The browser refreshes and display's the hero's information.

Format with the `UppercasePipe`

Modify the `hero.name` binding like this.

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

The browser refreshes and now the hero's name is displayed in capital letters.

The word `uppercase` in the interpolation binding, right after the pipe operator (|), activates the built-in `UppercasePipe`.

Pipes are a good way to format strings, currency amounts, dates and other display data. Angular ships with several built-in pipes and you can create your own.

Edit the hero

Users should be able to edit the hero name in an `<input>` textbox.

The textbox should both *display* the hero's `name` property and *update* that property as the user types. That means data flow from the component class *out to the screen* and from the screen *back to the class*.

To automate that data flow, setup a two-way data binding between the `<input>` form element and the `hero.name` property.

Two-way binding

Refactor the details area in the `HeroesComponent` template so it looks like this:

src/app/heroes/heroes.component.html (HeroesComponent's template)

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

[(ngModel)] is Angular's two-way data binding syntax.

Here it binds the `hero.name` property to the HTML textbox so that data can flow *in both directions:* from the `hero.name` property to the textbox, and from the textbox back to the `hero.name`.

The missing `FormsModule`

Notice that the app stopped working when you added `[(ngModel)]`.

To see the error, open the browser development tools and look in the console for a message like

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

Although `ngModel` is a valid Angular directive, it isn't available by default.

It belongs to the optional `FormsModule` and you must *opt-in* to using it.

`AppModule`

Angular needs to know how the pieces of your application fit together and what other files and libraries the app requires. This information is called *metadata*

Some of the metadata is in the `@Component` decorators that you added to your component classes. Other critical metadata is in `@NgModule` decorators.

The most important `@NgModule` decorator annotates the top-level **AppModule** class.

The Angular CLI generated an `AppModule` class in `src/app/app.module.ts` when it created the project. This is where you *opt-in* to the `FormsModule`.

Import `FormsModule`

Open `AppModule` (`app.module.ts`) and import the `FormsModule` symbol from the `@angular/forms` library.

app.module.ts (FormsModule symbol import)

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

Then add `FormsModule` to the `@NgModule` metadata's `imports` array, which contains a list of external modules that the app needs.

app.module.ts (@NgModule imports)

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

When the browser refreshes, the app should work again. You can edit the hero's name and see the changes reflected immediately in the `<h2>` above the textbox.

Declare `HeroesComponent`

Every component must be declared in *exactly one* NgModule.

You didn't declare the `HeroesComponent`. So why did the application work?

It worked because the Angular CLI declared `HeroesComponent` in the `AppModule` when it generated that component.

Open `src/app/app.module.ts` and find `HeroesComponent` imported near the top.

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

The `HeroesComponent` is declared in the `@NgModule.declarations` array.

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

Note that `AppModule` declares both application components, `AppComponent` and `HeroesComponent`.

Final code review

Your app should look like this live example. Here are the code files discussed on this page.

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() {
  }

}
```

src/app/heroes/heroes.component.html

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

src/app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms'; // <-- NgModel lives here

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

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

src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Tour of Heroes';
}
```

src/app/app.component.html

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

src/app/hero.ts

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

Summary

- You used the CLI to create a second `HeroesComponent`.
- You displayed the `HeroesComponent` by adding it to the `AppComponent` shell.
- You applied the `UppercasePipe` to format the name.
- You used two-way data binding with the `ngModel` directive.
- You learned about the `AppModule`.
- You imported the `FormsModule` in the `AppModule` so that Angular would recognize and apply the `ngModel` directive.
- You learned the importance of declaring components in the `AppModule` and appreciated that the CLI declared it for you.

© 2010–2018 Google, Inc.
Licensed under the Creative Commons Attribution License 4.0.
https://angular.io/tutorial/toh-pt1

Exported from DevDocs — https://devdocs.io

Display a Heroes List

In this page, you'll expand the Tour of Heroes app to display a list of heroes, and allow users to select a hero and display the hero's details.

Create mock heroes

You'll need some heroes to display.

Eventually you'll get them from a remote data server. For now, you'll create some *mock heroes* and pretend they came from the server.

Create a file called `mock-heroes.ts` in the `src/app/` folder. Define a `HEROES` constant as an array of ten heroes and export it. The file should look like this.

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' }
];
```

Displaying heroes

You're about to display the list of heroes at the top of the `HeroesComponent`.

Open the `HeroesComponent` class file and import the mock `HEROES`.

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

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

Add a `heroes` property to the class that exposes these heroes for binding.

```
heroes = HEROES;
```

List heroes with `*ngFor`

Open the `HeroesComponent` template file and make the following changes:

- Add an `<h2>` at the top,

- Below it add an HTML unordered list (`<ul>`)
- Insert an `<li>` within the `<ul>` that displays properties of a `hero`.
- Sprinkle some CSS classes for styling (you'll add the CSS styles shortly).

Make it look like this:

heroes.component.html (heroes template)

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

Now change the `<li>` to this:

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

The `*ngFor` is Angular's *repeater* directive. It repeats the host element for each element in a list.

In this example

- `<li>` is the host element
- `heroes` is the list from the `HeroesComponent` class.
- `hero` holds the current hero object for each iteration through the list.

> Don't forget the asterisk (*) in front of `ngFor`. It's a critical part of the syntax.

After the browser refreshes, the list of heroes appears.

Style the heroes

The heroes list should be attractive and should respond visually when users hover over and select a hero from the list.

In the first tutorial, you set the basic styles for the entire application in `styles.css`. That stylesheet didn't include styles for this list of heroes.

You could add more styles to `styles.css` and keep growing that stylesheet as you add components.

You may prefer instead to define private styles for a specific component and keep everything a component needs—the code, the HTML, and the CSS —together in one place.

This approach makes it easier to re-use the component somewhere else and deliver the component's intended appearance even if the global styles are different.

You define private styles either inline in the `@Component.styles` array or as stylesheet file(s) identified in the `@Component.styleUrls` array.

When the CLI generated the `HeroesComponent`, it created an empty `heroes.component.css` stylesheet for the `HeroesComponent` and pointed to it in `@Component.styleUrls` like this.

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

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

Open the `heroes.component.css` file and paste in the private CSS styles for the `HeroesComponent`. You'll find them in the final code review at the bottom of this guide.

Styles and stylesheets identified in `@Component` metadata are scoped to that specific component. The `heroes.component.css` styles apply only to the `HeroesComponent` and don't affect the outer HTML or the HTML in any other component.

Master/Detail

When the user clicks a hero in the **master** list, the component should display the selected hero's **details** at the bottom of the page.

In this section, you'll listen for the hero item click event and update the hero detail.

Add a click event binding

Add a click event binding to the `<li>` like this:

heroes.component.html (template excerpt)

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

This is an example of Angular's event binding syntax.

The parentheses around `click` tell Angular to listen for the `<li>` element's `click` event. When the user clicks in the `<li>`, Angular executes the `onSelect(hero)` expression.

`onSelect()` is a `HeroesComponent` method that you're about to write. Angular calls it with the `hero` object displayed in the clicked `<li>`, the same `hero` defined previously in the `*ngFor` expression.

Add the click event handler

Rename the component's `hero` property to `selectedHero` but don't assign it. There is no *selected hero* when the application starts.

Add the following `onSelect()` method, which assigns the clicked hero from the template to the component's `selectedHero`.

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

```
selectedHero: Hero;

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

Update the details template

The template still refers to the component's old `hero` property which no longer exists. Rename `hero` to `selectedHero`.

heroes.component.html (selected hero details)

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

Hide empty details with `*ngIf`

After the browser refreshes, the application is broken.

Open the browser developer tools and look in the console for an error message like this:

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

Now click one of the list items. The app seems to be working again. The heroes appear in a list and details about the clicked hero appear at the bottom of the page.

What happened?

When the app starts, the `selectedHero` is `undefined` *by design*.

Binding expressions in the template that refer to properties of `selectedHero` — expressions like `{{selectedHero.name}}` — *must fail* because there is no selected hero.

The fix

The component should only display the selected hero details if the `selectedHero` exists.

Wrap the hero detail HTML in a `<div>`. Add Angular's `*ngIf` directive to the `<div>` and set it to `selectedHero`.

> Don't forget the asterisk (*) in front of `ngIf`. It's a critical part of the syntax.

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

```
<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>
```

After the browser refreshes, the list of names reappears. The details area is blank. Click a hero and its details appear.

Why it works

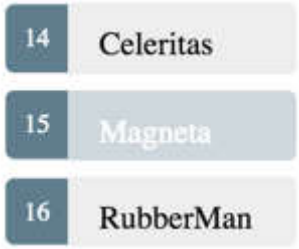
When `selectedHero` is undefined, the `ngIf` removes the hero detail from the DOM. There are no `selectedHero` bindings to worry about.

When the user picks a hero, `selectedHero` has a value and `ngIf` puts the hero detail into the DOM.

Style the selected hero

It's difficult to identify the *selected hero* in the list when all `<li>` elements look alike.

If the user clicks "Magneta", that hero should render with a distinctive but subtle background color like this:



That *selected hero* coloring is the work of the `.selected` CSS class in the styles you added earlier. You just have to apply the `.selected` class to the `<li>` when the user clicks it.

The Angular class binding makes it easy to add and remove a CSS class conditionally. Just add `[class.some-css-class]="some-condition"` to the element you want to style.

Add the following `[class.selected]` binding to the `<li>` in the `HeroesComponent` template:

heroes.component.html (toggle the 'selected' CSS class)

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

When the current row hero is the same as the `selectedHero`, Angular adds the `selected` CSS class. When the two heroes are different, Angular removes the class.

The finished `<li>` looks like this:

heroes.component.html (list item hero)

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

Final code review

Your app should look like this live example.

Here are the code files discussed on this page, including the `HeroesComponent` styles.

src/app/heroes/heroes.component.ts

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

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

  heroes = HEROES;

  selectedHero: Hero;


  constructor() { }

  ngOnInit() {
  }

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

src/app/heroes/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>

<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>
```

src/app/heroes/heroes.component.css

```
/* HeroesComponent's private CSS styles */
.selected {
  background-color: #CFD8DC !important;
  color: white;
}
.heroes {
  margin: 0 0 2em 0;
  list-style-type: none;
  padding: 0;
  width: 15em;
}
.heroes li {
  cursor: pointer;
  position: relative;
  left: 0;
  background-color: #EEE;
  margin: .5em;
  padding: .3em 0;
  height: 1.6em;
  border-radius: 4px;
}
.heroes li.selected:hover {
  background-color: #BBD8DC !important;
  color: white;
}
.heroes li:hover {
  color: #607D8B;
  background-color: #DDD;
  left: .1em;
}
.heroes .text {
  position: relative;
  top: -3px;
}
.heroes .badge {
  display: inline-block;
  font-size: small;
  color: white;
  padding: 0.8em 0.7em 0 0.7em;
  background-color: #607D8B;
  line-height: 1em;
  position: relative;
  left: -1px;
  top: -4px;
  height: 1.8em;
  margin-right: .8em;
  border-radius: 4px 0 0 4px;
}
```

Summary

- The Tour of Heroes app displays a list of heroes in a Master/Detail view.
- The user can select a hero and see that hero's details.
- You used `*ngFor` to display a list.
- You used `*ngIf` to conditionally include or exclude a block of HTML.
- You can toggle a CSS style class with a `class` binding.

© 2010–2018 Google, Inc.
Licensed under the Creative Commons Attribution License 4.0.
https://angular.io/tutorial/toh-pt2

Exported from DevDocs — https://devdocs.io

Master/Detail Components

At the moment, the `HeroesComponent` displays both the list of heroes and the selected hero's details.

Keeping all features in one component as the application grows will not be maintainable. You'll want to split up large components into smaller sub-components, each focused on a specific task or workflow.

In this page, you'll take the first step in that direction by moving the hero details into a separate, reusable `HeroDetailComponent`.

The `HeroesComponent` will only present the list of heroes. The `HeroDetailComponent` will present details of a selected hero.

Make the `HeroDetailComponent`

Use the Angular CLI to generate a new component named `hero-detail`.

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

The command scaffolds the `HeroDetailComponent` files and declares the component in `AppModule`.

Write the template

Cut the HTML for the hero detail from the bottom of the `HeroesComponent` template and paste it over the generated boilerplate in the `HeroDetailComponent` template.

The pasted HTML refers to a `selectedHero`. The new `HeroDetailComponent` can present *any* hero, not just a selected hero. So replace "selectedHero" with "hero" everywhere in the template.

When you're done, the `HeroDetailComponent` template should look like this:

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>
```

Add the `@Input()` hero property

The `HeroDetailComponent` template binds to the component's `hero` property which is of type `Hero`.

Open the `HeroDetailComponent` class file and import the `Hero` symbol.

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

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

The `hero` property must be an *Input* property, annotated with the `@Input()` decorator, because the *external* `HeroesComponent` will bind to it like this.

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

Amend the `@angular/core` import statement to include the `Input` symbol.

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

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

Add a `hero` property, preceded by the `@Input()` decorator.

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

That's the only change you should make to the `HeroDetailComponent` class. There are no more properties. There's no presentation logic. This component simply receives a hero object through its `hero` property and displays it.

Show the `HeroDetailComponent`

The `HeroesComponent` is still a master/detail view.

It used to display the hero details on its own, before you cut that portion of the template. Now it will delegate to the `HeroDetailComponent`.

The two components will have a parent/child relationship. The parent `HeroesComponent` will control the child `HeroDetailComponent` by sending it a new hero to display whenever the user selects a hero from the list.

You won't change the `HeroesComponent` *class* but you will change its *template*.

Update the `HeroesComponent` template

The `HeroDetailComponent` selector is `'app-hero-detail'`. Add an `<app-hero-detail>` element near the bottom of the `HeroesComponent` template, where the hero detail view used to be.

Bind the `HeroesComponent.selectedHero` to the element's `hero` property like this.

heroes.component.html (HeroDetail binding)

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

`[hero]="selectedHero"` is an Angular property binding.

It's a *one way* data binding from the `selectedHero` property of the `HeroesComponent` to the `hero` property of the target element, which maps to the `hero` property of the `HeroDetailComponent`.

Now when the user clicks a hero in the list, the `selectedHero` changes. When the `selectedHero` changes, the *property binding* updates `hero` and the `HeroDetailComponent` displays the new hero.

The revised `HeroesComponent` template should look like this:

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>
```

The browser refreshes and the app starts working again as it did before.

What changed?

As before, whenever a user clicks on a hero name, the hero detail appears below the hero list. Now the `HeroDetailComponent` is presenting those details instead of the `HeroesComponent`.

Refactoring the original `HeroesComponent` into two components yields benefits, both now and in the future:

1. You simplified the `HeroesComponent` by reducing its responsibilities.
2. You can evolve the `HeroDetailComponent` into a rich hero editor without touching the parent `HeroesComponent`.
3. You can evolve the `HeroesComponent` without touching the hero detail view.
4. You can re-use the `HeroDetailComponent` in the template of some future component.

Final code review

Here are the code files discussed on this page and your app should look like this live example.

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

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

@Component({
  selector: 'app-hero-detail',
  templateUrl: './hero-detail.component.html',
  styleUrls: ['./hero-detail.component.css']
})
export class HeroDetailComponent implements OnInit {
  @Input() hero: Hero;

  constructor() { }

  ngOnInit() {
  }

}
```

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>
```

src/app/heroes/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>
```

Summary

- You created a separate, reusable `HeroDetailComponent`.
- You used a property binding to give the parent `HeroesComponent` control over the child `HeroDetailComponent`.
- You used the `@Input` decorator to make the `hero` property available for binding by the external `HeroesComponent`.

© 2010–2018 Google, Inc.
Licensed under the Creative Commons Attribution License 4.0.
https://angular.io/tutorial/toh-pt3

Exported from DevDocs — https://devdocs.io

Services

The Tour of Heroes `HeroesComponent` is currently getting and displaying fake data.

After the refactoring in this tutorial, `HeroesComponent` will be lean and focused on supporting the view. It will also be easier to unit-test with a mock service.

Why services

Components shouldn't fetch or save data directly and they certainly shouldn't knowingly present fake data. They should focus on presenting data and delegate data access to a service.

In this tutorial, you'll create a `HeroService` that all application classes can use to get heroes. Instead of creating that service with `new`, you'll rely on Angular *dependency injection* to inject it into the `HeroesComponent` constructor.

Services are a great way to share information among classes that *don't know each other*. You'll create a `MessageService` and inject it in two places:

1. in `HeroService` which uses the service to send a message.
2. in `MessagesComponent` which displays that message.

Create the `HeroService`

Using the Angular CLI, create a service called `hero`.

```
ng generate service hero
```

The command generates skeleton `HeroService` class in `src/app/hero.service.ts` The `HeroService` class should look like the following example.

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

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

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

  constructor() { }

}
```

`@Injectable()` services

Notice that the new service imports the Angular `Injectable` symbol and annotates the class with the `@Injectable()` decorator. This marks the class as one that participates in the *dependency injection system*. The `HeroService` class is going to provide an injectable service, and it can also have its own injected dependencies. It doesn't have any dependencies yet, but it will soon.

The `@Injectable()` decorator accepts a metadata object for the service, the same way the `@Component()` decorator did for your component classes.

Get hero data

The `HeroService` could get hero data from anywhere—a web service, local storage, or a mock data source.

Removing data access from components means you can change your mind about the implementation anytime, without touching any components. They don't know how the service works.

The implementation in *this* tutorial will continue to deliver *mock heroes*.

Import the `Hero` and `HEROES`.

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

Add a `getHeroes` method to return the *mock heroes*.

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

Provide the `HeroService`

You must make the `HeroService` available to the dependency injection system before Angular can *inject* it into the `HeroesComponent`, as you will do below. You do this by registering a *provider*. A provider is something that can create or deliver a service; in this case, it instantiates the `HeroService` class to provide the service.

Now, you need to make sure that the `HeroService` is registered as the provider of this service. You are registering it with an *injector*, which is the object that is responsible for choosing and injecting the provider where it is required.

By default, the Angular CLI command `ng generate service` registers a provider with the *root injector* for your service by including provider metadata in the `@Injectable` decorator.

If you look at the `@Injectable()` statement right before the `HeroService` class definition, you can see that the `providedIn` metadata value is 'root':

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

When you provide the service at the root level, Angular creates a single, shared instance of `HeroService` and injects into any class that asks for it. Registering the provider in the `@Injectable` metadata also allows Angular to optimize an app by removing the service if it turns out not to be used after all.

> If you need to, you can register providers at different levels: in the `HeroesComponent`, in the `AppComponent`, in the `AppModule`. For instance, you could have told the CLI to provide the service at the module level automatically by appending `--module=app`.

```
ng generate service hero --module=app
```

To learn more about providers and injectors, see the Dependency Injection guide.

The `HeroService` is now ready to plug into the `HeroesComponent`.

This is a interim code sample that will allow you to provide and use the `HeroService`. At this point, the code will differ from the `HeroService` in the "final code review".

Update `HeroesComponent`

Open the `HeroesComponent` class file.

Delete the `HEROES` import, because you won't need that anymore. Import the `HeroService` instead.

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

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

Replace the definition of the `heroes` property with a simple declaration.

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

Inject the `HeroService`

Add a private `heroService` parameter of type `HeroService` to the constructor.

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

The parameter simultaneously defines a private `heroService` property and identifies it as a `HeroService` injection site.

When Angular creates a `HeroesComponent`, the Dependency Injection system sets the `heroService` parameter to the singleton instance of `HeroService`.

Add `getHeroes()`

Create a function to retrieve the heroes from the service.

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

Call it in `ngOnInit`

While you could call `getHeroes()` in the constructor, that's not the best practice.

Reserve the constructor for simple initialization such as wiring constructor parameters to properties. The constructor shouldn't *do anything*. It certainly shouldn't call a function that makes HTTP requests to a remote server as a *real* data service would.

Instead, call `getHeroes()` inside the *ngOnInit lifecycle hook* and let Angular call `ngOnInit` at an appropriate time *after* constructing a `HeroesComponent` instance.

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

See it run

After the browser refreshes, the app should run as before, showing a list of heroes and a hero detail view when you click on a hero name.

Observable data

The `HeroService.getHeroes()` method has a *synchronous signature*, which implies that the `HeroService` can fetch heroes synchronously. The `HeroesComponent` consumes the `getHeroes()` result as if heroes could be fetched synchronously.

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

This will not work in a real app. You're getting away with it now because the service currently returns *mock heroes*. But soon the app will fetch heroes from a remote server, which is an inherently *asynchronous* operation.

The `HeroService` must wait for the server to respond, `getHeroes()` cannot return immediately with hero data, and the browser will not block while the service waits.

`HeroService.getHeroes()` must have an *asynchronous signature* of some kind.

It can take a callback. It could return a `Promise`. It could return an `Observable`.

In this tutorial, `HeroService.getHeroes()` will return an `Observable` in part because it will eventually use the Angular `HttpClient.get` method to fetch the heroes and `HttpClient.get()` returns an `Observable`.

Observable `HeroService`

`Observable` is one of the key classes in the RxJS library.

In a later tutorial on HTTP, you'll learn that Angular's `HttpClient` methods return RxJS `Observable`s. In this tutorial, you'll simulate getting data from the server with the RxJS `of()` function.

Open the `HeroService` file and import the `Observable` and `of` symbols from RxJS.

src/app/hero.service.ts (Observable imports)

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

Replace the `getHeroes` method with this one.

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

`of(HEROES)` returns an `Observable<Hero[]>` that emits *a single value*, the array of mock heroes.

> In the HTTP tutorial, you'll call `HttpClient.get<Hero[]>()` which also returns an `Observable<Hero[]>` that emits *a single value*, an array of heroes from the body of the HTTP response.

Subscribe in `HeroesComponent`

The `HeroService.getHeroes` method used to return a `Hero[]`. Now it returns an `Observable<Hero[]>`.

You'll have to adjust to that difference in `HeroesComponent`.

Find the `getHeroes` method and replace it with the following code (shown side-by-side with the previous version for comparison)

heroes.component.ts (Observable)

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

heroes.component.ts (Original)

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

`Observable.subscribe()` is the critical difference.

The previous version assigns an array of heroes to the component's `heroes` property. The assignment occurs *synchronously*, as if the server could return heroes instantly or the browser could freeze the UI while it waited for the server's response.

That *won't work* when the `HeroService` is actually making requests of a remote server.

The new version waits for the `Observable` to emit the array of heroes— which could happen now or several minutes from now. Then `subscribe` passes the emitted array to the callback, which sets the component's `heroes` property.

This asynchronous approach *will work* when the `HeroService` requests heroes from the server.

Show messages

In this section you will

- add a `MessagesComponent` that displays app messages at the bottom of the screen.

- create an injectable, app-wide `MessageService` for sending messages to be displayed
- inject `MessageService` into the `HeroService`
- display a message when `HeroService` fetches heroes successfully.

Create `MessagesComponent`

Use the CLI to create the `MessagesComponent`.

```
ng generate component messages
```

The CLI creates the component files in the `src/app/messages` folder and declare `MessagesComponent` in `AppModule`.

Modify the `AppComponent` template to display the generated `MessagesComponent`

/src/app/app.component.html

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

You should see the default paragraph from `MessagesComponent` at the bottom of the page.

Create the `MessageService`

Use the CLI to create the `MessageService` in `src/app`.

```
ng generate service message
```

Open `MessageService` and replace its contents with the following.

/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 = [];
  }
}
```

The service exposes its cache of `messages` and two methods: one to `add()` a message to the cache and another to `clear()` the cache.

Inject it into the `HeroService`

Re-open the `HeroService` and import the `MessageService`.

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

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

Modify the constructor with a parameter that declares a private `messageService` property. Angular will inject the singleton `MessageService` into that property when it creates the `HeroService`.

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

> This is a typical "*service-in-service*" scenario: you inject the `MessageService` into the `HeroService` which is injected into the `HeroesComponent`.

Send a message from `HeroService`

Modify the `getHeroes` method to send a message when the heroes are fetched.

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

Display the message from `HeroService`

The `MessagesComponent` should display all messages, including the message sent by the `HeroService` when it fetches heroes.

Open `MessagesComponent` and import the `MessageService`.

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

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

Modify the constructor with a parameter that declares a **public** `messageService` property. Angular will inject the singleton `MessageService` into that property when it creates the `MessagesComponent`.

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

The `messageService` property **must be public** because you're about to bind to it in the template.

> Angular only binds to *public* component properties.

Bind to the `MessageService`

Replace the CLI-generated `MessagesComponent` template with the following.

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>
```

This template binds directly to the component's `messageService`.

- The `*ngIf` only displays the messages area if there are messages to show.
- An `*ngFor` presents the list of messages in repeated `<div>` elements.
- An Angular event binding binds the button's click event to `MessageService.clear()`.

The messages will look better when you add the private CSS styles to `messages.component.css` as listed in one of the "final code review" tabs below.

The browser refreshes and the page displays the list of heroes. Scroll to the bottom to see the message from the `HeroService` in the message area. Click the "clear" button and the message area disappears.

Final code review

Here are the code files discussed on this page and your app should look like this live example.

src/app/hero.service.ts

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

import { Observable, of } from 'rxjs';

import { Hero } from './hero';
import { HEROES } from './mock-heroes';
import { MessageService } from './message.service';

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

  constructor(private messageService: MessageService) { }

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

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 = [];
  }
}
```

src/app/heroes/heroes.component.ts

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

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

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

  selectedHero: Hero;

  heroes: Hero[];

  constructor(private heroService: HeroService) { }

  ngOnInit() {
    this.getHeroes();
  }

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

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

src/app/messages/messages.component.ts

```
import { Component, OnInit } from '@angular/core';
import { MessageService } from '../message.service';

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

  constructor(public messageService: MessageService) {}

  ngOnInit() {
  }

}
```

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>
```

src/app/messages/messages.component.css

```
/* MessagesComponent's private CSS styles */
h2 {
  color: red;
  font-family: Arial, Helvetica, sans-serif;
  font-weight: lighter;
}
body {
  margin: 2em;
}
body, input[text], button {
  color: crimson;
  font-family: Cambria, Georgia;
}

button.clear {
  font-family: Arial;
  background-color: #eee;
  border: none;
  padding: 5px 10px;
  border-radius: 4px;
  cursor: pointer;
  cursor: hand;
}
button:hover {
  background-color: #cfd8dc;
}
button:disabled {
  background-color: #eee;
  color: #aaa;
```

```
  cursor: auto;
}
button.clear {
  color: #888;
  margin-bottom: 12px;
}
```

src/app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { HeroesComponent } from './heroes/heroes.component';
import { HeroDetailComponent } from './hero-detail/hero-detail.component';
import { MessagesComponent } from './messages/messages.component';

@NgModule({
  declarations: [
    AppComponent,
    HeroesComponent,
    HeroDetailComponent,
    MessagesComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [
    // no need to place any providers due to the `providedIn` flag...
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

src/app/app.component.html

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

Summary

- You refactored data access to the `HeroService` class.
- You registered the `HeroService` as the *provider* of its service at the root level so that it can be injected anywhere in the app.
- You used Angular Dependency Injection to inject it into a component.
- You gave the `HeroService` *get data* method an asynchronous signature.
- You discovered `Observable` and the RxJS *Observable* library.
- You used RxJS `of()` to return an observable of mock heroes (`Observable<Hero[]>`).
- The component's `ngOnInit` lifecycle hook calls the `HeroService` method, not the constructor.
- You created a `MessageService` for loosely-coupled communication between classes.
- The `HeroService` injected into a component is created with another injected service, `MessageService` .

© 2010–2018 Google, Inc.
Licensed under the Creative Commons Attribution License 4.0.
https://angular.io/tutorial/toh-pt4

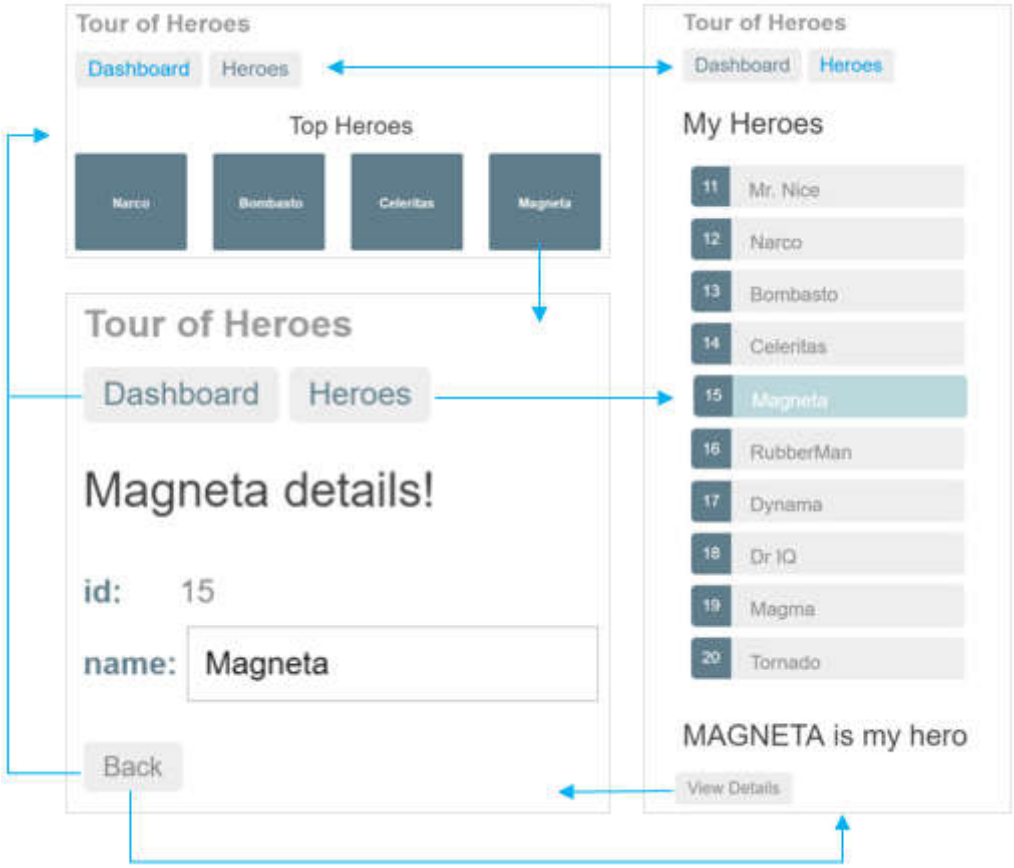
Exported from DevDocs — https://devdocs.io

Routing

There are new requirements for the Tour of Heroes app:

- Add a *Dashboard* view.
- Add the ability to navigate between the *Heroes* and *Dashboard* views.
- When users click a hero name in either view, navigate to a detail view of the selected hero.
- When users click a *deep link* in an email, open the detail view for a particular hero.

When you're done, users will be able to navigate the app like this:



Add the `AppRoutingModule`

An Angular best practice is to load and configure the router in a separate, top-level module that is dedicated to routing and imported by the root `AppModule`.

By convention, the module class name is `AppRoutingModule` and it belongs in the `app-routing.module.ts` in the `src/app` folder.

Use the CLI to generate it.

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

`--flat` puts the file in `src/app` instead of its own folder. `--module=app` tells the CLI to register it in the `imports` array of the `AppModule`.

The generated file looks like this:

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

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

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

You generally don't declare components in a routing module so you can delete the `@NgModule.declarations` array and delete `CommonModule` references too.

You'll configure the router with `Routes` in the `RouterModule` so import those two symbols from the `@angular/router` library.

Add an `@NgModule.exports` array with `RouterModule` in it. Exporting `RouterModule` makes router directives available for use in the `AppModule` components that will need them.

`AppRoutingModule` looks like this now:

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

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

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

Add routes

Routes tell the router which view to display when a user clicks a link or pastes a URL into the browser address bar.

A typical Angular `Route` has two properties:

1. `path` : a string that matches the URL in the browser address bar.
2. `component` : the component that the router should create when navigating to this route.

You intend to navigate to the `HeroesComponent` when the URL is something like `localhost:4200/heroes` .

Import the `HeroesComponent` so you can reference it in a `Route` . Then define an array of routes with a single `route` to that component.

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

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

Once you've finished setting up, the router will match that URL to `path: 'heroes'` and display the `HeroesComponent`.

`RouterModule.forRoot()`

You first must initialize the router and start it listening for browser location changes.

Add `RouterModule` to the `@NgModule.imports` array and configure it with the `routes` in one step by calling `RouterModule.forRoot()` *within* the `imports` array, like this:

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

> The method is called `forRoot()` because you configure the router at the application's root level. The `forRoot()` method supplies the service providers and directives needed for routing, and performs the initial navigation based on the current browser URL.

Add `RouterOutlet`

Open the `AppComponent` template replace the `<app-heroes>` element with a `<router-outlet>` element.

src/app/app.component.html (router-outlet)

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

You removed `<app-heroes>` because you will only display the `HeroesComponent` when the user navigates to it.

The `<router-outlet>` tells the router where to display routed views.

> The `RouterOutlet` is one of the router directives that became available to the `AppComponent` because `AppModule` imports `AppRoutingModule` which exported `RouterModule`.

Try it

You should still be running with this CLI command.

```
ng serve
```

The browser should refresh and display the app title but not the list of heroes.

Look at the browser's address bar. The URL ends in `/`. The route path to `HeroesComponent` is `/heroes`.

Append `/heroes` to the URL in the browser address bar. You should see the familiar heroes master/detail view.

Add a navigation link (`routerLink`)

Users shouldn't have to paste a route URL into the address bar. They should be able to click a link to navigate.

Add a `<nav>` element and, within that, an anchor element that, when clicked, triggers navigation to the `HeroesComponent`. The revised `AppComponent` template looks like this:

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>
```

A `routerLink` attribute is set to `"/heroes"`, the string that the router matches to the route to `HeroesComponent`. The `routerLink` is the selector for the `RouterLink` directive that turns user clicks into router navigations. It's another of the public directives in the `RouterModule`.

The browser refreshes and displays the app title and heroes link, but not the heroes list.

Click the link. The address bar updates to `/heroes` and the list of heroes appears.

> Make this and future navigation links look better by adding private CSS styles to `app.component.css` as listed in the final code review below.

Add a dashboard view

Routing makes more sense when there are multiple views. So far there's only the heroes view.

Add a `DashboardComponent` using the CLI:

```
ng generate component dashboard
```

The CLI generates the files for the `DashboardComponent` and declares it in `AppModule`.

Replace the default file content in these three files as follows and then return for a little discussion:

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">
    <div class="module hero">
      <h4>{{hero.name}}</h4>
    </div>
  </a>
</div>
```

src/app/dashboard/dashboard.component.ts

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

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

  constructor(private heroService: HeroService) { }

  ngOnInit() {
    this.getHeroes();
  }

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

src/app/dashboard/dashboard.component.css

```
/* DashboardComponent's private CSS styles */
[class*='col-'] {
  float: left;
  padding-right: 20px;
  padding-bottom: 20px;
}
[class*='col-']:last-of-type {
  padding-right: 0;
}
a {
  text-decoration: none;
}
*, *:after, *:before {
  -webkit-box-sizing: border-box;
  -moz-box-sizing: border-box;
  box-sizing: border-box;
}
h3 {
  text-align: center; margin-bottom: 0;
}
h4 {
  position: relative;
}
.grid {
  margin: 0;
}
.col-1-4 {
  width: 25%;
}
.module {
  padding: 20px;
  text-align: center;
```

```
  color: #eee;
  max-height: 120px;
  min-width: 120px;
  background-color: #607D8B;
  border-radius: 2px;
}
.module:hover {
  background-color: #EEE;
  cursor: pointer;
  color: #607d8b;
}
.grid-pad {
  padding: 10px 0;
}
.grid-pad > [class*='col-']:last-of-type {
  padding-right: 20px;
}
@media (max-width: 600px) {
  .module {
    font-size: 10px;
    max-height: 75px; }
}
@media (max-width: 1024px) {
  .grid {
    margin: 0;
  }
  .module {
    min-width: 60px;
  }
}
```

The *template* presents a grid of hero name links.

- The `*ngFor` repeater creates as many links as are in the component's `heroes` array.
- The links are styled as colored blocks by the `dashboard.component.css`.
- The links don't go anywhere yet but they will shortly.

The *class* is similar to the `HeroesComponent` class.

- It defines a `heroes` array property.
- The constructor expects Angular to inject the `HeroService` into a private `heroService` property.
- The `ngOnInit()` lifecycle hook calls `getHeroes`.

This `getHeroes` reduces the number of heroes displayed to four (2nd, 3rd, 4th, and 5th).

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

Add the dashboard route

To navigate to the dashboard, the router needs an appropriate route.

Import the `DashboardComponent` in the `AppRoutingModule`.

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

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

Add a route to the `AppRoutingModule.routes` array that matches a path to the `DashboardComponent`.

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

Add a default route

When the app starts, the browsers address bar points to the web site's root. That doesn't match any existing route so the router doesn't navigate anywhere. The space below the `<router-outlet>` is blank.

To make the app navigate to the dashboard automatically, add the following route to the `AppRoutingModule.Routes` array.

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

This route redirects a URL that fully matches the empty path to the route whose path is `'/dashboard'`.

After the browser refreshes, the router loads the `DashboardComponent` and the browser address bar shows the `/dashboard` URL.

Add dashboard link to the shell

The user should be able to navigate back and forth between the `DashboardComponent` and the `HeroesComponent` by clicking links in the navigation area near the top of the page.

Add a dashboard navigation link to the `AppComponent` shell template, just above the *Heroes* link.

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>
```

After the browser refreshes you can navigate freely between the two views by clicking the links.

Navigating to hero details

The `HeroDetailsComponent` displays details of a selected hero. At the moment the `HeroDetailsComponent` is only visible at the bottom of the `HeroesComponent`

The user should be able to get to these details in three ways.

1. By clicking a hero in the dashboard.
2. By clicking a hero in the heroes list.

3. By pasting a "deep link" URL into the browser address bar that identifies the hero to display.

In this section, you'll enable navigation to the `HeroDetailsComponent` and liberate it from the `HeroesComponent`.

Delete `hero details` from `HeroesComponent`

When the user clicks a hero item in the `HeroesComponent`, the app should navigate to the `HeroDetailComponent`, replacing the heroes list view with the hero detail view. The heroes list view should no longer show hero details as it does now.

Open the `HeroesComponent` template (`heroes/heroes.component.html`) and delete the `<app-hero-detail>` element from the bottom.

Clicking a hero item now does nothing. You'll fix that shortly after you enable routing to the `HeroDetailComponent`.

Add a `hero detail` route

A URL like `~/detail/11` would be a good URL for navigating to the *Hero Detail* view of the hero whose `id` is `11`.

Open `AppRoutingModule` and import `HeroDetailComponent`.

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

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

Then add a *parameterized* route to the `AppRoutingModule.routes` array that matches the path pattern to the *hero detail* view.

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

The colon (:) in the `path` indicates that `:id` is a placeholder for a specific hero `id`.

At this point, all application routes are in place.

src/app/app-routing.module.ts (all routes)

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

`DashboardComponent` hero links

The `DashboardComponent` hero links do nothing at the moment.

Now that the router has a route to `HeroDetailComponent`, fix the dashboard hero links to navigate via the *parameterized* dashboard route.

src/app/dashboard/dashboard.component.html (hero links)

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

You're using Angular interpolation binding within the `*ngFor` repeater to insert the current iteration's `hero.id` into each `routerLink`.

`HeroesComponent` hero links

The hero items in the `HeroesComponent` are `<li>` elements whose click events are bound to the component's `onSelect()` method.

src/app/heroes/heroes.component.html (list with 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>
```

Strip the `<li>` back to just its `*ngFor`, wrap the badge and name in an anchor element (`<a>`), and add a `routerLink` attribute to the anchor that is the same as in the dashboard template

src/app/heroes/heroes.component.html (list with links)

```
<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>
```

You'll have to fix the private stylesheet (`heroes.component.css`) to make the list look as it did before. Revised styles are in the final code review at the bottom of this guide.

Remove dead code (optional)

While the `HeroesComponent` class still works, the `onSelect()` method and `selectedHero` property are no longer used.

It's nice to tidy up and you'll be grateful to yourself later. Here's the class after pruning away the dead code.

src/app/heroes/heroes.component.ts (cleaned up)

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

  constructor(private heroService: HeroService) { }

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

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

Routable `HeroDetailComponent`

Previously, the parent `HeroesComponent` set the `HeroDetailComponent.hero` property and the `HeroDetailComponent` displayed the hero.

`HeroesComponent` doesn't do that anymore. Now the router creates the `HeroDetailComponent` in response to a URL such as `~/detail/11`.

The `HeroDetailComponent` needs a new way to obtain the *hero-to-display*.

- Get the route that created it,
- Extract the `id` from the route
- Acquire the hero with that `id` from the server via the `HeroService`

Add the following imports:

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

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

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

Inject the `ActivatedRoute`, `HeroService`, and `Location` services into the constructor, saving their values in private fields:

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

The `ActivatedRoute` holds information about the route to this instance of the `HeroDetailComponent`. This component is interested in the route's bag of parameters extracted from the URL. The "*id*" parameter is the `id` of the hero to display.

The `HeroService` gets hero data from the remote server and this component will use it to get the *hero-to-display*.

The `location` is an Angular service for interacting with the browser. You'll use it later to navigate back to the view that navigated here.

Extract the `id` route parameter

In the `ngOnInit()` lifecycle hook call `getHero()` and define it as follows.

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

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

The `route.snapshot` is a static image of the route information shortly after the component was created.

The `paramMap` is a dictionary of route parameter values extracted from the URL. The `"id"` key returns the `id` of the hero to fetch.

Route parameters are always strings. The JavaScript (+) operator converts the string to a number, which is what a hero `id` should be.

The browser refreshes and the app crashes with a compiler error. `HeroService` doesn't have a `getHero()` method. Add it now.

Add `HeroService.getHero()`

Open `HeroService` and add this `getHero()` method

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));
}
```

Note the backticks (`) that define a JavaScript *template literal* for embedding the `id`.

Like `getHeroes()`, `getHero()` has an asynchronous signature. It returns a *mock hero* as an `Observable`, using the RxJS `of()` function.

You'll be able to re-implement `getHero()` as a real `Http` request without having to change the `HeroDetailComponent` that calls it.

Try it

The browser refreshes and the app is working again. You can click a hero in the dashboard or in the heroes list and navigate to that hero's detail view.

If you paste `localhost:4200/detail/11` in the browser address bar, the router navigates to the detail view for the hero with `id: 11`, "Mr. Nice".

Find the way back

By clicking the browser's back button, you can go back to the hero list or dashboard view, depending upon which sent you to the detail view.

It would be nice to have a button on the `HeroDetail` view that can do that.

Add a *go back* button to the bottom of the component template and bind it to the component's `goBack()` method.

src/app/hero-detail/hero-detail.component.html (back button)

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

Add a `goBack()` *method* to the component class that navigates backward one step in the browser's history stack using the `Location` service that you injected previously.

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

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

Refresh the browser and start clicking. Users can navigate around the app, from the dashboard to hero details and back, from heroes list to the mini detail to the hero details and back to the heroes again.

You've met all of the navigational requirements that propelled this page.

Final code review

Here are the code files discussed on this page and your app should look like this live example.

AppRoutingModule, ***AppModule***, and ***HeroService***

src/app/app-routing.module.ts

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

import { DashboardComponent }   from './dashboard/dashboard.component';
import { HeroesComponent }      from './heroes/heroes.component';
import { HeroDetailComponent }  from './hero-detail/hero-detail.component';

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

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

src/app/app.module.ts

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

import { AppComponent }         from './app.component';
import { DashboardComponent }   from './dashboard/dashboard.component';
import { HeroDetailComponent }  from './hero-detail/hero-detail.component';
import { HeroesComponent }      from './heroes/heroes.component';
import { MessagesComponent }    from './messages/messages.component';

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

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    DashboardComponent,
    HeroesComponent,
    HeroDetailComponent,
    MessagesComponent
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

src/app/hero.service.ts

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

import { Observable, of } from 'rxjs';

import { Hero } from './hero';
import { HEROES } from './mock-heroes';
import { MessageService } from './message.service';

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

  constructor(private messageService: MessageService) { }

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

  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));
  }
}
```

AppComponent

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>
```

src/app/app.component.css

```
/* AppComponent's private CSS styles */
h1 {
  font-size: 1.2em;
  color: #999;
  margin-bottom: 0;
}
h2 {
  font-size: 2em;
  margin-top: 0;
  padding-top: 0;
}
nav a {
  padding: 5px 10px;
  text-decoration: none;
  margin-top: 10px;
  display: inline-block;
  background-color: #eee;
  border-radius: 4px;
}
nav a:visited, a:link {
  color: #607D8B;
}
nav a:hover {
  color: #039be5;
  background-color: #CFD8DC;
}
nav a.active {
  color: #039be5;
}
```

DashboardComponent

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>
```

src/app/dashboard/dashboard.component.ts

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

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

  constructor(private heroService: HeroService) { }

  ngOnInit() {
    this.getHeroes();
  }

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

src/app/dashboard/dashboard.component.css

```
/* DashboardComponent's private CSS styles */
[class*='col-'] {
  float: left;
  padding-right: 20px;
  padding-bottom: 20px;
}
[class*='col-']:last-of-type {
  padding-right: 0;
}
a {
  text-decoration: none;
}
*, *:after, *:before {
  -webkit-box-sizing: border-box;
  -moz-box-sizing: border-box;
  box-sizing: border-box;
}
h3 {
  text-align: center; margin-bottom: 0;
}
h4 {
  position: relative;
}
.grid {
  margin: 0;
}
.col-1-4 {
  width: 25%;
}
.module {
  padding: 20px;
  text-align: center;
  color: #eee;
  max-height: 120px;
```

```
  min-width: 120px;
  background-color: #607D8B;
  border-radius: 2px;
}
.module:hover {
  background-color: #EEE;
  cursor: pointer;
  color: #607d8b;
}
.grid-pad {
  padding: 10px 0;
}
.grid-pad > [class*='col-']:last-of-type {
  padding-right: 20px;
}
@media (max-width: 600px) {
  .module {
    font-size: 10px;
    max-height: 75px; }
}
@media (max-width: 1024px) {
  .grid {
    margin: 0;
  }
  .module {
    min-width: 60px;
  }
}
```

HeroesComponent

src/app/heroes/heroes.component.html

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

src/app/heroes/heroes.component.ts

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

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

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

  constructor(private heroService: HeroService) { }

  ngOnInit() {
```

```
    this.getHeroes();
  }

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

src/app/heroes/heroes.component.css

```
/* HeroesComponent's private CSS styles */
.heroes {
  margin: 0 0 2em 0;
  list-style-type: none;
  padding: 0;
  width: 15em;
}
.heroes li {
  position: relative;
  cursor: pointer;
  background-color: #EEE;
  margin: .5em;
  padding: .3em 0;
  height: 1.6em;
  border-radius: 4px;
}

.heroes li:hover {
  color: #607D8B;
  background-color: #DDD;
  left: .1em;
}

.heroes a {
  color: #888;
  text-decoration: none;
  position: relative;
  display: block;
  width: 250px;
}

.heroes a:hover {
  color:#607D8B;
}

.heroes .badge {
  display: inline-block;
  font-size: small;
  color: white;
  padding: 0.8em 0.7em 0 0.7em;
  background-color: #607D8B;
  line-height: 1em;
  position: relative;
  left: -1px;
  top: -4px;
  height: 1.8em;
```

```
  min-width: 16px;
  text-align: right;
  margin-right: .8em;
  border-radius: 4px 0 0 4px;
}
```

HeroDetailComponent

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

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

```
import { Component, OnInit, Input } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Location } from '@angular/common';

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

@Component({
  selector: 'app-hero-detail',
  templateUrl: './hero-detail.component.html',
  styleUrls: [ './hero-detail.component.css' ]
})
export class HeroDetailComponent implements OnInit {
  @Input() hero: Hero;

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

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

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

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

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

```
/* HeroDetailComponent's private CSS styles */
label {
  display: inline-block;
  width: 3em;
  margin: .5em 0;
  color: #607D8B;
  font-weight: bold;
}
input {
  height: 2em;
  font-size: 1em;
  padding-left: .4em;
}
button {
  margin-top: 20px;
  font-family: Arial;
  background-color: #eee;
  border: none;
  padding: 5px 10px;
  border-radius: 4px;
  cursor: pointer; cursor: hand;
}
button:hover {
  background-color: #cfd8dc;
}
button:disabled {
  background-color: #eee;
  color: #ccc;
  cursor: auto;
}
```

Summary

- You added the Angular router to navigate among different components.
- You turned the `AppComponent` into a navigation shell with `<a>` links and a `<router-outlet>`.
- You configured the router in an `AppRoutingModule`
- You defined simple routes, a redirect route, and a parameterized route.
- You used the `routerLink` directive in anchor elements.
- You refactored a tightly-coupled master/detail view into a routed detail view.
- You used router link parameters to navigate to the detail view of a user-selected hero.
- You shared the `HeroService` among multiple components.

© 2010–2018 Google, Inc.
Licensed under the Creative Commons Attribution License 4.0.
https://angular.io/tutorial/toh-pt5

Exported from DevDocs — https://devdocs.io

HTTP

In this tutorial, you'll add the following data persistence features with help from Angular's `HttpClient`.

- The `HeroService` gets hero data with HTTP requests.
- Users can add, edit, and delete heroes and save these changes over HTTP.
- Users can search for heroes by name.

When you're done with this page, the app should look like this live example.

Enable HTTP services

`HttpClient` is Angular's mechanism for communicating with a remote server over HTTP.

To make `HttpClient` available everywhere in the app,

- open the root `AppModule`,
- import the `HttpClientModule` symbol from `@angular/common/http`,
- add it to the `@NgModule.imports` array.

Simulate a data server

This tutorial sample *mimics* communication with a remote data server by using the *In-memory Web API* module.

After installing the module, the app will make requests to and receive responses from the `HttpClient` without knowing that the *In-memory Web API* is intercepting those requests, applying them to an in-memory data store, and returning simulated responses.

This facility is a great convenience for the tutorial. You won't have to set up a server to learn about `HttpClient`.

It may also be convenient in the early stages of your own app development when the server's web api is ill-defined or not yet implemented.

Important: the *In-memory Web API* module has nothing to do with HTTP in Angular.

If you're just *reading* this tutorial to learn about `HttpClient`, you can skip over this step. If you're *coding along* with this tutorial, stay here and add the *In-memory Web API* now.

Install the *In-memory Web API* package from *npm*

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

Import the `HttpClientInMemoryWebApiModule` and the `InMemoryDataService` class, which you will create in a moment.

src/app/app.module.ts (In-memory Web API imports)

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

Add the `HttpClientInMemoryWebApiModule` to the `@NgModule.imports` array— *after importing the* `HttpClient`, — while configuring it with the `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 }
)
```

The `forRoot()` configuration method takes an `InMemoryDataService` class that primes the in-memory database.

The *Tour of Heroes* sample creates such a class `src/app/in-memory-data.service.ts` which has the following content:

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

```
import { InMemoryDbService } from 'angular-in-memory-web-api';

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};
  }
}
```

This file replaces `mock-heroes.ts`, which is now safe to delete.

When your server is ready, detach the *In-memory Web API*, and the app's requests will go through to the server.

Now back to the `HttpClient` story.

Heroes and HTTP

Import some HTTP symbols that you'll need:

src/app/hero.service.ts (import HTTP symbols)

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

Inject `HttpClient` into the constructor in a private property called `http`.

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

Keep injecting the `MessageService`. You'll call it so frequently that you'll wrap it in private `log` method.

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

Define the `heroesUrl` with the address of the heroes resource on the server.

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

Get heroes with `HttpClient`

The current `HeroService.getHeroes()` uses the RxJS `of()` function to return an array of mock heroes as an `Observable<Hero[]>`.

src/app/hero.service.ts (getHeroes with RxJs 'of()')

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

Convert that method to use `HttpClient`

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

Refresh the browser. The hero data should successfully load from the mock server.

You've swapped `of` for `http.get` and the app keeps working without any other changes because both functions return an `Observable<Hero[]>`.

Http methods return one value

All `HttpClient` methods return an RxJS `Observable` of something.

HTTP is a request/response protocol. You make a request, it returns a single response.

In general, an observable *can* return multiple values over time. An observable from `HttpClient` always emits a single value and then completes, never to emit again.

This particular `HttpClient.get` call returns an `Observable<Hero[]>`, literally "*an observable of hero arrays*". In practice, it will only return a single hero array.

`HttpClient.get` returns response data

`HttpClient.get` returns the *body* of the response as an untyped JSON object by default. Applying the optional type specifier, `<Hero[]>` , gives you a typed result object.

The shape of the JSON data is determined by the server's data API. The *Tour of Heroes* data API returns the hero data as an array.

> Other APIs may bury the data that you want within an object. You might have to dig that data out by processing the `Observable` result with the RxJS `map` operator.
>
> Although not discussed here, there's an example of `map` in the `getHeroNo404()` method included in the sample source code.

Error handling

Things go wrong, especially when you're getting data from a remote server. The `HeroService.getHeroes()` method should catch errors and do something appropriate.

To catch errors, you **"pipe" the observable** result from `http.get()` through an RxJS `catchError()` operator.

Import the `catchError` symbol from `rxjs/operators` , along with some other operators you'll need later.

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

Now extend the observable result with the `.pipe()` method and give it a `catchError()` operator.

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

The `catchError()` operator intercepts an `Observable` **that failed**. It passes the error an *error handler* that can do what it wants with the error.

The following `handleError()` method reports the error and then returns an innocuous result so that the application keeps working.

handleError

The following `errorHandler()` will be shared by many `HeroService` methods so it's generalized to meet their different needs.

Instead of handling the error directly, it returns an *error handler* function to `catchError` that it has configured with both the name of the operation that failed and a safe return value.

```
/**
 * 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);
  };
}
```

After reporting the error to console, the handler constructs a user friendly message and returns a safe value to the app so it can keep working.

Because each service method returns a different kind of `Observable` result, `errorHandler()` takes a type parameter so it can return the safe value as the type that the app expects.

Tap into the `Observable`

The `HeroService` methods will **tap** into the flow of observable values and send a message (via `log()`) to the message area at the bottom of the page.

They'll do that with the RxJS `tap` operator, which *looks* at the observable values, does *something* with those values, and passes them along. The `tap` call back doesn't touch the values themselves.

Here is the final version of `getHeroes` with the `tap` that logs the operation.

```
/** 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', []))
    );
}
```

Get hero by id

Most web APIs support a *get by id* request in the form `api/hero/:id` (such as `api/hero/11`). Add a `HeroService.getHero()` method to make that request:

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}`))
  );
}
```

There are three significant differences from `getHeroes()`.

- it constructs a request URL with the desired hero's id.
- the server should respond with a single hero rather than an array of heroes.
- therefore, `getHero` returns an `Observable<Hero>` ("*an observable of Hero objects*") rather than an observable of hero *arrays* .

Update heroes

Editing a hero's name in the *hero detail* view. As you type, the hero name updates the heading at the top of the page. But when you click the "go back button", the changes are lost.

If you want changes to persist, you must write them back to the server.

At the end of the hero detail template, add a save button with a `click` event binding that invokes a new component method named `save()`.

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

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

Add the following `save()` method, which persists hero name changes using the hero service `updateHero()` method and then navigates back to the previous view.

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

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

Add *HeroService.updateHero()*

The overall structure of the `updateHero()` method is similar to that of `getHeroes()`, but it uses `http.put()` to persist the changed hero on the server.

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

```
/** 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'))
  );
}
```

The `HttpClient.put()` method takes three parameters

- the URL
- the data to update (the modified hero in this case)
- options

The URL is unchanged. The heroes web API knows which hero to update by looking at the hero's `id`.

The heroes web API expects a special header in HTTP save requests. That header is in the `httpOptions` constant defined in the `HeroService`.

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

Refresh the browser, change a hero name, save your change, and click the "go back" button. The hero now appears in the list with the changed name.

Add a new hero

To add a hero, this app only needs the hero's name. You can use an `input` element paired with an add button.

Insert the following into the `HeroesComponent` template, just after the heading:

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

```
<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>
```

In response to a click event, call the component's click handler and then clear the input field so that it's ready for another name.

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

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

When the given name is non-blank, the handler creates a `Hero`-like object from the name (it's only missing the `id`) and passes it to the services `addHero()` method.

When `addHero` saves successfully, the `subscribe` callback receives the new hero and pushes it into to the `heroes` list for display.

You'll write `HeroService.addHero` in the next section.

Add *HeroService.addHero()*

Add the following `addHero()` method to the `HeroService` class.

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()` differs from `updateHero` in two ways.

- it calls `HttpClient.post()` instead of `put()`.
- it expects the server to generates an id for the new hero, which it returns in the `Observable<Hero>` to the caller.

Refresh the browser and add some heroes.

Delete a hero

Each hero in the heroes list should have a delete button.

Add the following button element to the `HeroesComponent` template, after the hero name in the repeated `<li>` element.

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

The HTML for the list of heroes should look like this:

src/app/heroes/heroes.component.html (list of heroes)

```
<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>
```

To position the delete button at the far right of the hero entry, add some CSS to the `heroes.component.css`. You'll find that CSS in the final review code below.

Add the `delete()` handler to the component.

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

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

Although the component delegates hero deletion to the `HeroService`, it remains responsible for updating its own list of heroes. The component's `delete()` method immediately removes the *hero-to-delete* from that list, anticipating that the `HeroService` will succeed on the server.

There's really nothing for the component to do with the `Observable` returned by `heroService.delete()`. **It must subscribe anyway**.

> If you neglect to `subscribe()`, the service will not send the delete request to the server! As a rule, an `Observable` *does nothing* until something subscribes!
>
> Confirm this for yourself by temporarily removing the `subscribe()`, clicking "Dashboard", then clicking "Heroes". You'll see the full list of heroes again.

Add *HeroService.deleteHero()*

Add a `deleteHero()` method to `HeroService` like this.

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

```
/** 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'))
  );
}
```

Note that

- it calls `HttpClient.delete`.
- the URL is the heroes resource URL plus the `id` of the hero to delete
- you don't send data as you did with `put` and `post`.
- you still send the `httpOptions`.

Refresh the browser and try the new delete functionality.

Search by name

In this last exercise, you learn to chain `Observable` operators together so you can minimize the number of similar HTTP requests and consume network bandwidth economically.

You will add a *heroes search* feature to the *Dashboard*. As the user types a name into a search box, you'll make repeated HTTP requests for heroes filtered by that name. Your goal is to issue only as many requests as necessary.

HeroService.searchHeroes

Start by adding a `searchHeroes` method to the `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', []))
  );
}
```

The method returns immediately with an empty array if there is no search term. The rest of it closely resembles `getHeroes()`. The only significant difference is the URL, which includes a query string with the search term.

Add search to the Dashboard

Open the `DashboardComponent` *template* and Add the hero search element, `<app-hero-search>`, to the bottom of the `DashboardComponent` template.

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>
```

This template looks a lot like the `*ngFor` repeater in the `HeroesComponent` template.

Unfortunately, adding this element breaks the app. Angular can't find a component with a selector that matches `<app-hero-search>`.

The `HeroSearchComponent` doesn't exist yet. Fix that.

Create `HeroSearchComponent`

Create a `HeroSearchComponent` with the CLI.

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

The CLI generates the three `HeroSearchComponent` and adds the component to the `AppModule' declarations

Replace the generated `HeroSearchComponent` *template* with a text box and a list of matching search results like this.

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>
```

Add private CSS styles to `hero-search.component.css` as listed in the final code review below.

As the user types in the search box, a *keyup* event binding calls the component's `search()` method with the new search box value.

`AsyncPipe`

As expected, the `*ngFor` repeats hero objects.

Look closely and you'll see that the `*ngFor` iterates over a list called `heroes$`, not `heroes`.

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

The `$` is a convention that indicates `heroes$` is an `Observable`, not an array.

The `*ngFor` can't do anything with an `Observable`. But there's also a pipe character (`|`) followed by `async`, which identifies Angular's `AsyncPipe`.

The `AsyncPipe` subscribes to an `Observable` automatically so you won't have to do so in the component class.

Fix the `HeroSearchComponent` class

Replace the generated `HeroSearchComponent` class and metadata as follows.

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)),
    );
  }
}
```

Notice the declaration of `heroes$` as an `Observable`

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

You'll set it in `ngOnInit()`. Before you do, focus on the definition of `searchTerms`.

The `searchTerms` RxJS subject

The `searchTerms` property is declared as an 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` is both a source of *observable* values and an `Observable` itself. You can subscribe to a `Subject` as you would any `Observable`.

You can also push values into that `Observable` by calling its `next(value)` method as the `search()` method does.

The `search()` method is called via an *event binding* to the textbox's `keystroke` event.

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

Every time the user types in the textbox, the binding calls `search()` with the textbox value, a "search term". The `searchTerms` becomes an `Observable` emitting a steady stream of search terms.

Chaining RxJS operators

Passing a new search term directly to the `searchHeroes()` after every user keystroke would create an excessive amount of HTTP requests, taxing server resources and burning through the cellular network data plan.

Instead, the `ngOnInit()` method pipes the `searchTerms` observable through a sequence of RxJS operators that reduce the number of calls to the `searchHeroes()`, ultimately returning an observable of timely hero search results (each a `Hero[]`).

Here's the code.

```
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)` waits until the flow of new string events pauses for 300 milliseconds before passing along the latest string. You'll never make requests more frequently than 300ms.
- `distinctUntilChanged()` ensures that a request is sent only if the filter text changed.
- `switchMap()` calls the search service for each search term that makes it through `debounce` and `distinctUntilChanged`. It cancels and discards previous search observables, returning only the latest search service observable.

With the switchMap operator, every qualifying key event can trigger an `HttpClient.get()` method call. Even with a 300ms pause between requests, you could have multiple HTTP requests in flight and they may not return in the order sent.

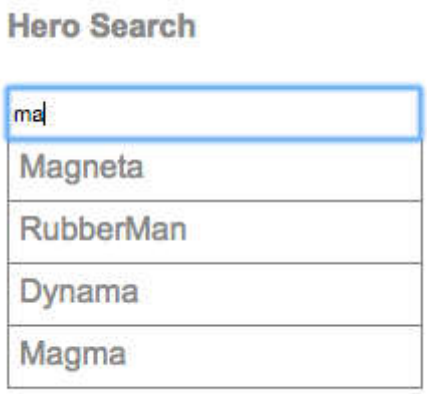
`switchMap()` preserves the original request order while returning only the observable from the most recent HTTP method call. Results from prior calls are canceled and discarded.

Note that *canceling* a previous `searchHeroes()` *Observable* doesn't actually abort a pending HTTP request. Unwanted results are simply discarded before they reach your application code.

Remember that the component *class* does not subscribe to the `heroes$` *observable*. That's the job of the `AsyncPipe` in the template.

Try it

Run the app again. In the *Dashboard*, enter some text in the search box. If you enter characters that match any existing hero names, you'll see something like this.



Final code review

Your app should look like this live example.

Here are the code files discussed on this page (all in the `src/app/` folder).

HeroService, InMemoryDataService, AppModule

hero.service.ts

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

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

import { Hero } from './hero';
import { MessageService } from './message.service';

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

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

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

  constructor(
```

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

  /** 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', []))
      );
  }

  /** GET hero by id. Return `undefined` when id not found */
  getHeroNo404<Data>(id: number): Observable<Hero> {
    const url = `${this.heroesUrl}/?id=${id}`;
    return this.http.get<Hero[]>(url)
      .pipe(
        map(heroes => heroes[0]), // returns a {0|1} element array
        tap(h => {
          const outcome = h ? `fetched` : `did not find`;
          this.log(`${outcome} hero id=${id}`);
        }),
        catchError(this.handleError<Hero>(`getHero id=${id}`))
      );
  }

  /** 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}`))
    );
  }

  /* 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', []))
    );
  }

  //////// Save methods //////////

  /** 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'))
    );
  }

  /** 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'))
    );
  }

  /** 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'))
    );
  }

  /**
   * 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);
    };
  }

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

in-memory-data.service.ts

```
import { InMemoryDbService } from 'angular-in-memory-web-api';

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};
  }
}
```

app.module.ts

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

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

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

import { AppComponent }         from './app.component';
import { DashboardComponent }   from './dashboard/dashboard.component';
import { HeroDetailComponent }  from './hero-detail/hero-detail.component';
import { HeroesComponent }      from './heroes/heroes.component';
import { HeroSearchComponent }  from './hero-search/hero-search.component';
import { MessagesComponent }    from './messages/messages.component';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    AppRoutingModule,
    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 }
    )
  ],
  declarations: [
    AppComponent,
    DashboardComponent,
    HeroesComponent,
    HeroDetailComponent,
    MessagesComponent,
    HeroSearchComponent
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

HeroesComponent

heroes/heroes.component.html

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

<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>

<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>
```

heroes/heroes.component.ts

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

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

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

  constructor(private heroService: HeroService) { }

  ngOnInit() {
    this.getHeroes();
  }

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

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

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

}
```

heroes/heroes.component.css

```
/* HeroesComponent's private CSS styles */
.heroes {
  margin: 0 0 2em 0;
  list-style-type: none;
  padding: 0;
  width: 15em;
}
.heroes li {
  position: relative;
  cursor: pointer;
  background-color: #EEE;
  margin: .5em;
  padding: .3em 0;
  height: 1.6em;
  border-radius: 4px;
}

.heroes li:hover {
  color: #607D8B;
  background-color: #DDD;
  left: .1em;
}

.heroes a {
  color: #888;
  text-decoration: none;
  position: relative;
  display: block;
  width: 250px;
}

.heroes a:hover {
  color:#607D8B;
}

.heroes .badge {
  display: inline-block;
  font-size: small;
  color: white;
  padding: 0.8em 0.7em 0 0.7em;
  background-color: #607D8B;
  line-height: 1em;
  position: relative;
  left: -1px;
  top: -4px;
  height: 1.8em;
  min-width: 16px;
  text-align: right;
  margin-right: .8em;
  border-radius: 4px 0 0 4px;
}

button {
  background-color: #eee;
  border: none;
  padding: 5px 10px;
  border-radius: 4px;
  cursor: pointer;
  cursor: hand;
```

```
  font-family: Arial;
}

button:hover {
  background-color: #cfd8dc;
}

button.delete {
  position: relative;
  left: 194px;
  top: -32px;
  background-color: gray !important;
  color: white;
}
```

HeroDetailComponent

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

hero-detail/hero-detail.component.ts

```
import { Component, OnInit, Input } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Location } from '@angular/common';

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

@Component({
  selector: 'app-hero-detail',
  templateUrl: './hero-detail.component.html',
  styleUrls: [ './hero-detail.component.css' ]
})
export class HeroDetailComponent implements OnInit {
  @Input() hero: Hero;

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

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

  getHero(): void {
    const id = +this.route.snapshot.paramMap.get('id');
```

```
    this.heroService.getHero(id)
      .subscribe(hero => this.hero = hero);
  }

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

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

HeroSearchComponent

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>
```

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)),
    );
  }
}
```

hero-search/hero-search.component.css

```
/* HeroSearch private styles */
.search-result li {
  border-bottom: 1px solid gray;
  border-left: 1px solid gray;
  border-right: 1px solid gray;
  width:195px;
  height: 16px;
  padding: 5px;
  background-color: white;
  cursor: pointer;
  list-style-type: none;
}

.search-result li:hover {
  background-color: #607D8B;
}

.search-result li a {
  color: #888;
  display: block;
  text-decoration: none;
}

.search-result li a:hover {
  color: white;
}
.search-result li a:active {
  color: white;
}
#search-box {
  width: 200px;
  height: 20px;
}


ul.search-result {
  margin-top: 0;
  padding-left: 0;
}
```

Summary

You're at the end of your journey, and you've accomplished a lot.

- You added the necessary dependencies to use HTTP in the app.
- You refactored `HeroService` to load heroes from a web API.
- You extended `HeroService` to support `post()`, `put()`, and `delete()` methods.
- You updated the components to allow adding, editing, and deleting of heroes.
- You configured an in-memory web API.
- You learned how to use observables.

This concludes the "Tour of Heroes" tutorial. You're ready to learn more about Angular development in the fundamentals section, starting with the Architecture guide.

© 2010–2018 Google, Inc.
Licensed under the Creative Commons Attribution License 4.0.
https://angular.io/tutorial/toh-pt6

Exported from DevDocs — https://devdocs.io