THE HERO EDITOR

We build a simple hero editor

Once Upon a Time

Every story starts somewhere. Our story starts where the QuickStart ends.

Run the live example for part 1

Follow the "QuickStart" steps. They provide the prerequisites, the folder structure, and the core files for our Tour of Heroes.

Copy the "QuickStart" code to a new folder and rename the folder `angular2-tour-of-heroes`. We should have the following structure:

```
angular2-tour-of-heroes
  app
    app.component.ts
    main.ts
  node_modules ...
  typings ...
  index.html
  package.json
  tsconfig.json
```

```
└ typings.json
```

Keep the app transpiling and running

We want to start the TypeScript compiler, have it watch for changes, and start our server. We'll do this by typing

```
npm start
```

This command runs the compiler in watch mode, starts the server, launches the app in a browser, and keeps the app running while we continue to build the Tour of Heroes.

Show our Hero

We want to display Hero data in our app

Let's add two properties to our `AppComponent`, a `title` property for the application name and a `hero` property for a hero named "Windstorm".

app.component.ts (AppComponent class)

```
export class AppComponent {
  public title = 'Tour of Heroes';
  public hero = 'Windstorm';
}
```

Now we update the template in the `@Component` decoration with data bindings to these new properties.

```
template: '<h1>{{title}}</h1><h2>{{hero}} details!</h2>'
```

The browser should refresh and display our title and hero.

The double curly braces tell our app to read the `title` and `hero` properties from the component and render them. This is the "interpolation" form of one-way

data binding.

> Learn more about interpolation in the Displaying Data chapter.

Hero object

At the moment, our hero is just a name. Our hero needs more properties. Let's convert the `hero` from a literal string to an interface.

Create a `Hero` interface with `id` and `name` properties. Keep this near the top of the `app.component.ts` file for now.

app.component.ts (Hero interface)

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

> **Interface or Class?**
>
> Why a `Hero` interface and not a `Hero` class? We want a strongly typed `Hero`. We get strong typing with either option. Our choice depends on how we intend to use the `Hero`.
>
> If we need a `Hero` that goes beyond simple properties, a `Hero` with logic and behavior, we must define a class. If we only need type checking, the interface is sufficient and lighter weight.
>
> Lighter weight? Transpiling a class to JavaScript produces code. Transpiling an interface produces — nothing. If the class does nothing (and there is nothing for a `Hero` class to do right now), we prefer an interface.

Now that we have a `Hero` interface, let's refactor our component's `hero`

property to be of type `Hero`. Then initialize it with an id of `1` and the name, "Windstorm".

app.component.ts (Hero property)

```
public hero: Hero = {
  id: 1,
  name: 'Windstorm'
};
```

Because we changed the hero from a string to an object, we update the binding in the template to refer to the hero's `name` property.

```
template: '<h1>{{title}}</h1><h2>{{hero.name}} details!</h2>'
```

The browser refreshes and continues to display our hero's name.

Adding more HTML

Displaying a name is good, but we want to see all of our hero's properties. We'll add a `<div>` for our hero's `id` property and another `<div>` for our hero's `name`.

```
template: '<h1>{{title}}</h1><h2>{{hero.name}} details!</h2><div>
<label>id: </label>{{hero.id}}</div><div><label>name: </label>
{{hero.name}}</div>'
```

Uh oh, our template string is getting long. We better take care of that to avoid the risk of making a typo in the template.

Multi-line template strings

We could make a more readable template with string concatenation but that gets ugly fast, it is harder to read, and it is easy to make a spelling error. Instead, let's take advantage of the template strings feature in ES2015 and TypeScript to maintain our sanity.

Change the quotes around the template to back-ticks and put the `<h1>`, `<h2>` and `<div>` elements on their own lines.

app.component.ts (AppComponent's template)

```
template:`
  <h1>{{title}}</h1>
  <h2>{{hero.name}} details!</h2>
  <div><label>id: </label>{{hero.id}}</div>
  <div><label>name: </label>{{hero.name}}</div>
  `
```

Editing Our Hero

We want to be able to edit the hero name in a textbox.

Refactor the hero name `<label>` with `<label>` and `<input>` elements as shown below:

app.component.ts (input element)

```
template:`
  <h1>{{title}}</h1>
  <h2>{{hero.name}} details!</h2>
  <div><label>id: </label>{{hero.id}}</div>
  <div>
    <label>name: </label>
    <div><input value="{{hero.name}}" placeholder="name">
</div>
  </div>
  `
```

We see in the browser that the hero's name does appear in the `<input>` textbox. But something doesn't feel right. When we change the name, we notice that our change is not reflected in the `<h2>`. We won't get the desired behavior with a one-way binding to `<input>`.

Two-Way Binding

We intend to display the name of the hero in the `<input>`, change it, and see

those changes wherever we bind to the hero's name. In short, we want two-way data binding.

Let's update the template to use the `ngModel` built-in directive for two-way binding.

> Learn more about `ngModel` in the Forms and Template Syntax chapters.

Replace the `<input>` with the following HTML

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

The browser refreshes. We see our hero again. We can edit the hero's name and see the changes reflected immediately in the `<h2>`.

The Road We've Travelled

Let's take stock of what we've built.

- Our Tour of Heroes uses the double curly braces of interpolation (a form of one-way data binding) to display the application title and properties of a `Hero` object.
- We wrote a multi-line template using ES2015's template strings to make our template readable.
- We can both display and change the hero's name after adding a two-way data binding to the `<input>` element using the built-in `ngModel` directive.
- The `ngModel` directive also propagates changes to every other binding of the `hero.name`.

Run the live example for part 1

Here's the complete `app.component.ts` as it stands now:

app.component.ts

```
import {Component} from 'angular2/core';

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

@Component({
  selector: 'my-app',
  template:`
    <h1>{{title}}</h1>
    <h2>{{hero.name}} details!</h2>
    <div><label>id: </label>{{hero.id}}</div>
    <div>
      <label>name: </label>
      <div><input [(ngModel)]="hero.name" placeholder="name">
</div>
    </div>
    `
})
export class AppComponent {
  public title = 'Tour of Heroes';
  public hero: Hero = {
    id: 1,
    name: 'Windstorm'
  };
}
```

The Road Ahead

Our Tour of Heroes only displays one hero and we really want to display a list of heroes. We also want to allow the user to select a hero and display their details. We'll learn more about how to retrieve lists, bind them to the template, and allow a user to select it in the next tutorial chapter.

> **Next Step**
>
> Master/Detail