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

content\_copyimport { 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)

content\_copyimport { HEROES } from '../mock-heroes';

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

content\_copyheroes = 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)

content\_copy<h2>My Heroes</h2>

<ul class="heroes">

<li>

<span class="badge">{{hero.id}}</span> {{hero.name}}

</li>

</ul>

Now change the <li> to this:

content\_copy<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.styleUrlsarray.

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)

content\_copy@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)

content\_copy<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)

content\_copyselectedHero: 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)

content\_copy<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:

content\_copyHeroesComponent.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)

content\_copy<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

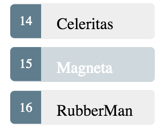
When selectedHero is undefined, the ngIf removes the hero detail from the DOM. There are no selectedHero bindings to worry about.

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)

content\_copy[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)

content\_copy<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

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

src/app/heroes/heroes.component.ts

src/app/heroes/heroes.component.html

src/app/heroes/heroes.component.css

content\_copy

1. import { Component, OnInit } from '@angular/core';
2. import { Hero } from '../hero';
3. import { HEROES } from '../mock-heroes';
5. @Component({
6. selector: 'app-heroes',
7. templateUrl: './heroes.component.html',
8. styleUrls: ['./heroes.component.css']
9. })
10. export class HeroesComponent implements OnInit {
12. heroes = HEROES;
14. selectedHero: Hero;

17. constructor() { }
19. ngOnInit() {
20. }
22. onSelect(hero: Hero): void {
23. this.selectedHero = hero;
24. }
25. }

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.

# 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 HeroDetailsComponent.

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

## Make the HeroDetailComponent

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

content\_copyng 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

content\_copy<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)

content\_copyimport { 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.

content\_copy<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)

content\_copyimport { Component, OnInit, Input } from '@angular/core';

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

content\_copy@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)

content\_copy<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

content\_copy<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 / download example.

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

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

src/app/heroes/heroes.component.html

content\_copy

1. import { Component, OnInit, Input } from '@angular/core';
2. import { Hero } from '../hero';
4. @Component({
5. selector: 'app-hero-detail',
6. templateUrl: './hero-detail.component.html',
7. styleUrls: ['./hero-detail.component.css']
8. })
9. export class HeroDetailComponent implements OnInit {
10. @Input() hero: Hero;
12. constructor() { }
14. ngOnInit() {
15. }
17. }

## 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.