The hero editor

Contents ›

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.

> For the sample app that this page describes, see the live example / download example.

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` along with a test file.

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 interface

A real hero is more than a name.

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

src/app/hero.ts

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

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

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 displays the hero's information.

Format with the *UppercasePipe*

Modify the `hero.name` binding like this.

src/app/heroes/heroes.component.html

```
<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 flows 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.

src/app/app.module.ts

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

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

src/app/app.module.ts

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

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

Final code review

Here are the code files discussed on this page.

src/app/heroes/heroes.component.ts | src/app/heroes/heroes.component.html | src/app/a

```
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() {
  }

}
```

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.

Version 10.0.10-local+sha.b32126c335.