MULTIPLE COMPONENTS

We refactor the master/detail view into separate components

Our app is growing. Use cases are flowing in for reusing components, passing data to components, and creating more reusable assets. Let's separate the heroes list from the hero details and make the details component reusable.

Run the live example for part 3

Where We Left Off

Before we continue with our Tour of Heroes, let’s verify we have the following structure. If not, we’ll need to go back and follow the previous chapters.

```
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 will keep the application running while we continue to build the Tour of Heroes.

Making a Hero Detail Component

Our heroes list and our hero details are in the same component in the same file. They're small now but each could grow. We are sure to receive new requirements for one and not the other. Yet every change puts both components at risk and doubles the testing burden without benefit. If we had to reuse the hero details elsewhere in our app, the heroes list would tag along for the ride.

Our current component violates the Single Responsibility Principle. It's only a tutorial but we can still do things right — especially if doing them right is easy and we learn how to build Angular apps in the process.

Let's break the hero details out into its own component.

Separating the Hero Detail Component

Add a new file named `hero-detail.component.ts` to the `app` folder and create `HeroDetailComponent` as follows.

hero-detail.component.ts (initial version)

```
import {Component} from 'angular2/core';
@Component({
  selector: 'my-hero-detail',
})
export class HeroDetailComponent {
```

```
}
```

> **Naming conventions**
>
> We like to identify at a glance which classes are components and which files contain components.
>
> Notice that we have an `AppComponent` in a file named `app.component.ts` and our new `HeroDetailComponent` is in a file named `hero-detail.component.ts`.
>
> All of our component names end in "Component". All of our component file names end in ".component".
>
> We spell our file names in lower dash case (AKA "kebab case") so we don't worry about case sensitivity on the server or in source control.

We begin by importing the `Component` function from Angular so that we have it handy when we create the metadata for our component.

We create metadata with the `@Component` decorator where we specify the selector name that identifies this component's element. Then we export the class to make it available to other components.

When we finish here, we'll import it into `AppComponent` and refer to its `<my-hero-detail>` element.

Hero Detail Template

At the moment, the *Heroes* and *Hero Detail* views are combined in one template in `AppComponent`. Let's **cut** the *Hero Detail* content from `AppComponent` and **paste** it into the new template property of `HeroDetailComponent`.

We previously bound to the `selectedHero.name` property of the `AppComponent`. Our `HeroDetailComponent` will have a `hero` property, not a `selectedHero` property. So we replace `selectedHero` with `hero` everywhere in our new template. That's our only change. The result looks like this:

hero-detail.component.ts (template)

```
  template: `
    <div *ngIf="hero">
      <h2>{{hero.name}} details!</h2>
      <div><label>id: </label>{{hero.id}}</div>
      <div>
        <label>name: </label>
        <input [(ngModel)]="hero.name" placeholder="name"/>
      </div>
    </div>
  `,
```

Now our hero detail layout exists only in the `HeroDetailComponent`.

Add the *hero* property

Let's add that `hero` property we were talking about to the component class.

```
  hero: Hero;
```

Uh oh. We declared the `hero` property as type `Hero` but our `Hero` interface is over in the `app.component.ts` file. We have two components, each in their own file, that need to reference the `Hero` interface.

We solve the problem by relocating the `Hero` interface from `app.component.ts` to its own `hero.ts` file.

hero.ts (Exported Hero interface)

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

We export the `Hero` interface from `hero.ts` because we'll need to reference it in both component files. Add the following import statement near the top of both `app.component.ts` and `hero-detail.component.ts`.

hero-detail.component.ts and app.component.ts (Import the Hero interface)

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

The *hero* property is an *input*

The `HeroDetailComponent` must be told what hero to display. Who will tell it? The parent `AppComponent`!

The `AppComponent` knows which hero to show: the hero that the user selected from the list. The user's selection is in its `selectedHero` property.

We will soon update the `AppComponent` template so that it binds its `selectedHero` property to the `hero` property of our `HeroDetailComponent`. The binding *might* look like this:

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

Notice that the `hero` property is the ***target*** of a property binding — it's in square brackets to the left of the (=).

Angular insists that we declare a ***target*** property to be an ***input*** property. If we don't, Angular rejects the binding and throws an error.

> We explain input properties in more detail here where we also explain why *target* properties require this special treament and *source* properties do not.

There are a couple of ways we can declare that `hero` is an *input*. We'll do it by adding an `inputs` array to the `@Component` metadata.

```
inputs: ['hero']
```

> Learn about the `@Input()` decorator way in the Attribute Directives chapter.

Refresh the AppComponent

We return to the `AppComponent` and teach it to use the `HeroDetailComponent`.

We begin by importing the `HeroDetailComponent` so we can refer to it.

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

Find the location in the template where we removed the *Hero Detail* content and add an element tag that represents the `HeroDetailComponent`.

```
<my-hero-detail></my-hero-detail>
```

> *my-hero-detail* is the name we set as the `selector` in the `HeroDetailComponent` metadata.

The two components won't coordinate until we bind the `selectedHero` property of the `AppComponent` to the `HeroDetailComponent` element's `hero` property like this:

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

The `AppComponent`'s template should now look like this

app.component.ts (Template)

```
template:`
  <h1>{{title}}</h1>
  <h2>My Heroes</h2>
  <ul class="heroes">
    <li *ngFor="#hero of heroes"
      [class.selected]="hero === selectedHero"
      (click)="onSelect(hero)">
```

```
      <span class="badge">{{hero.id}}</span> {{hero.name}}
    </li>
  </ul>
  <my-hero-detail [hero]="selectedHero"></my-hero-detail>
`,
```

Thanks to the binding, the `HeroDetailComponent` should receive the hero from the `AppComponent` and display that hero's detail beneath the list. The detail should update every time the user picks a new hero.

It's not happening yet!

We click among the heroes. No details. We look for an error in the console of the browser development tools. No error.

It is as if Angular were ignoring the new tag. That's because *it is ignoring the new tag*.

The *directives* array

A browser ignores HTML tags and attributes that it doesn't recognize. So does Angular.

We've imported `HeroDetailComponent`, we've used it in the template, but we haven't told Angular about it.

We tell Angular about it by listing it in the metadata `directives` array. Let's add that array property to the bottom of the `@Component` configuration object, immediately after the `template` and `styles` properties.

```
directives: [HeroDetailComponent]
```

It works!

When we view our app in the browser we see the list of heroes. When we select a hero we can see the selected hero’s details.

What's fundamentally new is that we can use this `HeroDetailComponent` to show hero details anywhere in the app.

We've created our first reusable component!

Reviewing the App Structure

Let's verify that we have the following structure after all of our good refactoring in this chapter:

Here are the code files we discussed in this chapter.

```
import {Component} from 'angular2/core';
import {Hero} from './hero';

@Component({
  selector: 'my-hero-detail',
  template: `
    <div *ngIf="hero">
      <h2>{{hero.name}} details!</h2>
      <div><label>id: </label>{{hero.id}}</div>
      <div>
        <label>name: </label>
        <input [(ngModel)]="hero.name" placeholder="name"/>
      </div>
    </div>
  `,
  inputs: ['hero']
})
```

```
export class HeroDetailComponent {
  hero: Hero;
}
```

The Road We've Travelled

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

- We created a reusable component
- We learned how to make a component accept input
- We learned to bind a parent component to a child component.
- We learned to declare the application directives we need in a `directives` array.

Run the live example for part 3.

The Road Ahead

Our Tour of Heroes has become more reusable with shared components.

We're still getting our (mock) data within the `AppComponent`. That's not sustainable. We should refactor data access to a separate service and share it among the components that need data.

We'll learn to create services in the next tutorial chapter.

> **Next Step**
>
> Services