MASTER/DETAIL

We build a master/detail page with a list of heroes

It Takes Many Heroes

Our story needs more heroes. We'll expand our Tour of Heroes app to display a list of heroes, allow the user to select a hero, and display the hero's details.

Run the live example for part 2

Let's take stock of what we'll need to display a list of heroes. First, we need a list of heroes. We want to display those heroes in the view's template, so we'll need a way to do that.

Where We Left Off

Before we continue with Part 2 of the Tour of Heroes, let's verify we have the following structure after Part 1. If not, we'll need to go back to Part 1 and figure out what we missed.

```
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.

Displaying Our Heroes

Creating heroes

Let's create an array of ten heroes at the bottom of `app.component.ts`.

app.component.ts (Hero array)

```
var 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" }
```

```
12. ];
```

The `HEROES` array is of type `Hero`, the interface defined in part one, to create an array of heroes. We aspire to fetch this list of heroes from a web service, but let's take small steps first and display mock heroes.

Exposing heroes

Let's create a public property in `AppComponent` that exposes the heroes for binding.

app.component.ts (Hero array property)

```
public heroes = HEROES;
```

We did not have to define the `heroes` type. TypeScript can infer it from the `HEROES` array.

> We could have defined the heroes list here in this component class. But we know that ultimately we'll get the heroes from a data service. Because we know where we are heading, it makes sense to separate the hero data from the class implementation from the start.

Displaying heroes in a template

Our component has `heroes`. Let's create an unordered list in our template to display them. We'll insert the following chunk of HTML below the title and above the hero details.

app.component.ts (Heroes template)

```
1. <h2>My Heroes</h2>
2. <ul class="heroes">
3.   <li>
4.     <!-- each hero goes here -->
5.   </li>
6. </ul>
```

Now we have a template that we can fill with our heroes.

Listing heroes with ngFor

We want to bind the array of `heroes` in our component to our template, iterate over them, and display them individually. We'll need some help from Angular to do this. Let's do this step by step.

First modify the `<li>` tag by adding the built-in directive `*ngFor`.

app.component.ts (ngFor)

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

The leading asterisk (`*`) in front of `ngFor` is a critical part of this syntax.

> The (`*`) prefix to `ngFor` indicates that the `<li>` element and its children constitute a master template.
>
> The `ngFor` directive iterates over the `heroes` array returned by the `AppComponent.heroes` property and stamps out instances of this template.
>
> The quoted text assigned to `ngFor` means "*take each hero in the `heroes` array, store it in the local `hero` variable, and make it available to the corresponding template instance*".
>
> The `#` prefix before "hero" identifies the `hero` as a local template variable. We can reference this variable within the template to access a hero's properties.
>
> Learn more about `ngFor` and local template variables in the Displaying Data and Template Syntax chapters.

Now we insert some content between the `<li>` tags that uses the `hero` template variable to display the hero's properties.

app.component.ts (ngFor template)

```
<li *ngFor="#hero of heroes">
  <span class="badge">{{hero.id}}</span> {{hero.name}}
</li>
```

When the browser refreshes, we see a list of heroes!

Styling our heroes

Our list of heroes looks pretty bland. We want to make it visually obvious to a user which hero we are hovering over and which hero is selected.

Let's add some styles to our component by setting the `styles` property on the `@Component` decorator to the following CSS classes:

app.component.ts (Styling)

```
styles:[`
  .selected {
    background-color: #CFD8DC !important;
    color: white;
  }
  .heroes {
    margin: 0 0 2em 0;
    list-style-type: none;
    padding: 0;
    width: 10em;
  }
  .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;
  }
`]
```

Notice that we again use the back-tick notation for multi-line strings.

When we assign styles to a component they are scoped to that specific component. Our styles will only apply to our `AppComponent` and won't "leak" to the outer HTML.

Our template for displaying the heroes should now look like this:

app.component.ts (Styled heroes)

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

That's a lot of styles! We can put them inline as shown here, or we can move them out to their own file which will make it easier to code our component. We'll do this in a later chapter. For now let's keep rolling.

Selecting a Hero

We have a list of heroes and we have a single hero displayed in our app. The list and the single hero are not connected in any way. We want the user to select a hero from our list, and have the selected hero appear in the details view. This UI pattern is widely known as "master-detail". In our case, the master is the heroes list and the detail is the selected hero.

Let's connect the master to the detail through a `selectedHero` component property bound to a click event.

Click event

We modify the `<li>` by inserting an Angular event binding to its click event.

app.component.ts (Capturing the click event)

```
1. <li *ngFor="#hero of heroes" (click)="onSelect(hero)">
2.   <span class="badge">{{hero.id}}</span> {{hero.name}}
3. </li>
```

Focus on the event binding

```
(click)="onSelect(hero)"
```

The parenthesis identify the `<li>` element's `click` event as the target. The expression to the right of the equal sign calls the `AppComponent` method, `onSelect()`, passing the local template variable `hero` as an argument. That's the same `hero` variable we defined previously in the `ngFor`.

> Learn more about Event Binding in the User Input and Templating Syntax chapters.

Add the click handler

Our event binding refers to an `onSelect` method that doesn't exist yet. We'll add that method to our component now.

What should that method do? It should set the component's selected hero to the hero that the user clicked.

Our component doesn't have a "selected hero" yet either. We'll start there.

Expose the selected hero

We no longer need the static `hero` property of the `AppComponent`. **Replace** it with this simple `selectedHero` property:

app.component.ts (selectedHero)

```
selectedHero: Hero;
```

We've decided that none of the heroes should be selected before the user picks a hero so we won't initialize the `selectedHero` as we were doing with `hero`.

Now **add an `onSelect` method** that sets the `selectedHero` property to the `hero` the user clicked.

app.component.ts (onSelect)

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

We will be showing the selected hero's details in our template. At the moment, it is still referring to the old `hero` property. Let's fix the template to bind to the new `selectedHero` property.

app.component.ts (Binding to the selectedHero's name)

```
1. <h2>{{selectedHero.name}} details!</h2>
2. <div><label>id: </label>{{selectedHero.id}}</div>
3. <div>
4.     <label>name: </label>
5.     <input [(ngModel)]="selectedHero.name"
   placeholder="name"/>
6. </div>
```

Hide the empty detail with ngIf

When our app loads we see a list of heroes, but a hero is not selected. The `selectedHero` is `undefined`. That's why we'll see the following error in the browser's console:

```
EXCEPTION: TypeError: Cannot read property 'name' of undefined in
[null]
```

Remember that we are displaying `selectedHero.name` in the template. This name property does not exist because `selectedHero` itself is undefined.

We'll address this problem by keeping the hero detail out of the DOM until there is a selected hero.

We wrap the HTML hero detail content of our template with a `<div>`. Then we add the `ngIf` built-in directive and set it to the `selectedHero` property of our component.

app.component.ts (ngIf)

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

> Remember that the leading asterisk (`*`) in front of `ngIf` is a critical part of this syntax.

When there is no `selectedHero`, the `ngIf` directive removes the hero detail HTML from the DOM. There will be no hero detail elements and no bindings to worry about.

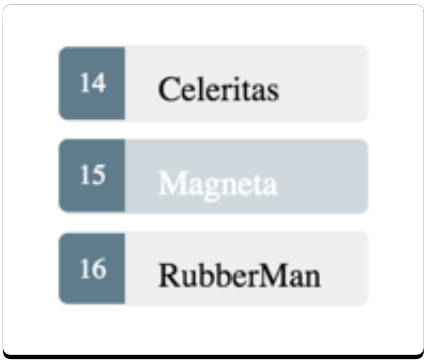
When the user picks a hero, `selectedHero` becomes "truthy" and `ngIf` puts the hero detail content into the DOM and evaluates the nested bindings.

> `ngIf` and `ngFor` are called "structural directives" because they can change the structure of portions of the DOM. In other words, they give structure to the way Angular displays content in the DOM.
>
> Learn more about `ngIf`, `ngFor` and other structural directives in the Structural Directives and Template Syntax chapters.

The browser refreshes and we see the list of heroes but not the selected hero detail. The `ngIf` keeps it out of the DOM as long as the `selectedHero` is undefined. When we click on a hero in the list, the selected hero displays in the hero details. Everything is working as we expect.

Styling the selection

We see the selected hero in the details area below but we can't quickly locate that hero in the list above. We can fix that by applying the `selected` CSS class to the appropriate `<li>` in the master list. For example, when we select Magneta from the heroes list, we can make it pop out visually by giving it a subtle background color as shown here.



We'll add a property binding on `class` for the `selected` class to the template. We'll set this to an expression that compares the current `selectedHero` to the `hero`.

The key is the name of the CSS class (`selected`). The value is `true` if the two heroes match and `false` otherwise. We're saying "*apply the `selected` class if the heroes match, remove it if they don't*".

app.component.ts (Setting the CSS class)

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

Notice in the template that the `class.selected` is surrounded in square brackets (`[]`). This is the syntax for a Property Binding, a binding in which data flows one way from the data source (the expression `hero === selectedHero`) to a property of `class` .

app.component.ts (Styling each hero)

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

> Learn more about Property Binding in the Template Syntax chapter.

The browser reloads our app. We select the hero Magneta and the selection is clearly identified by the background color.

We select a different hero and the tell-tale color switches to that hero.

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>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>
    <div *ngIf="selectedHero">
      <h2>{{selectedHero.name}} details!</h2>
      <div><label>id: </label>{{selectedHero.id}}</div>
      <div>
        <label>name: </label>
        <input [(ngModel)]="selectedHero.name"
placeholder="name"/>
      </div>
    </div>
  `,
  styles:[`
    .selected {
      background-color: #CFD8DC !important;
      color: white;
    }
    .heroes {
      margin: 0 0 2em 0;
      list-style-type: none;
      padding: 0;
      width: 10em;
    }
    .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;
    }
  `]
})
export class AppComponent {
  title = 'Tour of Heroes';
  heroes = HEROES;
  selectedHero: Hero;

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

var 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" }
];
```

The Road We've Travelled

Here's what we achieved in this chapter:

- Our Tour of Heroes now displays a list of selectable heroes

- We added the ability to select a hero and show the hero's details
- We learned how to use the built-in directives `ngIf` and `ngFor` in a component's template

Run the live example for part 2

The Road Ahead

Our Tour of Heroes has grown, but it's far from complete. We can't put the entire app into a single component. We need to break it up into sub-components and teach them to work together as we learn in the next chapter.

> **Next Step**
>
> Multiple Components