Create the component

Using the Angular CLI, generate a new component named employeees.

ng generate component employeees

The CLI creates a new folder, src/app/employeees/ and generates the three files of the EmployeeesComponent.

The EmployeeesComponent class file is as follows:

app/employeees/employeees.component.ts (initial version)

import { Component, OnInit } from '@angular/core';

@Component({

selector: 'app-employeees',

templateUrl: './employeees.component.html',

styleUrls: ['./employeees.component.css']

})

export class EmployeeesComponent 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-employeees', 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 employee property

Add a employee property to the EmployeeesComponent for a employee named "Windstorm."

employeees.component.ts (employee property)

employee = 'Windstorm';

### Show the employee

Open the employeees.component.html template file. Delete the default text generated by the Angular CLI and replace it with a data binding to the new employee property.

employeees.component.html

{{employee}}

## Show the EmployeeesComponent view

To display the EmployeeesComponent, you must add it to the template of the shell AppComponent.

Remember that app-employeees is the element selector for the EmployeeesComponent. So add an <app-employeees> element to the AppComponenttemplate file, just below the title.

src/app/app.component.html

<h1>{{title}}</h1>

<app-employeees></app-employeees>

Assuming that the CLI ng serve command is still running, the browser should refresh and display both the application title and the employee name.

## Create a Employee class

A real employee is more than a name.

Create a Employee class in its own file in the src/app folder. Give it id and name properties.

src/app/employee.ts

export class Employee {

id: number;

name: string;

}

Return to the EmployeeesComponent class and import the Employee class.

Refactor the component's employee property to be of type Employee. Initialize it with an id of 1 and the name Windstorm.

The revised EmployeeesComponent class file should look like this:

src/app/employeees/employeees.component.ts

import { Component, OnInit } from '@angular/core';

import { Employee } from '../employee';

@Component({

selector: 'app-employeees',

templateUrl: './employeees.component.html',

styleUrls: ['./employeees.component.css']

})

export class EmployeeesComponent implements OnInit {

employee: Employee = {

id: 1,

name: 'Windstorm'

};

constructor() { }

ngOnInit() {

}

}

The page no longer displays properly because you changed the employee from a string to an object.

## Show the employee object

Update the binding in the template to announce the employee's name and show both id and name in a details layout like this:

employeees.component.html (EmployeeesComponent's template)

<h2>{{ employee.name }} Details</h2>

<div><span>id: </span>{{employee.id}}</div>

<div><span>name: </span>{{employee.name}}</div>

The browser refreshes and display's the employee's information.

## Format with the UppercasePipe

Modify the employee.name binding like this.

<h2>{{ employee.name | uppercase }} Details</h2>

The browser refreshes and now the employee'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 employee

Users should be able to edit the employee name in an <input> textbox.

The textbox should both display the employee's name property and update that property as the user types. That means data flow 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 employee.name property.

### Two-way binding

Refactor the details area in the EmployeeesComponent template so it looks like this:

src/app/employeees/employeees.component.html (EmployeeesComponent's template)

<div>

<label>name:

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

</label>

</div>

**[(ngModel)]** is Angular's two-way data binding syntax.

Here it binds the employee.name property to the HTML textbox so that data can flow in both directions: from the employee.name property to the textbox, and from the textbox back to the employee.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 @NgModuledecorator 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 employee's name and see the changes reflected immediately in the <h2> above the textbox.

### Declare EmployeeesComponent

Every component must be declared in exactly one NgModule.

You didn't declare the EmployeeesComponent. So why did the application work?

It worked because the Angular CLI declared EmployeeesComponent in the AppModule when it generated that component.

Open src/app/app.module.ts and find EmployeeesComponent imported near the top.

import { EmployeeesComponent } from './employeees/employeees.component';

The EmployeeesComponent is declared in the @NgModule.declarations array.

declarations: [

AppComponent,

EmployeeesComponent

],

Note that AppModule declares both application components, AppComponent and EmployeeesComponent.

## Final code review

Your app should look like this live example / download example. Here are the code files discussed on this page.

src/app/employeees/employeees.component.ts

src/app/employeees/employeees.component.html

src/app/app.module.ts

src/app/app.component.ts

src/app/app.component.html

src/app/employee.ts

1. import { Component, OnInit } from '@angular/core';
2. import { Employee } from '../employee';
4. @Component({
5. selector: 'app-employeees',
6. templateUrl: './employeees.component.html',
7. styleUrls: ['./employeees.component.css']
8. })
9. export class EmployeeesComponent implements OnInit {
10. employee: Employee = {
11. id: 1,
12. name: 'Windstorm'
13. };
15. constructor() { }
17. ngOnInit() {
18. }
20. }

## Summary

* You used the CLI to create a second EmployeeesComponent.
* You displayed the EmployeeesComponent 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.

**Display a List/ ngFor**

## Create mock employeees

You'll need some employeees to display.

Eventually you'll get them from a remote data server. For now, you'll create some mock employeees and pretend they came from the server.

Create a file called mock-employeees.ts in the src/app/ folder. Define a EMPLOYEEES constant as an array of ten employeees and export it. The file should look like this.

src/app/mock-employeees.ts

import { Employee } from './employee';

export const EMPLOYEEES: Employee[] = [

{ 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 employeees

You're about to display the list of employeees at the top of the EmployeeesComponent.

Open the EmployeeesComponent class file and import the mock EMPLOYEEES.

src/app/employeees/employeees.component.ts (import EMPLOYEEES)

import { EMPLOYEEES } from '../mock-employeees';

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

employeees = EMPLOYEEES;

### List employeees with \*ngFor

Open the EmployeeesComponent 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 employee.
* Sprinkle some CSS classes for styling (you'll add the CSS styles shortly).

Make it look like this:

employeees.component.html (employeees template)

<h2>My Employeees</h2>

<ul class="employeees">

<li>

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

</li>

</ul>

Now change the <li> to this:

<li \*ngFor="let employee of employeees">

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
* employeees is the list from the EmployeeesComponent class.
* employee holds the current employee 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 employeees appears.

### Style the employeees

The employeees list should be attractive and should respond visually when users hover over and select a employee 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 employeees.

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 EmployeeesComponent, it created an empty employeees.component.css stylesheet for the EmployeeesComponent and pointed to it in @Component.styleUrls like this.

src/app/employeees/employeees.component.ts (@Component)

@Component({

selector: 'app-employeees',

templateUrl: './employeees.component.html',

styleUrls: ['./employeees.component.css']

})

Open the employeees.component.css file and paste in the private CSS styles for the EmployeeesComponent. 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 employeees.component.css styles apply only to the EmployeeesComponent and don't affect the outer HTML or the HTML in any other component.

## Master/Detail

When the user clicks a employee in the **master** list, the component should display the selected employee's **details** at the bottom of the page.

In this section, you'll listen for the employee item click event and update the employee detail.

### Add a click event binding

Add a click event binding to the <li> like this:

employeees.component.html (template excerpt)

<li \*ngFor="let employee of employeees" (click)="onSelect(employee)">

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(employee) expression.

onSelect() is a EmployeeesComponent method that you're about to write. Angular calls it with the employee object displayed in the clicked <li>, the same employee defined previously in the \*ngFor expression.

### Add the click event handler

Rename the component's employee property to selectedEmployee but don't assign it. There is no selected employee when the application starts.

Add the following onSelect() method, which assigns the clicked employee from the template to the component's selectedEmployee.

src/app/employeees/employeees.component.ts (onSelect)

selectedEmployee: Employee;

onSelect(employee: Employee): void {

this.selectedEmployee = employee;

}

### Update the details template

The template still refers to the component's old employee property which no longer exists. Rename employee to selectedEmployee.

employeees.component.html (selected employee details)

<h2>{{ selectedEmployee.name | uppercase }} Details</h2>

<div><span>id: </span>{{selectedEmployee.id}}</div>

<div>

<label>name:

<input [(ngModel)]="selectedEmployee.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:

EmployeeesComponent.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 employeees appear in a list and details about the clicked employee appear at the bottom of the page.

#### What happened?

When the app starts, the selectedEmployee is undefined by design.

Binding expressions in the template that refer to properties of selectedEmployee — expressions like {{selectedEmployee.name}} — must failbecause there is no selected employee.

#### The fix

The component should only display the selected employee details if the selectedEmployee exists.

Wrap the employee detail HTML in a <div>. Add Angular's \*ngIf directive to the <div> and set it to selectedEmployee.

Don't forget the asterisk (\*) in front of ngIf. It's a critical part of the syntax.

src/app/employeees/employeees.component.html (\*ngIf)

<div \*ngIf="selectedEmployee">

<h2>{{ selectedEmployee.name | uppercase }} Details</h2>

<div><span>id: </span>{{selectedEmployee.id}}</div>

<div>

<label>name:

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

</label>

</div>

</div>

After the browser refreshes, the list of names reappears. The details area is blank. Click a employee and its details appear.

#### Why it works

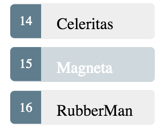
When selectedEmployee is undefined, the ngIf removes the employee detail from the DOM. There are no selectedEmployee bindings to worry about.

When the user picks a employee, selectedEmployee has a value and ngIf puts the employee detail into the DOM.

### Style the selected employee

It's difficult to identify the selected employee in the list when all <li> elements look alike.

If the user clicks "Magneta", that employee should render with a distinctive but subtle background color like this:



That selected employee 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 EmployeeesComponent template:

employeees.component.html (toggle the 'selected' CSS class)

[class.selected]="employee === selectedEmployee"

When the current row employee is the same as the selectedEmployee, Angular adds the selected CSS class. When the two employeees are different, Angular removes the class.

The finished <li> looks like this:

employeees.component.html (list item employee)

<li \*ngFor="let employee of employeees"

[class.selected]="employee === selectedEmployee"

(click)="onSelect(employee)">

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

</li>

## Final code review

Your app should look like this live example / download example.

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

src/app/employeees/employeees.component.ts

src/app/employeees/employeees.component.html

src/app/employeees/employeees.component.css

1. import { Component, OnInit } from '@angular/core';
2. import { Employee } from '../employee';
3. import { EMPLOYEEES } from '../mock-employeees';
5. @Component({
6. selector: 'app-employeees',
7. templateUrl: './employeees.component.html',
8. styleUrls: ['./employeees.component.css']
9. })
10. export class EmployeeesComponent implements OnInit {
12. employeees = EMPLOYEEES;
14. selectedEmployee: Employee;

17. constructor() { }
19. ngOnInit() {
20. }
22. onSelect(employee: Employee): void {
23. this.selectedEmployee = employee;
24. }
25. }

Master/Detail Components

## Make the EmployeeDetailComponent

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

ng generate component employee-detail

The command scaffolds the EmployeeDetailComponent files and declares the component in AppModule.

### Write the template

Cut the HTML for the employee detail from the bottom of the EmployeeesComponent template and paste it over the generated boilerplate in the EmployeeDetailComponent template.

The pasted HTML refers to a selectedEmployee. The new EmployeeDetailComponent can present any employee, not just a selected employee. So replace "selectedEmployee" with "employee" everywhere in the template.

When you're done, the EmployeeDetailComponent template should look like this:

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

<div \*ngIf="employee">

<h2>{{ employee.name | uppercase }} Details</h2>

<div><span>id: </span>{{employee.id}}</div>

<div>

<label>name:

<input [(ngModel)]="employee.name" placeholder="name"/>

</label>

</div>

</div>

### Add the @Input() employee property

The EmployeeDetailComponent template binds to the component's employee property which is of type Employee.

Open the EmployeeDetailComponent class file and import the Employee symbol.

src/app/employee-detail/employee-detail.component.ts (import Employee)

import { Employee } from '../employee';

The employee property must be an Input property, annotated with the @Input() decorator, because the external EmployeeesComponent will bind to it like this.

<app-employee-detail [employee]="selectedEmployee"></app-employee-detail>

Amend the @angular/core import statement to include the Input symbol.

src/app/employee-detail/employee-detail.component.ts (import Input)

import { Component, OnInit, Input } from '@angular/core';

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

@Input() employee: Employee;

That's the only change you should make to the EmployeeDetailComponent class. There are no more properties. There's no presentation logic. This component simply receives a employee object through its employee property and displays it.

## Show the EmployeeDetailComponent

The EmployeeesComponent is still a master/detail view.

It used to display the employee details on its own, before you cut that portion of the template. Now it will delegate to the EmployeeDetailComponent.

The two components will have a parent/child relationship. The parent EmployeeesComponent will control the child EmployeeDetailComponent by sending it a new employee to display whenever the user selects a employee from the list.

You won't change the EmployeeesComponent class but you will change its template.

### Update the EmployeeesComponent template

The EmployeeDetailComponent selector is 'app-employee-detail'. Add an <app-employee-detail> element near the bottom of the EmployeeesComponent template, where the employee detail view used to be.

Bind the EmployeeesComponent.selectedEmployee to the element's employee property like this.

employeees.component.html (EmployeeDetail binding)

<app-employee-detail [employee]="selectedEmployee"></app-employee-detail>

[employee]="selectedEmployee" is an Angular property binding.

It's a one way data binding from the selectedEmployee property of the EmployeeComponent to the employee property of the target element, which maps to the employee property of the EmployeeDetailComponent.

Now when the user clicks a employee in the list, the selectedEmployee changes. When the selectedEmployee changes, the property binding updates employee and the EmployeeDetailComponent displays the new employee.

The revised EmployeeesComponent template should look like this:

employeees.component.html

<h2>My Employeees</h2>

<ul class="employeees">

<li \*ngFor="let employee of employeees"

[class.selected]="employee === selectedEmployee"

(click)="onSelect(employee)">

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

</li>

</ul>

<app-employee-detail [employee]="selectedEmployee"></app-employee-detail>

The browser refreshes and the app starts working again as it did before.

## What changed?

As before, whenever a user clicks on a employee name, the employee detail appears below the employee list. Now the EmployeeDetailComponent is presenting those details instead of the EmployeeesComponent.

Refactoring the original EmployeeesComponent into two components yields benefits, both now and in the future:

1. You simplified the EmployeeesComponent by reducing its responsibilities.
2. You can evolve the EmployeeDetailComponent into a rich employee editor without touching the parent EmployeeesComponent.
3. You can evolve the EmployeeesComponent without touching the employee detail view.
4. You can re-use the EmployeeDetailComponent 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/employee-detail/employee-detail.component.ts

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

src/app/employeees/employeees.component.html

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