Angular
Essentials

Published By:

INFRAGISTICS

By: Rabi Kiran, Mahesh Sabnis, Suprotim Agarwal

A Note from Infragistics

Thanks for downloading *Angular Essentials*. Our goal is to provide the development community with the resources and tools that will help you build great modern web applications. This book will be a great help to developers who wish to learn how to build client-side JavaScript applications with Angular.

This book begins with the basics—giving you a foundation to build upon. You'll then move on to advanced topics, learning the features of Angular you will need to build enterprise-class web applications.

While *Angular Essentials* focuses on components and services built into the Angular framework, enterprise-class applications often require more specialized and feature-rich components. And the best components available for these applications are in *Infragistics Ignite UI Components for JavaScript/HTML5 and ASP.NET MVC and Ignite UI for Angular.* Ignite UI is a complete library of HTML and JavaScript controls and tools that enables developers to quickly and easily build high-performing modern web applications on every device—desktop, tablet and phone—with the most popular modern web frameworks.

These features include:

A complete set of Grid controls, including OLAP and Tree grids

A wide range of data visualization options, including a long list of charts, geographical maps and interactive gauges.

The fastest rendering charts and grids available, even with large data sets

Data-bound controls that can bind to local, remote or even real-time streaming data sources

Over 60 components designed to help you build and deliver your application quickly

Ignite UI for Angular has a complete set of component wrappers that enable you to use our components in your Angular application as if they were native Angular components. You don't get some watered-down framework that's missing key features your users need; Ignite UI provides all the functionality from our product in Angular, today.

In addition to this book, we've prepared nine lessons (found at the back of this book) to help you not only get a better understanding of how to work with Angular, but also how Ignite UI can be a great addition to your developer toolbox. Follow them and learn how Ignite UI (*which you can try now for free*) offers the full-featured controls you need without compromise.

Enjoy the book,

James Bender
Infragistics Product Manager, Ignite UI
JBender@infragistics.com

Angular
Essentials

INFRAGISTICS®

Download the Code: *infragistics.com/products/ignite-ui/angular-essentials-get-the-code*

Chapter 1

ECMAScript Overview

ECMASCRIPT IS A LANGUAGE standardized by ECMA International and overseen by the TC39 committee. JavaScript is a language based on the implementations of ECMAScript. In June 2009, the 5th edition of ECMAScript called ECMAScript 5 or ES5, was standardized. This standard was implemented by all modern browsers.

In June 2015, the ECMAScript committee released a major update of the specifications and added some much-needed features to make the job of JavaScript developers much easier. This 6th edition of ECMAScript was previously called ECMAScript 6 or ES6. It has now been renamed ECMAScript 2015.

In June 2016, ECMAScript 2016—the 7th edition of ECMAScript was released.

Describing all of the improvements made by ECMAScript 2015 would require an entire book. Instead, this chapter will serve as an overview of some basic features that we'll use throughout this book. If you are interested in a deep dive in ES6, you may refer to: *https://hacks.mozilla.org/category/es6-in-depth/*.

This chapter assumes that you have basic working knowledge of JavaScript and have worked on at least one object-oriented language such as C# or Java.

> **NOTE:** The ECMAScript 2015 and 2015 standard has been implemented in most modern browsers, but not in all of them. To use ECMAScript 2015 code in browsers that support ES5, you can use *transpilers* which takes ES6 code and converts it to its ES5 equivalent that most browsers can understand. To check for a list of features supported by a particular browser or a transpiler, visit the ES6 compatibility table: http://kangax.github.io/compat-table/es6/.

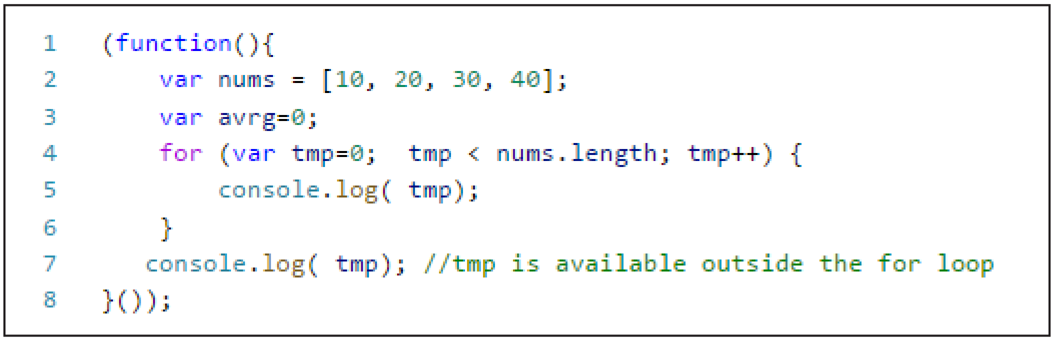
New ECMAScript 2015 Features

BLOCK SCOPING WITH LET AND CONST KEYWORDS

Up to ECMAScript 5, JavaScript variables had only two types of scopes: global and function. JavaScript lacked the *block* scope—a scope where variables are declared to be made as local as possible.

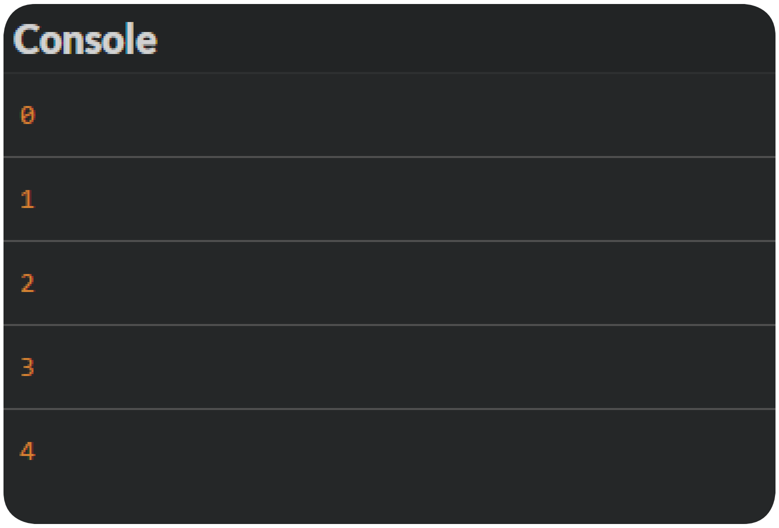
Consider this piece of code:

Listing 1.1: *for loop variable*



```
(function(){
    var nums = [10, 20, 30, 40];
    var avrg=0;
    for (var tmp=0;  tmp < nums.length; tmp++) {
        console.log( tmp);
    }
   console.log( tmp); //tmp is available outside the for loop
}());
```

After running this code, you'll see the following output on the console:



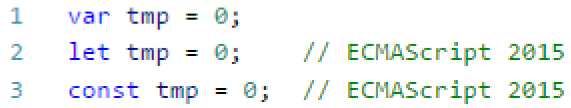
```
0
1
2
3
4
```

For a programmer who has worked on any other programming language like C, C# or Java, declaring *tmp* inside a *for* loop means the intent is to use *tmp* only within the context of that loop. The *tmp* variable should not be accessible outside the loop. The expected output would be an error saying *tmp* is undeclared.

However, in ECMAScript 5, the *tmp* variable actually scopes itself to the enclosing function, which can be confusing.

To eliminate this confusion, ECMAScript 2015 introduces the *block* scope. Block scoping works on the bracket level, rather than the function level. This allows the scope of the variable to be limited to only the block in which it is declared. ECMA members could not change the behavior of the *var* keyword, as that would break backward compatibility. Hence, two new keywords were introduced: **let** and **const**.

To understand *let* and *const*, and the difference from *var*, let us first consider the following statements, declared *outside* a block:



```
var tmp = 0;
let tmp = 0;     // ECMAScript 2015
const tmp = 0;   // ECMAScript 2015
```

The first statement creates a global variable, which is a property of the global object.

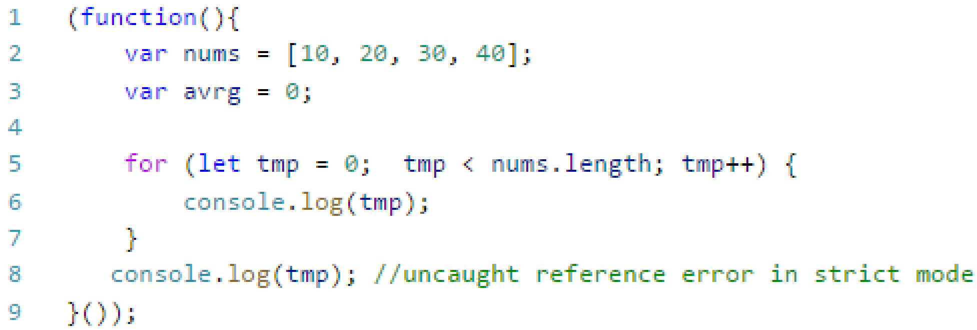
The second statement creates a global variable, however, this variable does not belong to the global object.

The third statement is a global constant, and is not a property of the global object.

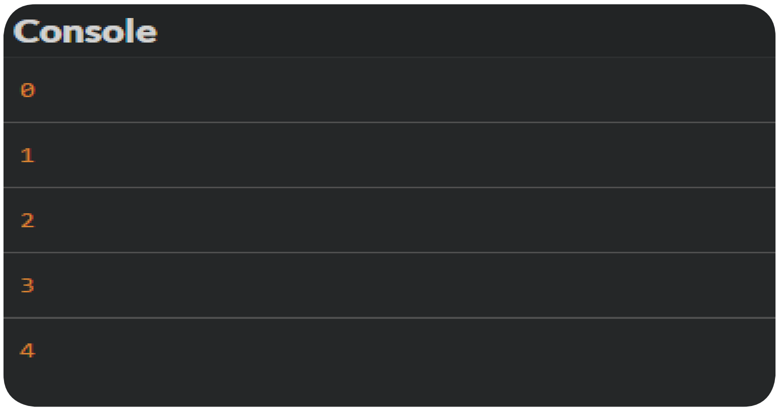
LET KEYWORD

Simply put, *let* is the new *var*. However, the *let* keyword allows JavaScript developers to define block-level variables, rather than global variables or function block variables.

Consider the same example as seen in Listing 1.1, albeit with the *let* keyword instead of *var*.



```
(function(){
    var nums = [10, 20, 30, 40];
    var avrg = 0;

    for (let tmp = 0;  tmp < nums.length; tmp++) {
        console.log(tmp);
    }
   console.log(tmp); //uncaught reference error in strict mode
}());
```



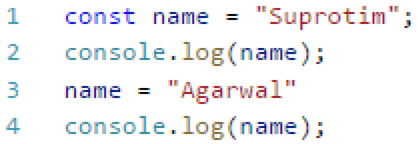
```
Console
0
1
2
3
4
```

With the *let* keyword, the output is as expected. In a strict mode, the last statement would throw an error as *tmp* is not declared at the function level.

CONST KEYWORD

In ECMAScript 5, there is no direct way to define a constant. ECMAScript 2015 solves this problem with the *const* keyword.

Like *let*, *const* also has a block-level scope, but it requires you to provide an initializer before you can use it. Moreover, once you declare a variable using *const*, you can't change its value.



```
const name = "Suprotim";
console.log(name);
name = "Agarwal"
console.log(name);
```

While the first output is Suprotim, the second output is `"TypeError: Assignment to constant variable",` since you cannot change the value of the constant once it's created.

NOTE: const has nothing to do with immutability of values. When it is said that its value cannot be changed, what is meant is that there is a flag set on the identifier binding, and it is this binding that is immutable.

Template literals

Template literals are a new way to define strings. Since template literals are declared using backticks (`) as shown here:

```
let str = `This is a template literal`;
```

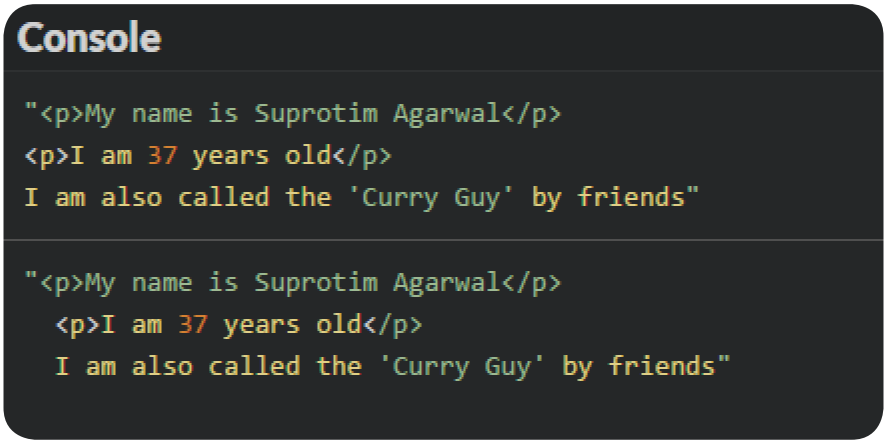
You can now do the following without having to escape double quotes:

```
let str = `This is a "template" literal`;
```

Template literals look better and are easier to read, especially when used in multi-line strings, and while concatenating strings. Here's an example:

```
let employee = {
  name: 'Suprotim Agarwal', age: 37, nickname: 'Curry Guy'
};
let concatStr =
    "<p>My name is " + employee.name + "</p>\n" +
    "<p>I am " + employee.age + " years old</p>\n" +
    "I am also called the \"" + employee.nickname +"\" by friends";

let templateStr =
 `<p>My name is ${employee.name}</p>
  <p>I am ${employee.age} years old</p>
  I am also called the "${employee.nickname}" by friends`;

console.log(concatStr);
console.log(templateStr);
```

Output:



```
"<p>My name is Suprotim Agarwal</p>
<p>I am 37 years old</p>
I am also called the 'Curry Guy' by friends"

"<p>My name is Suprotim Agarwal</p>
  <p>I am 37 years old</p>
  I am also called the 'Curry Guy' by friends"
```

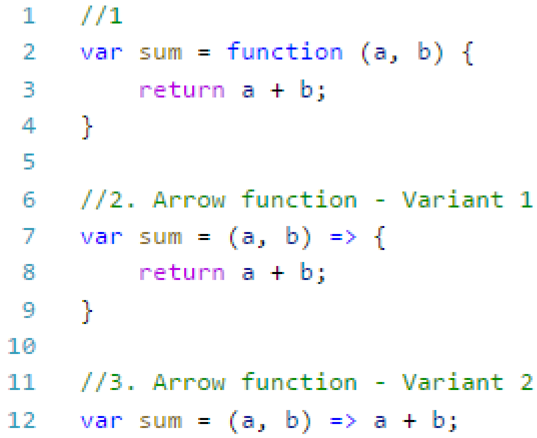
As you can see, in concatStr we had to resort to messy concatenations, and escape sequences. However, using the new template literals feature of ECMAScript 2015, we were able to use backticks (`) and placeholders ${} to achieve the same output.

These placeholders can contain any JavaScript expression including a variable, object property access, and even function calls. These tag expressions get called with the processed template literal, which you can then manipulate before outputting them.

Arrow Functions

Arrow functions allow you to write an inline function that can be passed in to another function. Arrow functions were introduced in ECMAScript 2015. They are syntactically very similar to C# lambda functions.

Here's an example of using an Arrow function:



```
//1
var sum = function (a, b) {
    return a + b;
}

//2. Arrow function - Variant 1
var sum = (a, b) => {
    return a + b;
}

//3. Arrow function - Variant 2
var sum = (a, b) => a + b;
```

The first example uses a JavaScript function.

The second example uses Arrow function in a statement body.

The third example uses Arrow function in an expression body.

One huge advantage of the Arrow function is that it retains the lexically-scoped 'this' variable. In other words, arrow functions share the same lexical *this* as their surrounding code. Sometimes when the lexical scope of 'this' changes, you lose track of what it was referring to. However, with Arrow functions, the value of 'this' is preserved.

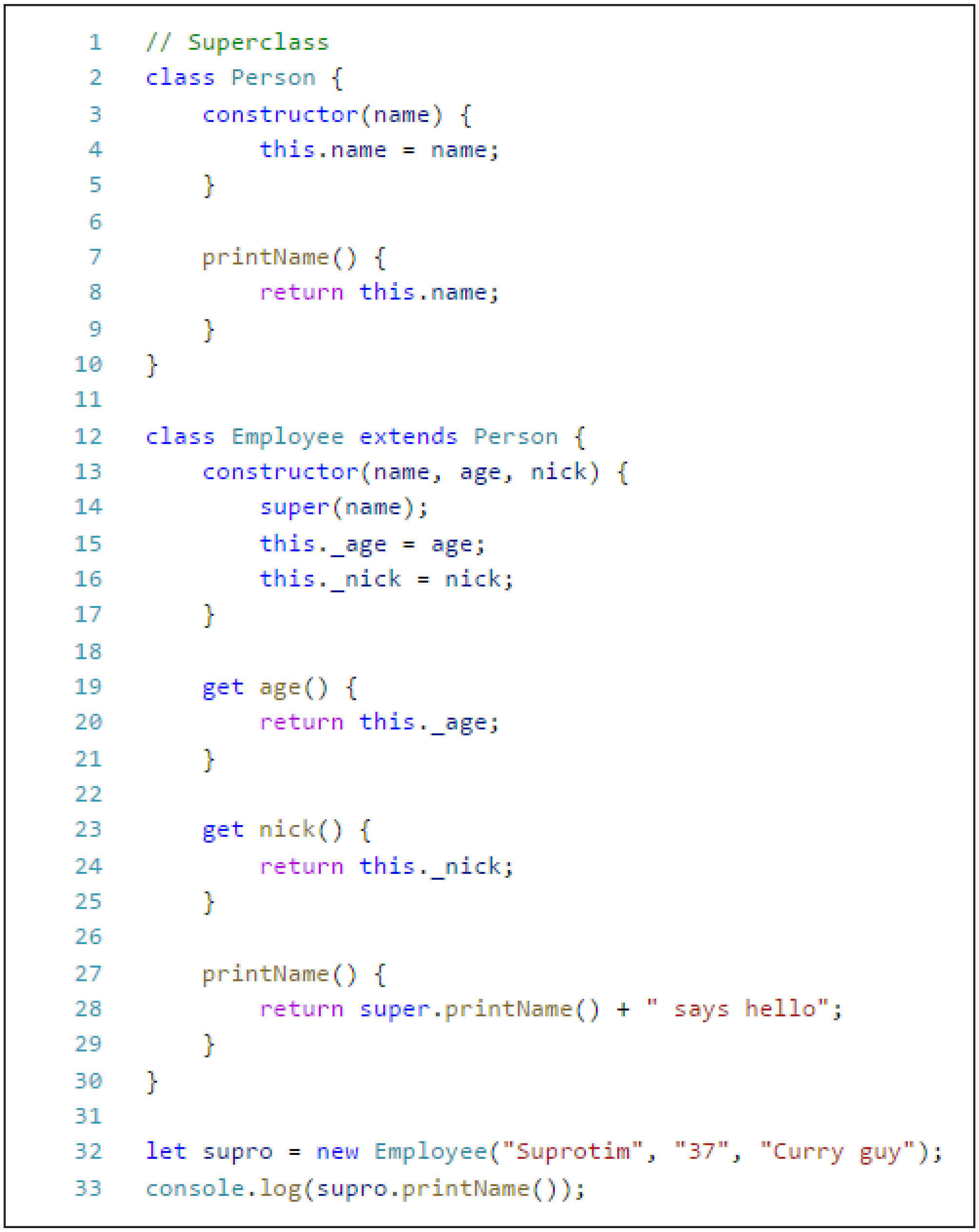
Classes and Inheritance

Despite the fact that we cannot compose classes in JavaScript as we do in Object Oriented languages, we can compose comparative constructs using the *prototype* property of Objects. There is a learning curve around the *prototype* property before one can start utilizing it serenely to compose OOPs like code in JavaScript. To ease this learning curve, ECMAScript 2015 uses *classes*. Classes in ECMAScript 2015 are similar to *prototype-based inheritance*. You can consider them as syntactic sugar over the latter.

Using ECMAScript 2015 classes, we can write classes, create objects, inherit and override features of parent class inside child classes.

Here's an example of using Classes and Inheritance in ECMAScript 2015:

Listing 1.2 Classes and Inheritance in ES6



```
// Superclass
class Person {
    constructor(name) {
        this.name = name;
    }

    printName() {
        return this.name;
    }
}

class Employee extends Person {
    constructor(name, age, nick) {
        super(name);
        this._age = age;
        this._nick = nick;
    }

    get age() {
        return this._age;
    }

    get nick() {
        return this._nick;
    }

    printName() {
        return super.printName() + " says hello";
    }
}

let supro = new Employee("Suprotim", "37", "Curry guy");
console.log(supro.printName());
```

Classes can be **inherited** from other classes using the *extends* keyword. In this code, we have a parent class *Person*, and a child class *Employee* that is an extension of the *Person* class. Here *Person* is also referred to as the superclass, whereas as *Employee* is referred to as subclass.

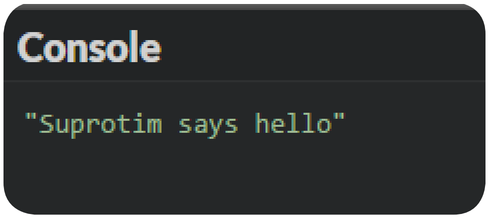
The **constructor** is a function that gets called when you create a new instance of the class. In our example, we have used `let supro = new Employee("Suprotim", "37", "Curry guy");` to invoke the constructor.

> **NOTE:** In JavaScript, there can only be one constructor for a class.

super is used in subclass *Employee* to refer to the superclass *Person*. Note that in the constructor of *Employee*, we are calling *super* as if it were a function. Internally, this calls the superclass *Person's* constructor function, and initializes the new object that was created by the keyword *new*.

super can also be used to access all members of a parent class. In Listing 1.2, we are using super.printName() in Employees printName() method, to call the superclass Person's printName() method.

Running the code in Listing 1.2 will produce the following output:



```
"Suprotim says hello"
```

> **NOTE:** Object.create can be used to do prototypical inheritance without using constructor functions.

Modules

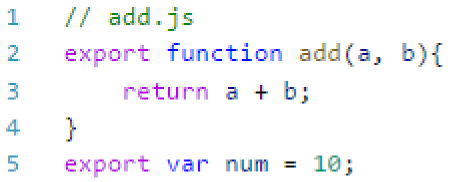
In ECMAScript 2015, a module is a file containing JavaScript code. ECMAScript supports exporting and importing modules across different files.

There's no *module* keyword per-se in ECMAScript 2015. A module is just like a script, except that it is in strict-mode and contains import and export keywords.

export: All members of a module are local to it. In order to make it public, use the export keyword.

import: When you run a module with an import declaration, the imported modules are loaded first, then the module body is executed.

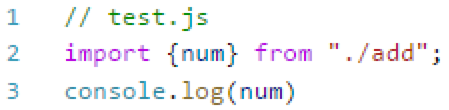
Let's take the example of a simple add.js file:



```
// add.js
export function add(a, b){
    return a + b;
}
export var num = 10;
```

Here we are exporting functions along with variables one by one.

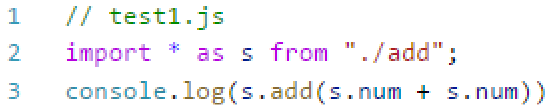
To use it inside another file, say test.js, do the following:



```
// test.js
import {num} from "./add";
console.log(num)
```

Here we are importing only the *num* variable from add.js.

We can also import everything inside add.js using import *:



```
// test1.js
import * as s from "./add";
console.log(s.add(s.num + s.num))
```

NOTE: You can export all members of a Module using export *.

Conclusion

This chapter presented a quick overview of some ECMAScript 2015 features that you will be using throughout this book. For a detailed tutorial, reference: https://hacks.mozilla.org/category/es6-in-depth/.

Chapter 2

TypeScript Overview

IN THE FIRST CHAPTER, we discussed how ECMAScript or ES lays down the standard for scripting language and how JavaScript is based on this standard.

ES5 (JavaScript 1.5) was released in June 2011 and has the most consistent support across all browsers. In June 2015, ECMAScript 2015 (previously known as ES6) was approved, followed by ECMAScript 2016 (previously known as ES7) in June 2016. (Yes! June is ECMAScript month.) The improvements in the new ECMAScript specification boost productivity by providing features aimed at modern application development. However, most modern browsers do not support ECMAScript 2015/2016, so ECMAScript 2015 code must be "transpiled" to ES5 via npm, or via a code editor like Visual Studio Code, before it can be used. The transpilation process involves converting ES 2015 syntax to comparable ES5 syntax (standard JavaScript) so that most modern browsers can execute it. This is a win-win situation for both browsers and developers, as the browsers get code they can understand, while developers get to use all the latest productivity features of ECMAScript.

Angular 2 applications can be written with both ES5 and/or ECMAScript 2015, along with TypeScript.

Introducing TypeScript

TypeScript is an open-source programming language from Microsoft, written on top of JavaScript to support types. First announced in October 2012, it comes with powerful type-checking abilities and object-oriented features. It also leverages ES6 as part of its core foundation. Angular 2 uses TypeScript as its primary language for application development.

All of the proposed features of ES 2015 and ES 2016 are implemented by TypeScript, or are in the process of being implemented. These features are converted into their ES5 equivalent when the TypeScript files are *transpiled*, making it possible to use the latest features of JavaScript even when the browsers have not implemented them natively.

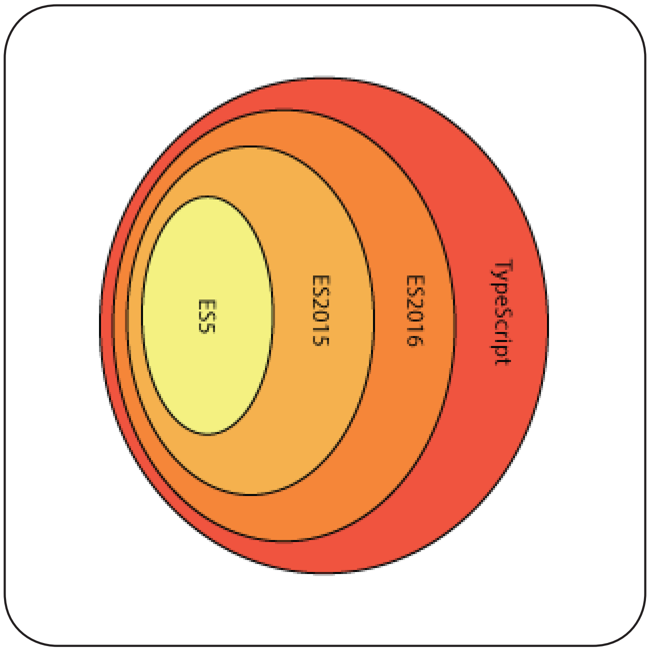
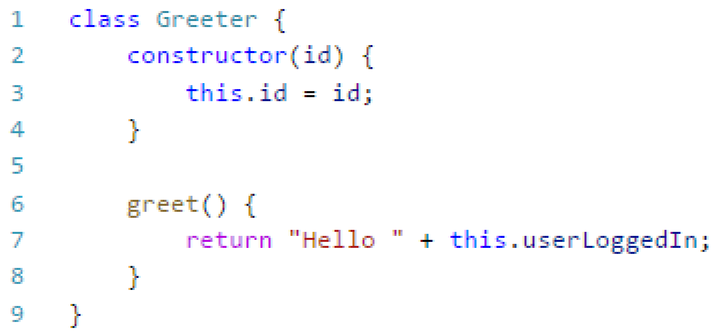


Figure 1: *Relationship between ECMAScript and TypeScript*

Here are some important points to keep in mind:

- TypeScript is a typed superset of JavaScript and compiles to plain JavaScript.
- TypeScript can be used to write everything that can be written in JavaScript.
- TypeScript files have the extension .ts
- TypeScript is not a completely new language and is not intended to replace JavaScript in any way. It adheres to all principles of JavaScript and just adds types on top of it. Strict type checking system makes the code more predictable. The type system of the language helps in solving many issues during development, which are challenging to catch until runtime.
- Since TypeScript supports types, it looks very familiar to any other typed OOPs languages like C# and Java.
- TypeScript is not executed on the browser. Instead, the transpiled code gets executed in the browser.

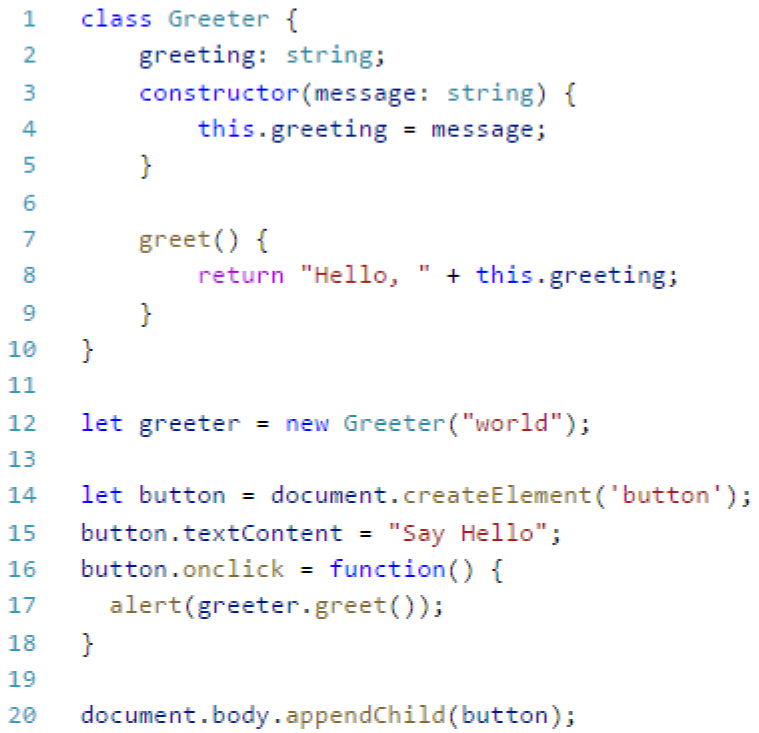
Your ES6 code can be quickly converted to TypeScript code by just renaming the file extension from .js to .ts. For example, see this piece of code of a file test.js, which is valid ECMAScript 2015, as well as TypeScript. Just renaming this file to test.ts makes it TypeScript:



```
class Greeter {
    constructor(id) {
        this.id = id;
    }

    greet() {
        return "Hello " + this.userLoggedIn;
    }
}
```

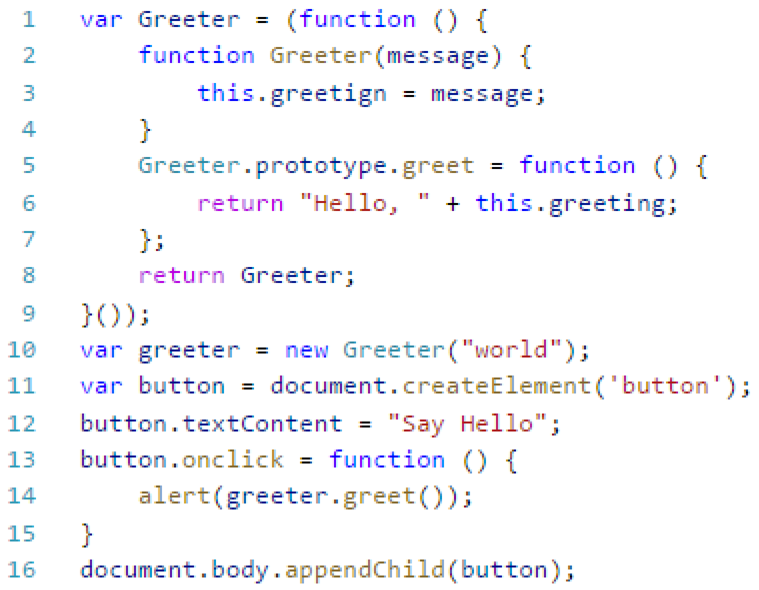
TypeScript can target multiple versions of JavaScript, including ES5, ECMAScript 2015 or 2016.

Forexample, consider this TypeScript class:



```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }

    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter = new Greeter("world");

let button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function() {
  alert(greeter.greet());
}

document.body.appendChild(button);
```

It easily gets transpiled to ES5:



```
var Greeter = (function () {
    function Greeter(message) {
        this.greetign = message;
    }
    Greeter.prototype.greet = function () {
        return "Hello, " + this.greeting;
    };
    return Greeter;
}());
var greeter = new Greeter("world");
var button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function () {
    alert(greeter.greet());
}
document.body.appendChild(button);
```

The same holds true for any existing JavaScript library. All you need is a type definition file declaring types for the APIs exposed by the library. The GitHub project *Definitely Typed* is a repository for high- quality TypeScript type definitions. These files are made available through a package manager to make it easier to install and use the type definition files.

> **Note 1:** You can also try transpilation online using TypeScript playground: *http://www.typescriptlang.org/play/*

> **Note 2:** We will use VS Code throughout this book to write Angular applications using TypeScript. Chapter 5—Angular 2 Development Environment gives you a step by step overview of using TypeScript using VS Code.

TYPESCRIPT CRASH COURSE

Now that we have taken a quick overview of TypeScript and why it exists, let's look at just a few of the more important features that will be used in the rest of the book. For a detailed tutorial on TypeScript, please refer to http://www.typescriptlang.org/docs/tutorial.html

TYPES

TypeScript defines a set of basic/primitive and general purpose types.

PRIMITIVE TYPES

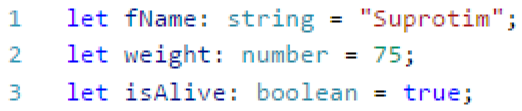
TypeScript inherits five primitive types from JavaScript—String, Number, Boolean, undefined and null.

String: can be assigned with single or double quotes. You can also use template strings.

Number: stores any number as integer or float. All numbers in TypeScript are floating point values. TypeScript supports hexadecimal, decimal literals, and also supports binary and octal literals.

Boolean: stores true or false values.

For example:



```
let fName: string = "Suprotim";
let weight: number = 75;
let isAlive: boolean = true;
```

Undefined refers to a variable that has no value or object assigned to it, whereas ***null*** refers to a variable that has been assigned an object, but that object is null.

Apart from these primitive types, we also have Array, Enum, Any, and Void.

ARRAYS

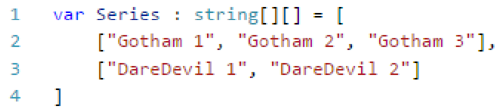
You can create Arrays in TypeScript in two ways:

```
var listNum:number[] = [20,20,40];
```

or using Generics and "<>"

```
var listNum:Array<number> = [20,30,40];
```

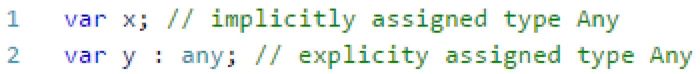
TypeScript also supports Arrays of Arrays, also called as Multidimensional arrays:



```
var Series : string[][] = [
    ["Gotham 1", "Gotham 2", "Gotham 3"],
    ["DareDevil 1", "DareDevil 2"]
]
```

ANY

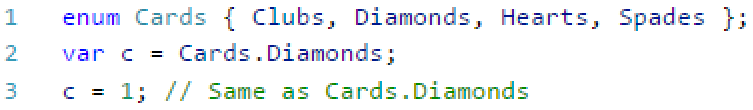
If you are declaring a variable without using a type and without assigning any value to it, the TLS assigns it the *any* type. Think of *any* as a supertype of all types, which is used whenever TypeScript is unable to infer a type. You can also assign the *any* type explicitly, which allows it to assume any value—from a simple string, number, Boolean, or even a complex object.



```
var x; // implicitly assigned type Any
var y : any; // explicity assigned type Any
```

ENUM

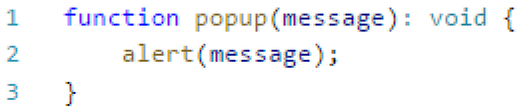
Enumerations give a friendly name to a set of values:



```
enum Cards { Clubs, Diamonds, Hearts, Spades };
var c = Cards.Diamonds;
c = 1; // Same as Cards.Diamonds
```

VOID

Void signifies that a method does not have a return value:

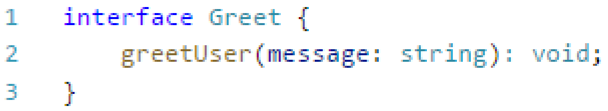


```
function popup(message): void {
    alert(message);
}
```

INTERFACES

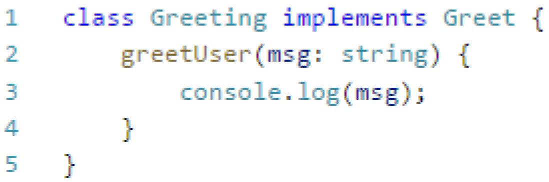
In TypeScript, interfaces are used to describe types. They are compile-time constructs that are used to define the contract for a class by just declaring the methods and fields.

Consider the following interface:



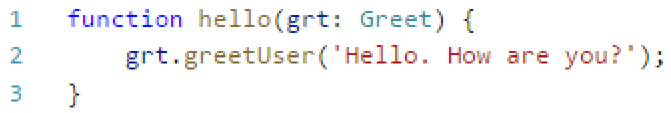
```
interface Greet {
    greetUser(message: string): void;
}
```

This interface is inherited by a class, and the class implementing the interface defines all members of the interface. To implement the interface, use *implements*:



```
class Greeting implements Greet {
    greetUser(msg: string) {
        console.log(msg);
    }
}
```

TypeScript interfaces can apply to functions, properties, and also to arrays. Here's an example of applying it to a function:

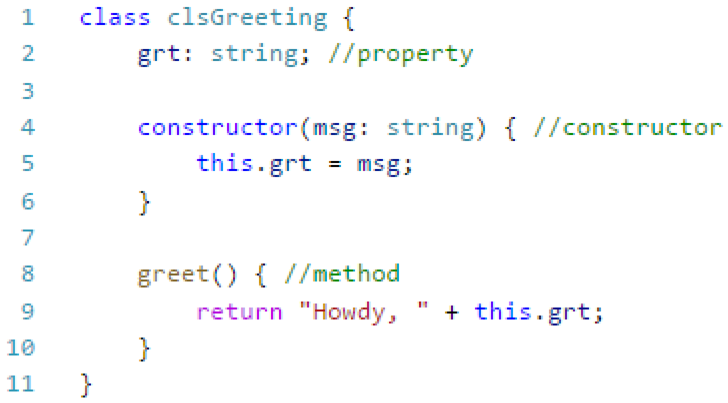


```
function hello(grt: Greet) {
    grt.greetUser('Hello. How are you?');
}
```

Classes

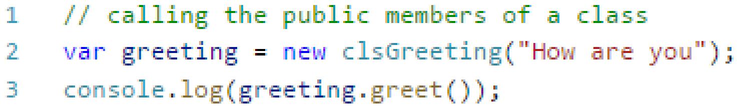
Classes are used to define blueprints of objects. Although classes are not present in ECMAScript 5, they are a part of ECMAScript 2016 and TypeScript. The class syntax system of TypeScript is very similar to C#, as well as the ECMAScript 2016.

Here's an example of a class with a property, constructor, and method:



```
class clsGreeting {
    grt: string; //property

    constructor(msg: string) { //constructor
        this.grt = msg;
    }

    greet() { //method
        return "Howdy, " + this.grt;
    }
}
```

The default access specifier in TypeScript is *public*. In our case, method *greet()* is public. We can instantiate the class using the `new` keyword, and call the public members of the class using the object (greeting):



```
// calling the public members of a class
var greeting = new clsGreeting("How are you");
console.log(greeting.greet());
```

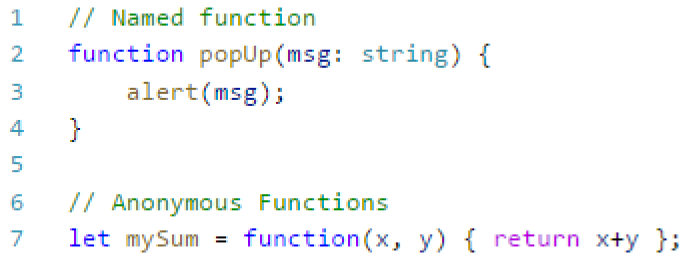
NOTE: If you do not want a field or method to be accessed directly using the object, make it explicitly private.

NOTE: TypeScript has no equivalent to C# Struts.

Functions

Functions describe how to do things and are the building blocks of any application. TypeScript functions can be created as named functions (class methods), as anonymous functions (no name), and as global functions (created outside a class). The functions can have typed arguments and a return type.

Consider the following named and anonymous functions:

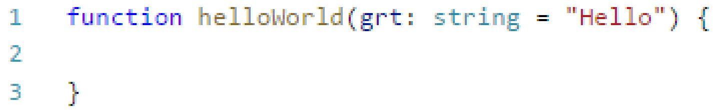


```
// Named function
function popUp(msg: string) {
    alert(msg);
}

// Anonymous Functions
let mySum = function(x, y) { return x+y };
```

> **NOTE:** TypeScript doesn't allow passing variables of different types into the function unless there are explicit declarations of the function with those types.

DEFAULT VALUE FOR PARAMETERS

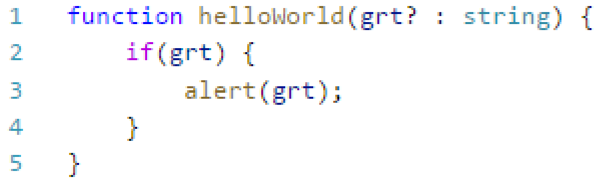
You can pass a default value to a function parameter:



```
function helloWorld(grt: string = "Hello") {

}
```

OPTIONAL PARAMETERS

Functions can have Optional parameters as well:



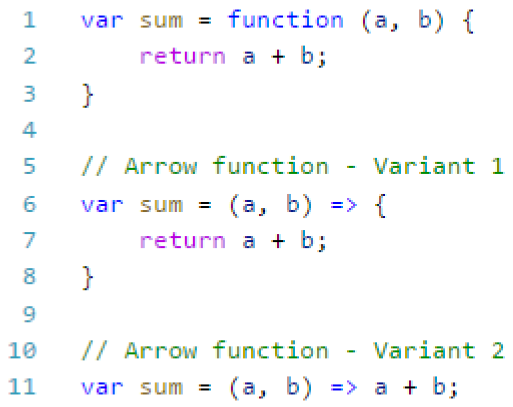
```
function helloWorld(grt? : string) {
    if(grt) {
        alert(grt);
    }
}
```

ARROW FUNCTIONS

Arrow function allows you to write an inline function that can be passed in to another function. Arrow functions are not present in ECMAScript 2015, but they are in

ECMAScript 2016 and TypeScript. They are syntactically very similar to C# lambda functions.

Here’s an example of using an Arrow function:



```
var sum = function (a, b) {
    return a + b;
}

// Arrow function - Variant 1
var sum = (a, b) => {
    return a + b;
}

// Arrow function - Variant 2
var sum = (a, b) => a + b;
```

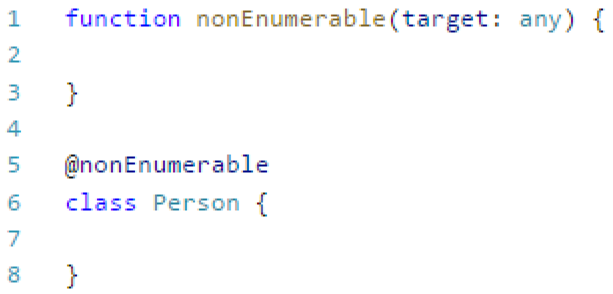
One huge advantage of the Arrow function is that retains the lexically-scoped ‘this’ variable. Sometimes when the lexical scope of ‘this’ changes, you lose track of what it was referring to. However, with Arrow functions, the value of ‘this’ is preserved.

DECORATORS

Decorators are used to extend the behavior of a class, property, method, or method parameter without modifying the implementation. In some languages, decorators are referred to as *annotations*. Decorators use the form @expression, where expression must be a function that will be called at runtime with information about the decorated entity.

Creating and using decorators is very easy.

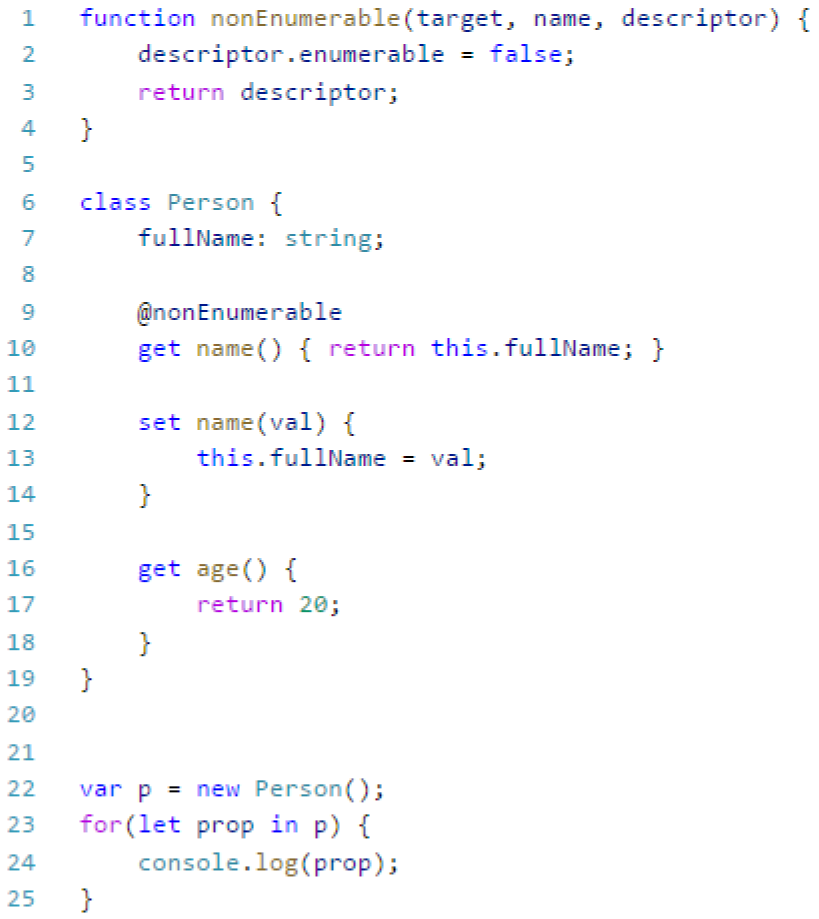
Here’s a simple decorator example:



```
function nonEnumerable(target: any) {

}

@nonEnumerable
class Person {

}
```

A custom decorator is a function that accepts some arguments containing details of the target on which it is applied. It can modify the way the target works using this information.

The following snippet defines and uses a decorator:



```
function nonEnumerable(target, name, descriptor) {
    descriptor.enumerable = false;
    return descriptor;
}

class Person {
    fullName: string;

    @nonEnumerable
    get name() { return this.fullName; }

    set name(val) {
        this.fullName = val;
    }

    get age() {
        return 20;
    }
}


var p = new Person();
for(let prop in p) {
    console.log(prop);
}
```

Here the decorator *nonEnumerable* sets the enumerable property of a field to *false*. After this, the property won't be encountered when we run a *for…in* loop on the object. The loop written at the end of the snippet prints the property *age* alone.

Conclusion

TypeScript has a number of useful features and we have barely scratched the surface. The concepts explained in this chapter will help you code Angular 2 applications, which make use of TypeScript extensively. To dive deeper into TypeScript, refer to: *https://www.typescriptlang.org/docs/tutorial.html*.

Chapter 3

Angular 2 New Features

A Brief History of Angular

Web development has experienced a number of changes in the last couple of years. Web pages that were rendered completely from the server, got a paradigm shift with technologies like Adobe Flash and Microsoft Silverlight. These plugins added richness to the browsers to improve the user experience. Unfortunately, as browsers were made to understand and render something that was not built into them, they became bulky, inconsistent, and error-prone. However, the richness brought about by each plugin resulted in significant changes to the language of the web(HTML).

With HTML5, browsers absorbed this richness natively along with a number of new elements: storage mechanisms, improvements to APIs, and new technologies for real-time communication. These additions to the browsers over the past few years made them more consistent and advanced than ever before. These changes naturally brought a lot of power to the developers working on front-end technologies.

The availability of this type of richness in HTML5 resulted in a wider usage of JavaScript. Developers around the world started innovating with the combination of JavaScript and the power of HTML5 to build sophisticated web applications. These applications incorporated the power and capability that Flash and Silverlight had previously provided, all the while being truly native applications. JavaScript libraries like jQuery further changed the game by allowing developers an easy way to deal with browser inconsistencies.

Web developers started using these newly acquired powers to their fullest potential. Client-side scripting started getting heavier and the web was more feature-rich than ever. The richness improved each day and eventually Single Page Application (SPA) design pattern started gaining a lot of traction. An SPA moves much of the logic that was previously executed on the server to the client using JavaScript. This approach makes lightweight calls to the server to fetch data and then performs operations like binding data, view rendering, user interaction, and more—all on the client.

Initially, developers could select from multiple libraries specialized to handle different aspects of an SPA, but building a full-fledged SPA was no easy task. Integrating disparate libraries required a lot of work, making the process of building and maintaining the application difficult and steepening the learning curve. What the industry needed was a single framework that seamlessly integrated all or most of the aspects of an SPA.

Angular was released by Google to ease the pain in SPA development. As a single framework responsible for handling all of the aspects needed to build an SPA, it acts as a one-stop-shop for most of the needs of a single page application.

Why Angular?

Angular is a single framework meant to be the basis for easily building an SPA from scratch. While Angular easy to use, it is built to withstand the complexities involved in large projects by naturally lending itself to unit testing, modular design, industry standard design patterns, and best practices.

Angular provides a more expressive way to deal with the presentation layer. Unlike jQuery, which directly manipulates the DOM (making it a challenge to predict changes in the view), Angular provides a way to teach new tricks to HTML via *directives*. For example, consider the following piece of HTML:

```
<calendar year="2016" month="April"></calendar>
```

The moment you read this fragment, you can clearly see that the element is going to render a calendar with April as its month and 2016 as the year. With jQuery, achieving something as declarative as this would be a challenge.

The framework defines a set of general purpose directives and gives developers the option to write custom directives at will. Using directives, you can build your own HTML elements or choose to extend existing elements or directives. The hope for this type of expressiveness is that anyone will understand the most essential parts of an Angular application by simply reading the HTML templates of the application.

Extensibility is one of the most important non-functional requirements of software. Features of Angular like *modularity*, *dependency injection*, and *enable extensibility* make it easy to test Angular code. Angular's modular design naturally avoids using global scope, which makes it easy to build unit tests. It is possible to unit test almost any piece of code written using Angular. What's more, the Angular team will even provide mocks and helpers around most of the built-in objects.

Why a Complete Rewrite?

The first version of Angular, Angular 1.x was built when browsers still had inconsistencies, JavaScript was not matured enough to support large applications, and the tools around front-end development were still in their early stages. Angular 1.x tried to solve most of these problems by building abstractions. Though the framework was good at that time for building large JavaScript applications, the abstractions that did exist made it harder for the framework to adapt to the changes happening around the front end space. Following are some of the key reasons for rewriting the framework from Angular 1 to Angular 2.

After Angular 2 announcement, the library previously known as AngularJS was renamed to Angular.

A CHALLENGE TO EMBRACE NEW FEATURES ON THE WEB

The design of Angular 1 makes it difficult to use many of the advancements in JavaScript and new features of the browser. In many cases, the feature cannot be used unless we create an Angular wrapper around it. For example, any API has to be wrapped around a service and to use a new DOM property, a custom directive would be required. When using features like *promises* in ES6 in an Angular application, you need to keep calling the digest cycle manually to have the page recognize changes in the application. Further, using sophisticated features like *custom Web Components* in an Angular 1 application is even more difficult, as it would require a lot of custom work to make the two paradigms talk to each other.

It is almost impossible to change this behavior by modifying the code of the existing framework, as it involves a substantial amount of change to the initial design. Further, making such a big change to the existing code involves a lot of risk, and, needless to say, much rework in testing and debugging the framework.

HEAVY CHANGE DETECTION SYSTEM

Angular 1 uses dirty checking to keep the model and view in sync and uses watchers to check for the changes made to an object. As a result, pages begin to slow down as the number of watchers grow. Building complex components becomes very complicated as they often make use of binding expressions and filters, which then require additional dirty checking and more watchers.

The *scope* object used for data binding in Angular 1 makes working with controllers and directives unpredictable after a certain extent. This unpredictability can make debugging the application difficult.

Change detection is even more complicated when a third-party library is introduced into an Angular application. It is often quite difficult to use a third party non-Angular library with Angular 1 without wrapping it inside an Angular block. As the library raises events or initiates a change in the Model, you need to manually tell Angular about the event. Otherwise, the change detection doesn't kick in, and any changes made to the Model are not reflected in the View.

INEFFICIENT INTERACTIONS WITH DOM

Angular 1 uses jqLite or jQuery (when available) to perform DOM manipulations. These wrappers around the DOM objects handle browser inconsistencies, but also make the application inefficient as the application pays a performance penalty in order to wrap or unwrap DOM elements. Beyond the technical issues, to be effective in Angular you also must learn jQuery to perform DOM manipulations in Angular 1.

COMPLEX DIRECTIVE STRUCTURE

The Directive Definition Object (DDO) defined in Angular 1 is difficult to learn. It provides too many options and some of them cannot be used together. In addition, Angular 1 directives provide many low-level options that can puzzle even an experienced developer learning Angular for the first time.

All of these reasons combined brought the Angular team to the conclusion that a full rewrite was in order.

New Features in Angular 2

Angular 2 is built from scratch using the latest and greatest features available in the web platform. It is built to support the scale at which JavaScript is used in modern web development to create rich web applications and hybrid mobile applications. The framework went through significant changes to the core design to use the most modern APIs available on the web platform. Angular 2 adapts the conceptual strengths from Angular 1 and implements them using improved approaches. Angular 2 continues to embrace principles like extensibility, testability, modularity, and separation of concerns while making it easier for developers use Angular. The following are a few ways that development is getting easier in Angular 2.

LOOKS MORE LIKE PLAIN JAVASCRIPT

One of the goals of Angular 2 is to fully embrace features of JavaScript. Rather than building its own abstractions, Angular 2 leverages the features of ES6 and ES7 like classes, decorators, modules and various others, so that developers don't miss using the power found in the core of JavaScript.

EFFICIENT CHANGE DETECTION

A rich internet application would be incomplete without a mechanism for keeping objects and views in sync. Angular 2 comes with a much-improved and flexible change detection mechanism. This new system allows applications to take advantage of the observable objects as well as immutable objects. The framework understands how these objects work. Angular doesn't keep checking for changes in the values of such objects. Instead, it uses the notification system built into these objects to detect the changes which gives the application an opportunity to use the best possible change detection-enabled object.

The change detection system doesn't require manual intervention. Angular uses *zones* to manage all operations and zones run on every browser event. Whether it is an asynchronous operation, a user action, or an event from a third party library; zones are smart enough to run the change detection again when these events occur.

REDUCE DIRECT DOM MANIPULATION

Angular 2 embraces improved bindings beyond what is available in Angular 1. Rather than using jqLite or jQuery, Angular 2 uses the browser's APIs for DOM manipulation in an attempt to totally abstract the DOM manipulations from developers. To do so, it embraces bindings and doesn't provide direct access to the DOM elements in most situations. This architecture makes the same piece of code that runs on the web and mobile require a slightly different setup of Angular 2 to support both the platforms.

BETTER TEMPLATING

Angular 2 doesn't wrap the DOM properties and events in individual directives. Instead, DOM properties and events can be data-bound to a model without wrapping it in an Angular block. The DOM property must be enclosed inside square brackets (e.g. [title]), and the DOM event must be enclosed inside parentheses (e.g. (click)) to indicate that they are data bound. This takes away the pain of writing custom code to perform operations when a model, bound to a property, changes. Further, this approach can bind a model to any newly introduced HTML property or event without making any changes to the framework and the application code.

EMBRACING COMPONENTS

An Angular 2 application starts with a component and it goes on building components at every level of the application. The components are built on top of the HTML5 Web Components standards, so they feel more native to the browser now. The browser's DOM APIs can understand these elements and they can operate along with the Angular components. As there is no layer sitting between a component and the browser, they can be easily made accessible.

SEO FRIENDLY

As single page applications are rendered completely on the client side, the web crawlers cannot parse the content and hence the pages become less searchable. To overcome this, Angular 2 supports server-side rendering. The application can be pre-rendered on the server and served on the client side. Since the content delivered over the network is rendered HTML, web crawlers can read this content and search for content in the rendered view.

Conclusion

Angular 2 comes with a set of promising features. It leverages advancements made in modern development and features an extensible model for development. Angular 2 addresses many of the pain points found in Angular 1 today. Nevertheless, it respects the concepts in Angular 1 that made the framework so successful and it builds upon each feature. The forthcoming chapters detail the changes and advancements coming in Angular 2.

Chapter 4

Angular 2 Application Structure

ANGULAR 2 USES *COMPONENTS* extensively. A typical Angular 2 application consists of components to define every part of the page. The components are added to Angular *modules*.

Think of an Angular 2 application as a tree of components. These components are linked together. The top most component is the Root component which contains all other components, and is used by the Angular *bootstrapper* to start the application.

Here's a typical Component tree in an Angular 2 application:

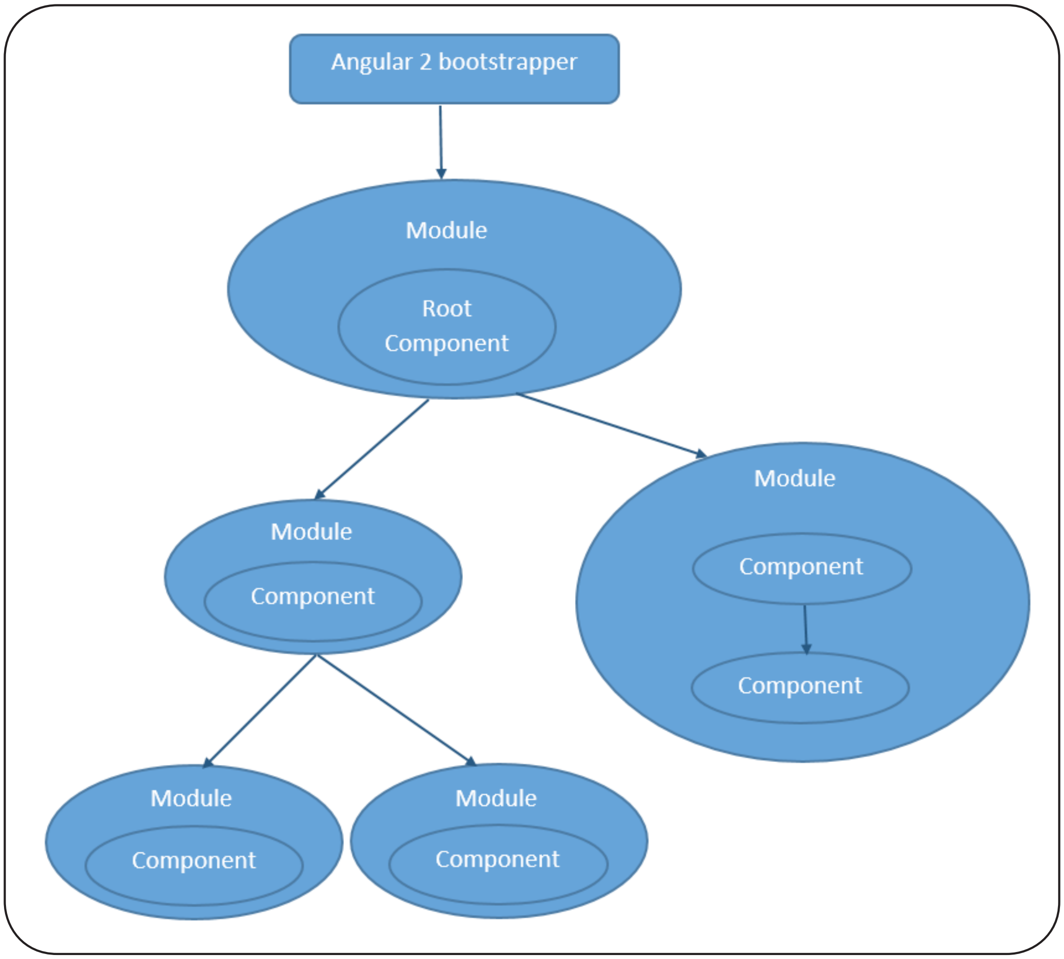


Figure 4.1: *Component tree structure in an Angular 2 application*

Every component is a self-sufficient piece of UI, screen, or route. A component is a combination of a view and its view-model. It uses *Services* to get the data and displays that data in views.

When the application is divided into multiple views and the views have to be loaded based on the current URL, then each route is handled with a component. The route invokes this component when it has to be loaded.

Angular 2 applications are modular. Angular 2 makes use of several ES6 modules to keep the source code modularized, and it uses its own module system to group a set of related Angular blocks together. It is important to understand that the purpose of the ES6 modular systems is different from that of the Angular modules system.

The ES6 module system helps keep the source files as lean as possible. Each file contains a component, a directive, a service, or any other block. These files export their objects using the *export* keyword, and they are imported by other modules using the *import* keyword. The third party libraries used in the application are also loaded as ES6 modules. On the other hand, the Angular modules are used to group a set of Angular 2 blocks together. These modules can be used to split an application into multiple modules based on the functionality of a set of blocks. One module can access other modules to use their functionality.

Angular 2 makes extensive use of Dependency Injection (DI) to load required objects into any code block. It provides a single API for DI, which comes with all the power needed for a complex application. The combination of modularity and DI makes Angular 2 code much cleaner to read and to test.

An Angular 2 application is built using several Components, Services, Directives, Pipes and other pieces. Let's explore what each one does.

Directives

Directives are not new in Angular 2, but have been improved upon since Angular 1. The Directives architecture in Angular 2 reduces the need for direct DOM manipulation by providing a better binding system. Unlike Angular 1, where a directive has to be named in camel case notation and used on the UI with dashed notation, Angular 2 has a unified way of naming and using the directive.

Angular 2 has three types of directives:

Components

An Angular 2 application starts with a component; every route is associated with a component and uses components to define different levels of the application. The components consist of an HTML template as well as the logic to build a view-model for this template. The HTML template uses Angularís binding syntax to bind the properties and methods of the view-model in the view. A component can load another component in its template and interact with it.

Components use the existing features of *HTML5 web components* to bring the functionality closer to the browser. Angular 2 has a built-in emulation to fill these features in unsupported browsers. These features allow the component to be self-sufficient and isolated from rest of the HTML. Components leverage the feature of *Shadow DOM*, thus allowing an option to write styles specific to the component that don't affect the rest of the page.

The components have life cycle hooks like ngOnInit(), ngOnDestroy() that allow the application to respond to key lifecycle events of the component.

DECORATOR DIRECTIVES

Decorator directives extend the behavior of an existing HTML element or an existing component. These are the simplest kind of directives. They perform small sets of actions, but at times these small features are critical for business. A decorator directive can interact with its host via events. Using these events efficiently reduces the amount of DOM manipulation one needs to perform. They can be used for actions like handling certain events, applying a style, validating a value, and similar operations.

STRUCTURAL DIRECTIVES

The structural directives deal with the template rendered inside an element. They can manipulate the template, depending upon the need. It doesn't manipulate the DOM inside the target directly, rather it uses the *ViewComponentRef* service provided by Angular 2 to add or remove elements inside the target. This behavior makes the directive *platform agnostic*.

Directives define lifecycle hooks. They can be used to detect when a lifecycle event occurs and act accordingly. Applications written in TypeScript can use the interfaces corresponding to the lifecycle hooks; doing so will result in better tooling support.

Change Detection

At the heart of every front-end framework is a technique to detect changes made to the objects. Whenever the values of objects are bound on the UI change, the framework needs to be notified so that it can update the UI to reflect these changes. This technique is called *Change Detection*. Angular 2 brings a much more powerful and efficient way to detect changes on the objects. It comes with a built-in change detection mechanism, and allows the applications built on the framework to use a third party technique as well. The framework has an open end that allows the use of objects that provide a better mechanism to detect changes. As of now, the following object types are used with Angular 2:

IMMUTABLE OBJECTS

An immutable object is one that is recreated when any change is made to the object. It has a built-in notification system, which notifies users of the object about changes. Listeners can be attached to these notifications and can perform an action corresponding to the change.

OBSERVABLE OBJECTS

An observable object has a reactive way to notify changes. It emits an event when an object changes its value. Users of the object will have to subscribe to such change events and then perform the required operation.

Angular 2 understands these notification systems. It listens to their events when these objects are bound to the application. The DOM tree, under a component using one of these objects, changes only when the notification is received.

When observables or immutables are not used for binding, Angular uses dirty checking to check for changes in the objects. On every browser event, it kicks in the change detection and the dirty checker scans view models of all the views and applies the changes.

When Angular encounters an observable or an immutable object, it doesn't scan the entire tree for changes. Instead the component sub-tree under a component is scanned when Angular receives a notification from the object about the change. This model makes the application efficient, as it doesn't need to traverse through the entire application on every browser event.

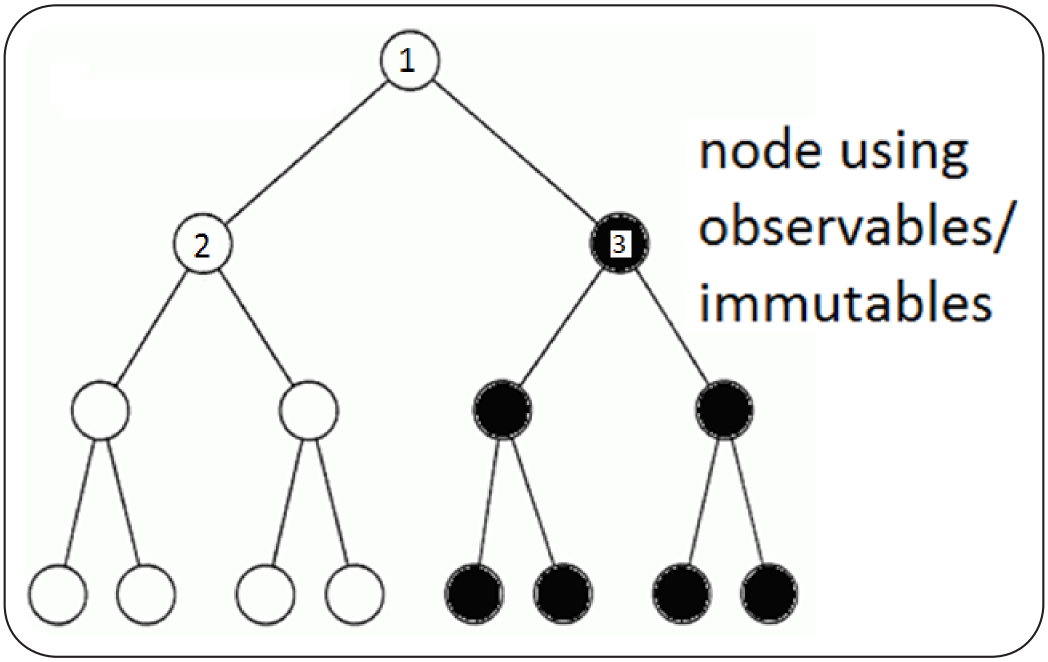


Figure 4.2: *Change detection*

Figure 4.2 shows an example of how change detection works. Each node in the tree is a component. The component 3 uses an observable or immutable type of object and rest of the components are using default JavaScript objects. When data in the components 1, 2, or the child components of 2 are modified, the change detection system checks for any changes in all of the components marked in white. The component 3 and its children are checked for changes only when there is a change in the data in the component 3.

It is possible to use both immutable and observable type of objects in one application. Different components can use different types of objects. The recommended approach by the Angular team is to mix them by using observable for data received from any request sent to a server, and to use immutable for the objects inside the application.

Services

Services are simple ES6 or TypeScript classes which perform an operation like fetching data from an API, maintaining a WebSocket connecting to interact with the server, handling business logic or any reusable logic. A service can be injected into another service, a component, a directive, or any Angular 2 code block. Services help in achieving Single Responsibility Principle (SRP) and keeping the code cleaner.

Pipes

The data received from a source has to be displayed to the user. Sometimes the data received might not make the user happy, as the user may want to see that piece of data in a different format. Pipes are used in such cases. A pipe takes a piece of data, transforms it to a different format, and returns it to the binding expression. Pipes are used in the binding expressions, along with the model used to bind.

Forms

Accepting user inputs is one of the most essential parts of any application. At times, it becomes a challenge to handle the user inputs when the business needs many fields to be filled in by the user and has a lot of validations to perform. Angular 2 provides good support for *Forms*. A form in an Angular 2 application that can either be driven by a template or by a model.

Template-driven forms are composed in HTML and the elements in the form are bound to a model. Behavior of every element in the form can be inspected in its component using the *ngControl* field on the element. The changes on the element can be tracked, value can be validated, and the validation status can be attached using this field. This approach is more declarative as most of the form lives in HTML.

In case of model-driven forms, a form model object is created in the component and is assigned to the HTML form element using the *ngFormModel* property. Angular 2 understands this property and it keeps both the entry in the form model object and the view in sync. The component can make use of the form model object to inspect values in the form.

Routing

The framework that supports Single Page Application development allows switching the views on the client side without refreshing the whole page. It updates a portion of the page and changes the URL so that one can bookmark it, and come back to the specific view instead of starting the navigation again from the first page. This is called client side routing.

Like every SPA framework, Angular 2 supports routing. Angular 2 has a router called *Component Router, which is* named so because it loads components. It provides a way to define routes, provides directives to load a route, and to link a route. *Component Router* can be used to define nested routes. Like directives, a router also provides several life cycle hooks. Using them, routes can be authenticated, data required by the page can be loaded before loading the route, or operations like unbinding an event can be performed.

Dependency Injection

Dependency Injection (DI) is a programming pattern that is used to inject dependencies from an external source, rather than creating them in the code block itself. Angular 2 has its own DI framework. Using DI, the services can be injected into any code block in Angular 2. Unlike Angular 1.x, which had multiple ways to inject dependencies, Angular 2 has a single syntax to inject dependencies. The DI system in Angular 2 also works in a hierarchical fashion. The dependencies of a component have to be declared *before* injecting them. They can be declared either in the module or in the component. Once declared, they can be injected anywhere in the module or at any level under that component.

It is also possible to make a dependency singleton or non-singleton, depending on the need. A dependency can be marked as optional as well; the code block using the dependency is made responsible to handle it when the dependency is not available. Angular 2 provides ways to inject a child class object when a parent class is used for injection, a static object value instead of a newly instantiated object, and dynamic dependencies based on presence of certain values in the environment.

The DI system makes the application testable. DI can be used in the spec files to override the dependencies with mocked objects using some simple APIs. This prevents the tests from hitting the actual definition of the classes, and isolates the tests from rest of the world.

Modules

As mentioned earlier, modules in Angular 2 are used to group a set of related components, directives, pipes, and services together. Angular 2's module system is different from the ES6 module system. The ES6 module system encapsulates the contents of a file, and the Angular 2 module system encapsulates a set of Angular blocks. Here are some features of a module:

- The blocks added to a module can be used inside the module.
- A module can import one or more modules to use the code from that module.
- An Angular 2 library can make use of a module to export its functionality to rest of the world.
- An Angular 2 application bootstraps with a module.
- A module can declare one or more of its components as bootstrap components.
- Execution of the module will begin with the bootstrapped component.

Conclusion

Angular 2 comes with a number of new features, as well as improved versions of some Angular 1 features. Each of these features is designed with ease of use and ease of maintenance in mind. We will dive deeper into each of these concepts as we move forward.

Chapter

5

Angular 2 Development Environment

THE PREVIOUS CHAPTERS (see Chapters 3 and 4) explored some new features that Angular 2 brings to the table. Now that we have a basic understanding of what Angular 2 is all about, let's see these features in action. In this chapter, we will create a development environment or a workspace which will be used as a base for all the upcoming chapters.

Setting Up the Environment

WRITE BETTER SOFTWARE WITH THE RIGHT TOOLS—CHOOSING AN EDITOR

Today there are a wide number of tools and IDEs available for JavaScript and TypeScript development. Some options for Windows, Mac, and Linux users are: Visual Studio Code (Free), Atom (Free), NetBeans (Free), Sublime Text (Commercial for continued use), and WebStorm (Commercial). Tools can be selected depending on the features they provide, as well as our comfort level. For this book, the editor of choice is **Visual Studio Code** (VS Code).

VS Code is a beautiful, light weight, free and open source editor from Microsoft. It is available for Windows, Mac, and Linux. VS Code provides very good support for front-end development in general. For example, VS Code can open any folder containing code; it need not be a special folder with some configurations and project files.

The editor has nice tooling support for TypeScript as it provides good Intellisense, detects presence of type definition files, provides code navigation, reports possible typing errors, and understands ES2015 and ES2016 syntax based on TypeScript configuration in a project. These features make VS Code a good candidate for modern web development. To explore TypeScript, download and install it at: https://www.typescriptlang.org/.

WHY TYPESCRIPT?

Though TypeScript is not the only language supported by an Angular 2 application (Angular 2 applications can be written using both ES2015 and ES2016 versions of

JavaScript and Dart too), it is highly recommended to use TypeScript because of the features we have already discussed in Chapter 2.

Installing Dependencies

Let's take a moment to understand what is needed to build and run an Angular 2 application. For the development environment, the following tools should be installed on the machine and in the project:

1. Node.js
2. TypeScript compiler
3. Angular 2 packages and dependencies of Angular 2
4. SystemJS
5. Koa.js, a node.js based server framework

Out of these, the first two dependencies have to be installed at the system level, and the next three have to be installed in the project.

Node.js is the server platform that provides a way to write server-side code using JavaScript. It is also widely used as a development environment to build front-end applications. If Node.js is not already installed, download the latest version from the official site (http://www.nodejs.org) and install it. Once the installation is successful, Node.js and the Node package manager (npm) will be installed on the system.

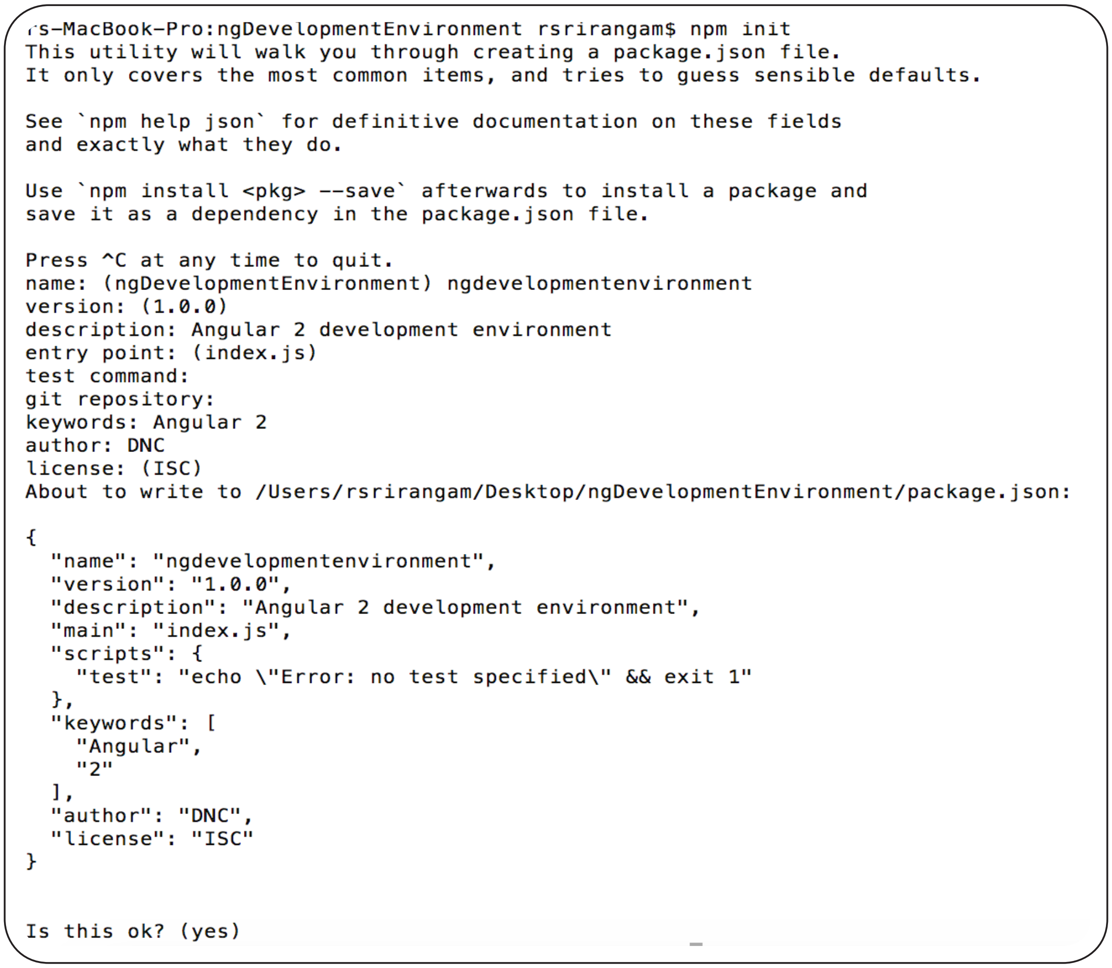
The global npm package of TypeScript must be also be installed. The purpose of the TypeScript packages is to compile TypeScript files from command line. As discussed in chapter 2, TypeScript has to be converted to JavaScript before loading on the browser. The TypeScript package helps in transpiling TypeScript code to JavaScript. If these packages are not yet installed, run the following commands to install them globally:

```
> npm install –g typescript
```

To set up the project, create a new folder and name it *ng2DevelopmentEnvironment*. Feel free to change the name. Open this folder in VS Code. Open a command prompt and move to the newly created folder. As a first step, add *package.json* to the project. This file is to keep track of the installed npm dependencies in the project. There is no need to type this file manually, it can be generated using the following command:

```
> npm init
```

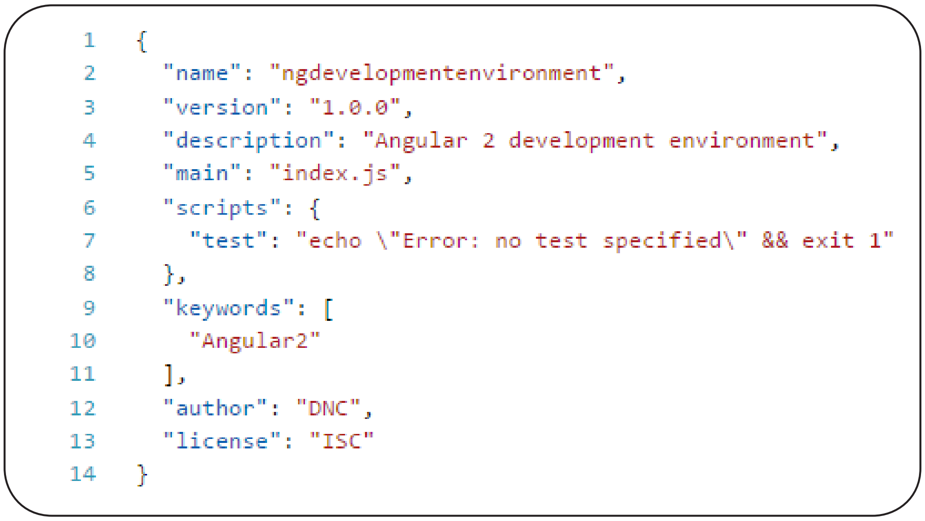
After running this command, answer a couple of questions about different values to be stored in package.json file. Figure 5.1 shows an instance of the command being executed:



```
rs-MacBook-Pro:ngDevelopmentEnvironment rsrirangam$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (ngDevelopmentEnvironment) ngdevelopmentenvironment
version: (1.0.0)
description: Angular 2 development environment
entry point: (index.js)
test command:
git repository:
keywords: Angular 2
author: DNC
license: (ISC)
About to write to /Users/rsrirangam/Desktop/ngDevelopmentEnvironment/package.json:

{
  "name": "ngdevelopmentenvironment",
  "version": "1.0.0",
  "description": "Angular 2 development environment",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "Angular",
    "2"
  ],
  "author": "DNC",
  "license": "ISC"
}


Is this ok? (yes)
```

Figure 5.1 —Initializing package.json

Once the questions are answered, a package.json file gets created as shown in Figure 5.2



```
{
  "name": "ngdevelopmentenvironment",
  "version": "1.0.0",
  "description": "Angular 2 development environment",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "Angular2"
  ],
  "author": "DNC",
  "license": "ISC"
}
```

Figure 5.2: *Initial package.json*

Let's install the required packages in the application. The following commands install these packages:

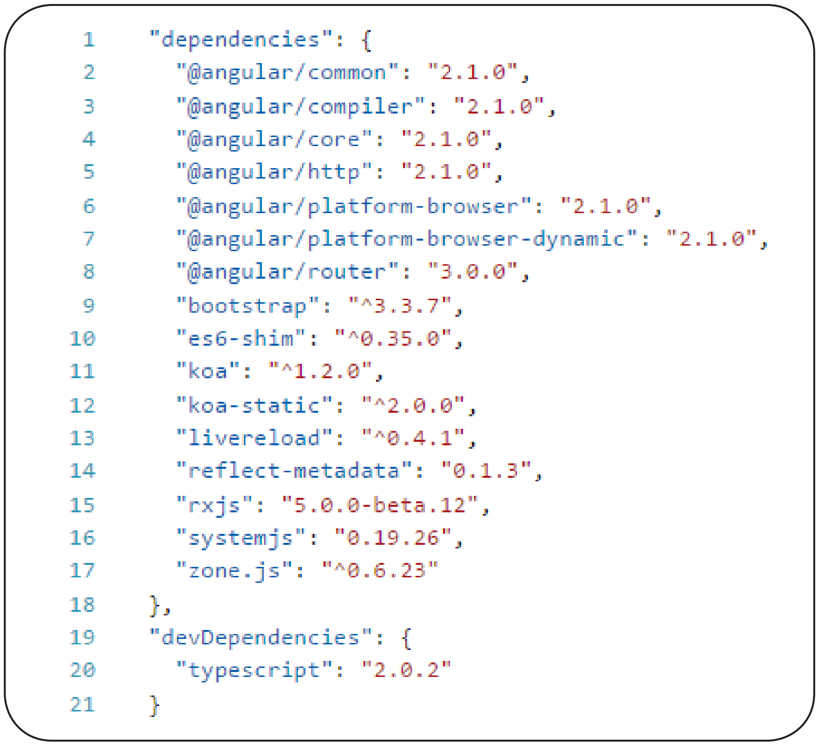
```
> npm install @angular/common --save

> npm install @angular/compiler --save

> npm install @angular/core --save

> npm install @angular/http --save

> npm install @angular/platform-browser --save

> npm install @angular/platform-browser-dynamic --save

> npm install @angular/router --save

> npm install es6-shim --save

> npm install reflect-metadata --save

> npm install rxjs --save

> npm install systemjs --save

> npm install koa --save

> npm install koa-static --save

> npm install livereload --save
```

Though TypeScript is installed globally, having a local version as well gives the flexibility of using multiple versions of TypeScript on the same machine, and the local package comes in handy when the application has to be built on a build system. The following command installs TypeScript:

```
> npm install typescript --save-dev
```

NOTE: The libraries are installed using the --save option and TypeScript is installed using the --save-dev option. The difference is, the packages installed using the --save option are production dependencies, whereas the packages installed using the --save-dev option are required in the development environment. The libraries that are referred in the source code of the application are installed using the --save option. The packages used to create the development tasks like transpiling the code, minifying and uglifying the code, and to watch for changes during development, are installed using the --save-dev option.

The Angular2 package has a couple of peer dependencies, installed along with it and are saved into the package.json file. After running the above commands, the Dependencies section of package.json file looks as shown in Figure 5.3:



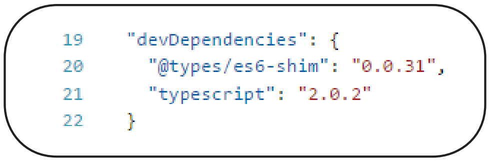
```
"dependencies": {
  "@angular/common": "2.1.0",
  "@angular/compiler": "2.1.0",
  "@angular/core": "2.1.0",
  "@angular/http": "2.1.0",
  "@angular/platform-browser": "2.1.0",
  "@angular/platform-browser-dynamic": "2.1.0",
  "@angular/router": "3.0.0",
  "bootstrap": "^3.3.7",
  "es6-shim": "^0.35.0",
  "koa": "^1.2.0",
  "koa-static": "^2.0.0",
  "livereload": "^0.4.1",
  "reflect-metadata": "0.1.3",
  "rxjs": "5.0.0-beta.12",
  "systemjs": "0.19.26",
  "zone.js": "^0.6.23"
},
"devDependencies": {
  "typescript": "2.0.2"
}
```

Figure 5.3: *dependencies and devDependencies sections*

The TypeScript typings for *angular2*, *rxjs*, *zone.js* and *reflect-metadata* are installed with the packages. The typings of *es6-shim* have to be installed using npm. Run the following command to get the typings of es6-shim:

```
> npm install @types/es6-shim --save-dev
```

It adds an entry to the *devDependencies* section in package.json file, and the modified section is shown in the following figure:



```
"devDependencies": {
  "@types/es6-shim": "0.0.31",
  "typescript": "2.0.2"
}
```

Figure 5.4: *devDependencies after installing types*

NOTE: You might have used the package managers like tsd or typings to install the TypeScript typings. The TypeScript team created the scoped @types package in npm to simplify the process of installing the type definitions. TypeScript version 2.0 and above can work with @types. typings and tsd will continue to work, but the packages may not get the new versions once adoption of @types becomes wider.

Setting Up TypeScript Transpilation

Transpilation is a process of converting the code into a language of the same level. We are already familiar with the term compilation, which is the process of converting code of a language to a lower level language. As both TypeScript and JavaScript are languages of the same level, the conversion process is called transpilation. To see how TypeScript transpilation works, add a sample TypeScript file. Add a new folder to the application and name it *app*. A new folder can be added to the current folder by using VS Code's 'New Folder' button, as shown in Figure 5.5.

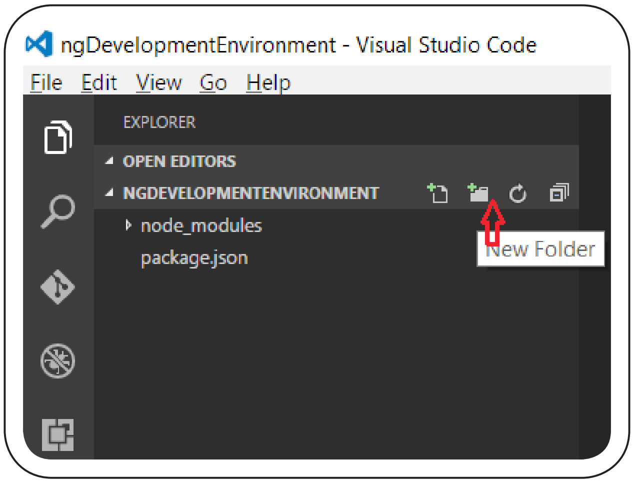


Figure 5.5: *Adding a new folder using VS Code*

Now add a new file to this folder and name it *main.ts*. In VS Code, click on the 'New File' button (see Figure 5.6) or use the context menu.

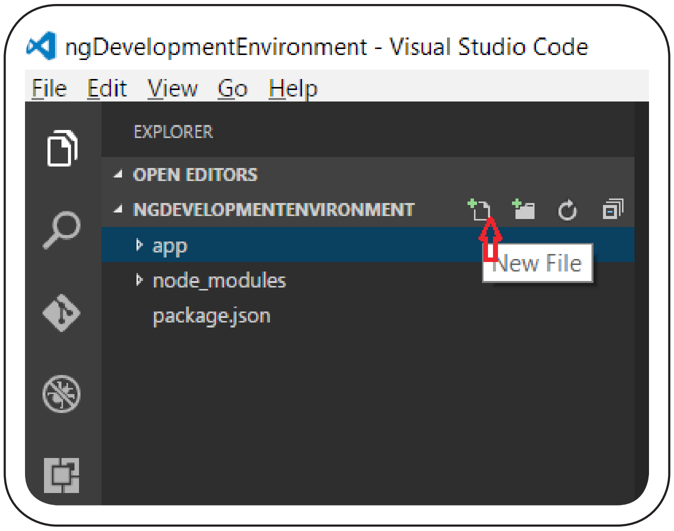
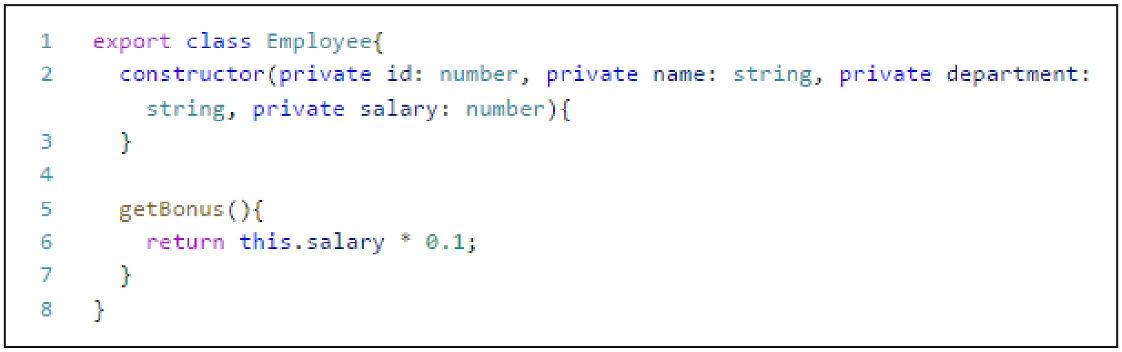


Figure 5.6: *Adding a new file using VS Code*

Let's add a sample class to this class. This class will be removed later.

Listing 5.1: *A sample TypeScript snippet*



```
export class Employee{
  constructor(private id: number, private name: string, private department: string, private salary: number){
  }

  getBonus(){
    return this.salary * 0.1;
  }
}
```

To transpile this file, run the following command:

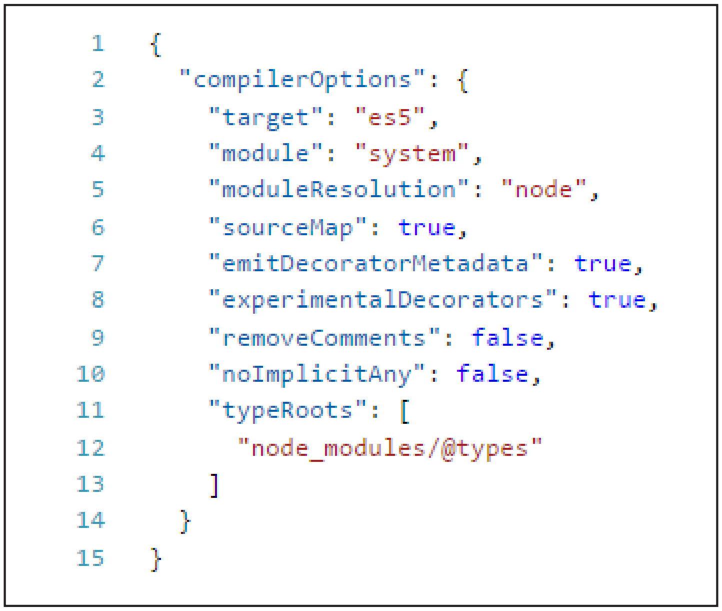
> tsc app/main.ts

This command fails to execute due to the *export* keyword of ES6 module syntax. To resolve this error, provide the ES5 module syntax to produce in the transpiled file. The following command transpiles the TypeScript file to ES5 with SystemJS module syntax:

> tsc app/main.ts --module system

A number of other options can be provided with this command. Adding these options to the command every time may lead to errors. To simplify, create a configuration file to store these options; this file will be referred by TypeScript when the *tsc* command is executed. The following is the content of tsconfig.json file needed for this application:

Listing 5.2: *Content of tsconfig.json*



```
{
  "compilerOptions": {
    "target": "es5",
    "module": "system",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false,
    "typeRoots": [
      "node_modules/@types"
    ]
  }
}
```

Some important parts of the above configuration file are as follows:

- Target version of JavaScript is ES5
- Module system to produce is SystemJS
- ES6 modules are referred from *node_modules* folder along with the current working directory
- Source maps for TypeScript debugging in the browser
- Decorator metadata is included in the transpiled files and experimental decorators are enabled
- typeRoots specifies the folder from where the type definitions have to be referred

Save this file and run the following command from root of the project:

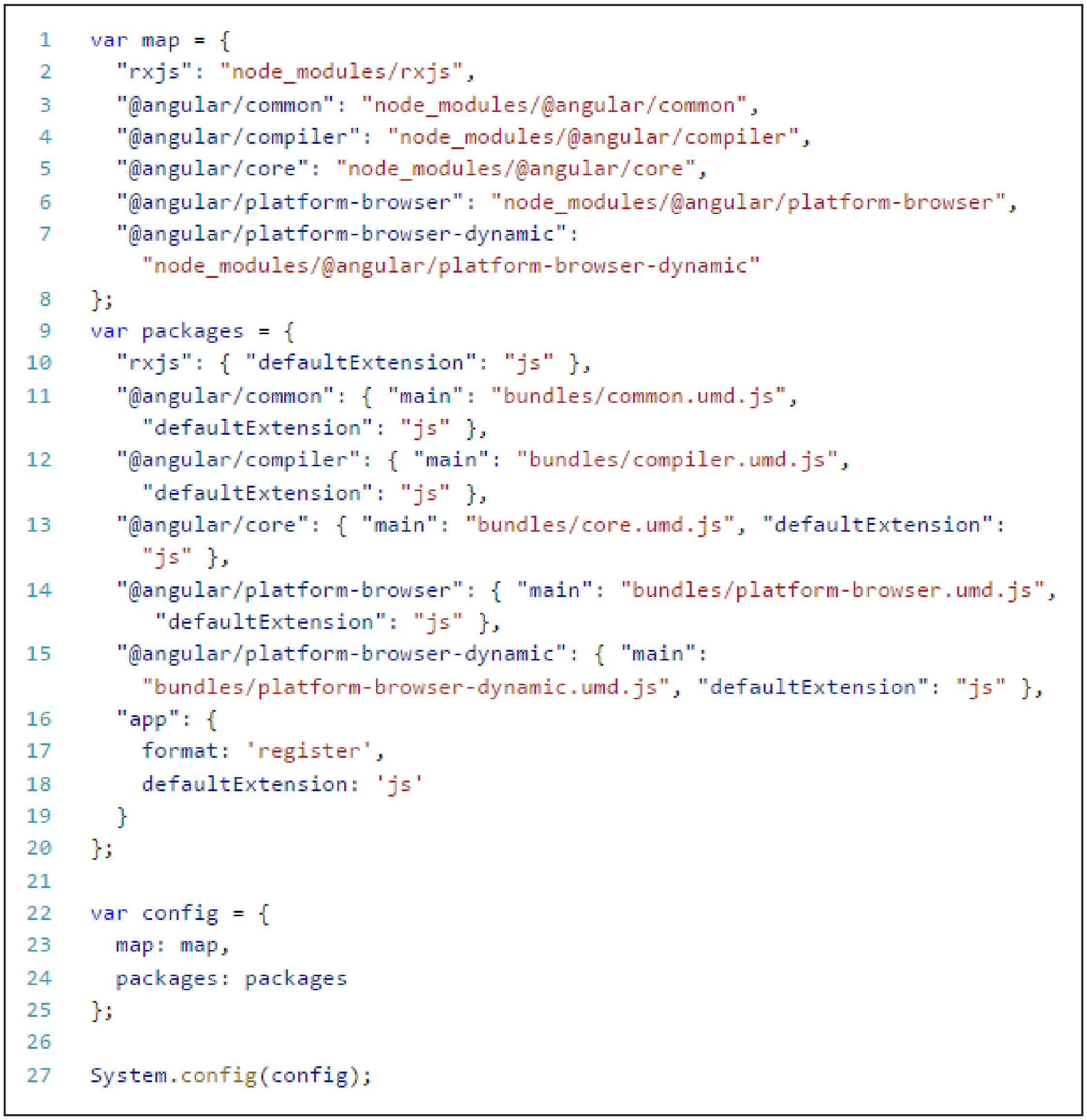
```
> tsc
```

This command transpiles all TypeScript files in the application. As there is just one file present under app folder, it is transpiled, and the transpiled JavaScript file is stored in the same folder along with its source map file.

Setting Up SystemJS Configuration

In order to start the application, load the Angular 2 module files and its dependencies in the right order. These files must be loaded using a module loader. As SystemJS will be used as the module loader, configure the modules needed to be loaded using it. The following snippet shows the SystemJS configuration for the files that need to loaded:

Listing 5.3: *Content of systemjs.config.js*



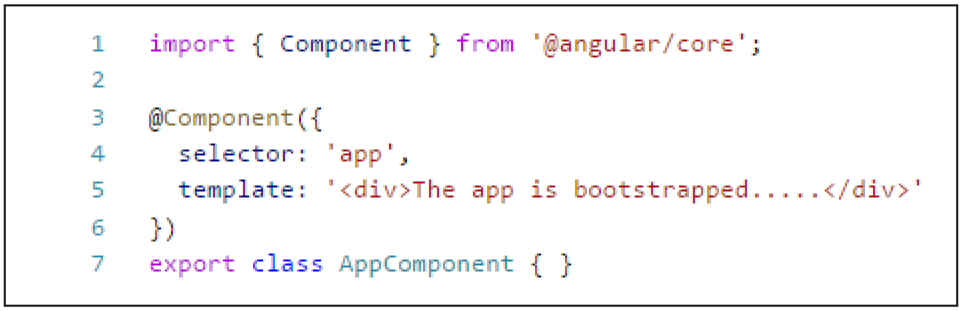
```
var map = {
  "rxjs": "node_modules/rxjs",
  "@angular/common": "node_modules/@angular/common",
  "@angular/compiler": "node_modules/@angular/compiler",
  "@angular/core": "node_modules/@angular/core",
  "@angular/platform-browser": "node_modules/@angular/platform-browser",
  "@angular/platform-browser-dynamic":
    "node_modules/@angular/platform-browser-dynamic"
};
var packages = {
  "rxjs": { "defaultExtension": "js" },
  "@angular/common": { "main": "bundles/common.umd.js",
    "defaultExtension": "js" },
  "@angular/compiler": { "main": "bundles/compiler.umd.js",
    "defaultExtension": "js" },
  "@angular/core": { "main": "bundles/core.umd.js", "defaultExtension":
    "js" },
  "@angular/platform-browser": { "main": "bundles/platform-browser.umd.js",
     "defaultExtension": "js" },
  "@angular/platform-browser-dynamic": { "main":
    "bundles/platform-browser-dynamic.umd.js", "defaultExtension": "js" },
  "app": {
    format: 'register',
    defaultExtension: 'js'
  }
};

var config = {
  map: map,
  packages: packages
};

System.config(config);
```

Listing 5.3 registers maps to the source folders, which will help refer to the folders when maps are used while loading modules. The packages section registers the file to be loaded when a module is loaded, and the default extension to be used (when the module refers to other files as modules). The last entry in the packages object configures the format in which the modules of the application must be loaded, as well as the default extension of the files to be loaded.

Starting Angular 2 Application

Now that the workspace is set up, write a small amount of Angular 2 code to see the workspace in action. To do so, add a component to the application. Add a new file to the folder *app* and name it *app.component.ts*. This component will have an inline template specifying that the application is bootstrapped. Add the following code to this file:

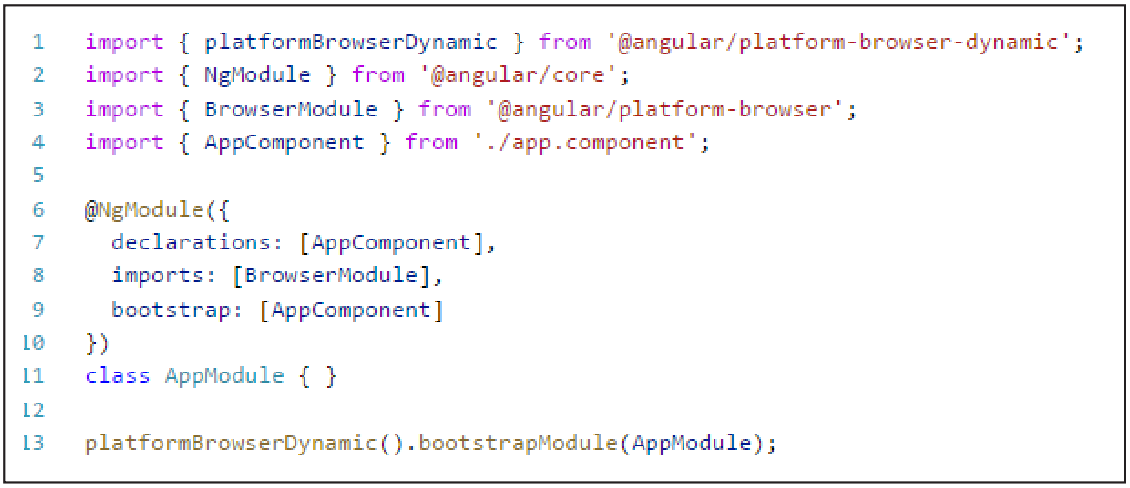
Listing 5.4: *Code of AppComponent*



```
import { Component } from '@angular/core';

@Component({
  selector: 'app',
  template: '<div>The app is bootstrapped.....</div>'
})
export class AppComponent { }
```

The *main.ts* file will have a module to register this component and to bootstrap the application. Mention the component *AppComponent* in the *bootstrap* property of the *NgModule* decorator. Replace the content in the *main.ts* file under app folder with the following code:

Listing 5.5: *Code of main.ts file*

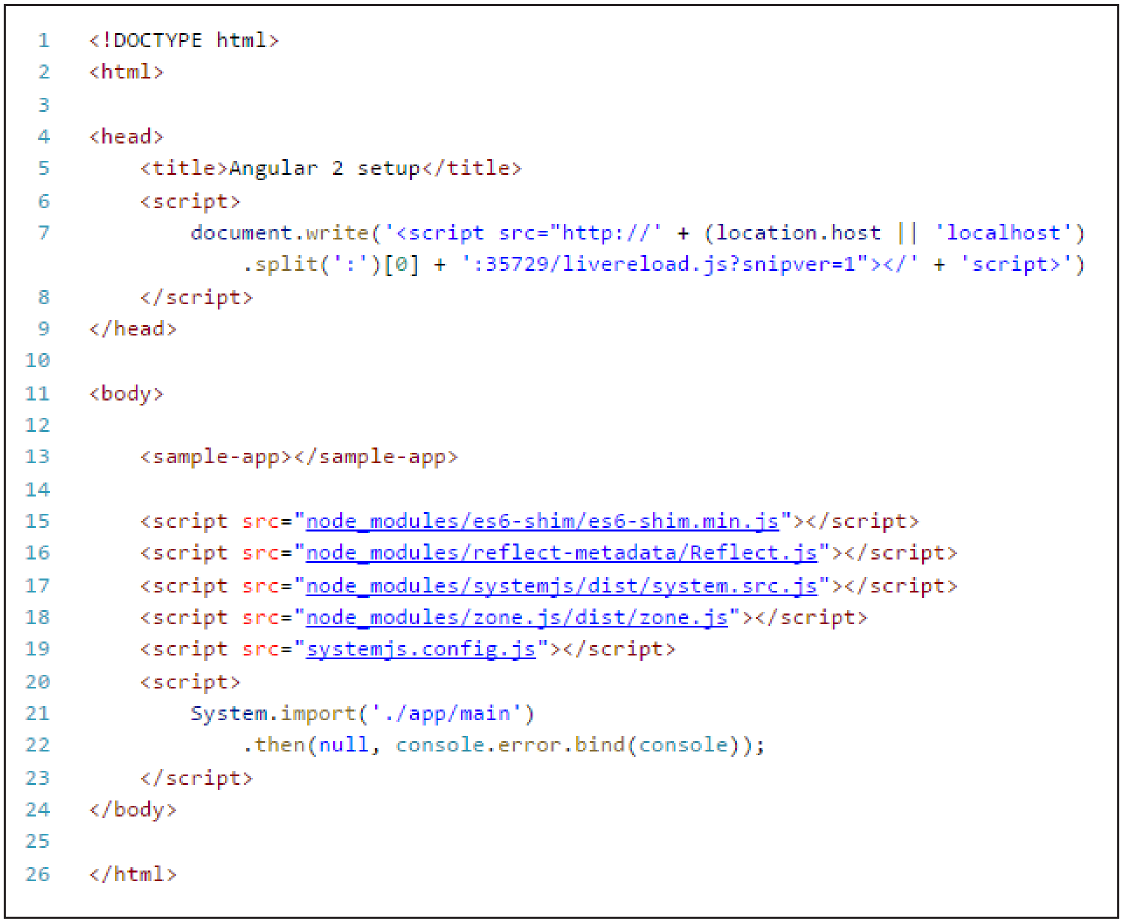


```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  bootstrap: [AppComponent]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

This code defines an Angular 2 component and bootstraps the application using it. It needs the dependencies *bootstrap* and *Component* from Angular 2, so it loads them using the ES6 module import syntax. The component has an inline HTML template, which will be rendered inside the component. The class is passed into the *bootstrap* function, which starts the application using this component. Creating a component will be discussed in detail in Chapter 6.

To see things in action, add a HTML page to load this file. Add a new file to the application and name it *index.html and add* the following content to it:

Listing 5.6: *Code of index.html*



```
<!DOCTYPE html>
<html>

<head>
    <title>Angular 2 setup</title>
    <script>
        document.write('<script src="http://' + (location.host || 'localhost')
            .split(':')[0] + ':35729/livereload.js?snipver=1"></' + 'script>')
    </script>
</head>

<body>

    <sample-app></sample-app>

    <script src="node_modules/es6-shim/es6-shim.min.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="systemjs.config.js"></script>
    <script>
        System.import('./app/main')
            .then(null, console.error.bind(console));
    </script>
</body>

</html>
```

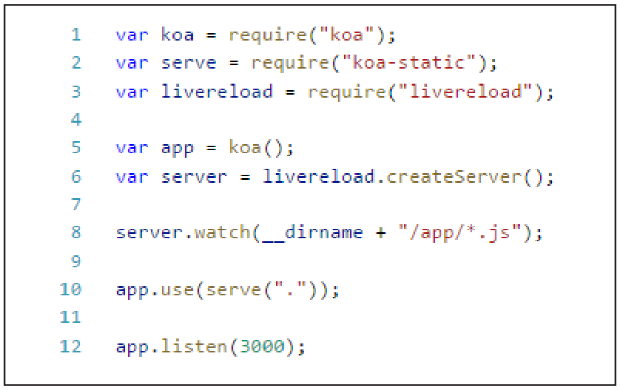
This file loads all of the required libraries and polyfills to run Angular 2 on most of the browsers. The SystemJS configuration file created in the previous step is loaded, followed by the main module of the application. This module, in turn, loads the modules of the Angular 2 framework.

Since SystemJS loads the files using XMLHttpRequest (XHR), a web server is needed to run the application.

Setting Up A Node.js Server

The application needs a Node.js server to start serve the index.html file and a few APIs. The npm packages for the server are already installed in the project. The code required to setup the server has been provided in the downloadable files. Copy the file *server.js* from the sample and paste it in your project. This file has the following content:

Listing 5.7: *Code of server.js*



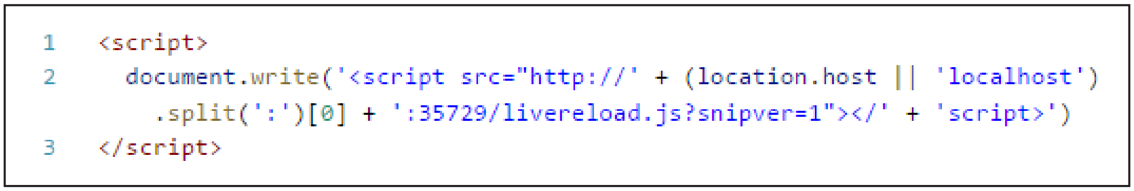
```
var koa = require("koa");
var serve = require("koa-static");
var livereload = require("livereload");

var app = koa();
var server = livereload.createServer();

server.watch(__dirname + "/app/*.js");

app.use(serve("."));

app.listen(3000);
```

This file starts a Node.js server from the current working directory on port 3000. It allows loading any static content relative to the current directory. The server also starts a *livereload* server to refresh the browser when there is a change in any of the JavaScript files under app folder.

> **NOTE:** As mentioned earlier, this sample uses Koa.js, a Node.js based framework. It is a light-weight, robust framework, built using the features of JavaScript introduced in ES2015 and above. To learn more about Koa, you may *visit the official site*: http://koajs.com.

Add a script element to index.html to refresh the browser whenever *livereload* detects a change.

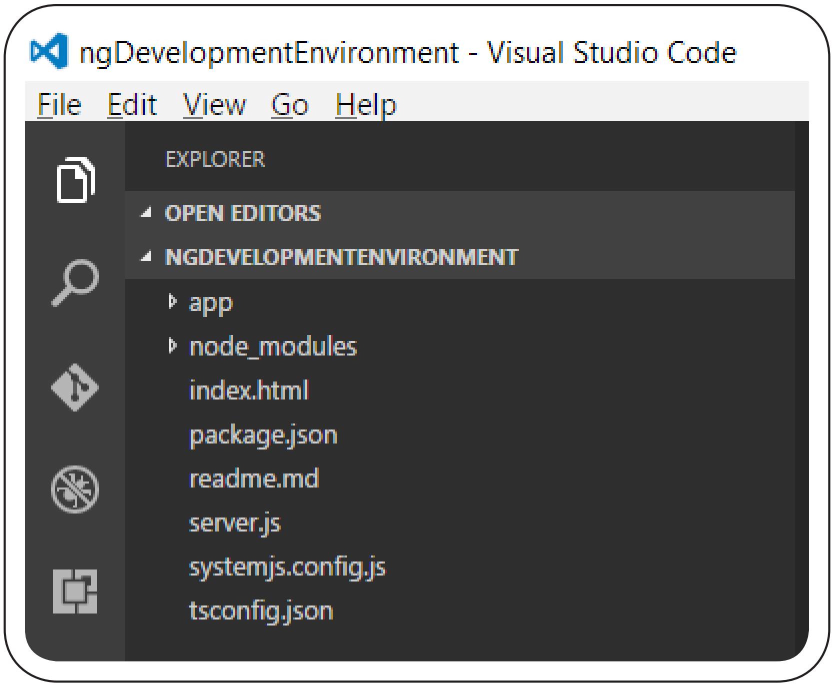
Listing 5.8: *Script tag for livereload*



```
<script>
  document.write('<script src="http://' + (location.host || 'localhost')
    .split(':')[0] + ':35729/livereload.js?snipver=1"></' + 'script>')
</script>
```

livereload is an implementation of the LiveReload server in Node.js, which monitors files for changes and reloads your web browser.

All the required files to run our demo are now present in the application. At this stage, the file explorer pane of VS Code looks like Figure 5.7:



ngDevelopmentEnvironment - Visual Studio Code

File Edit View Go Help

EXPLORER

OPEN EDITORS

NGDEVELOPMENTENVIRONMENT

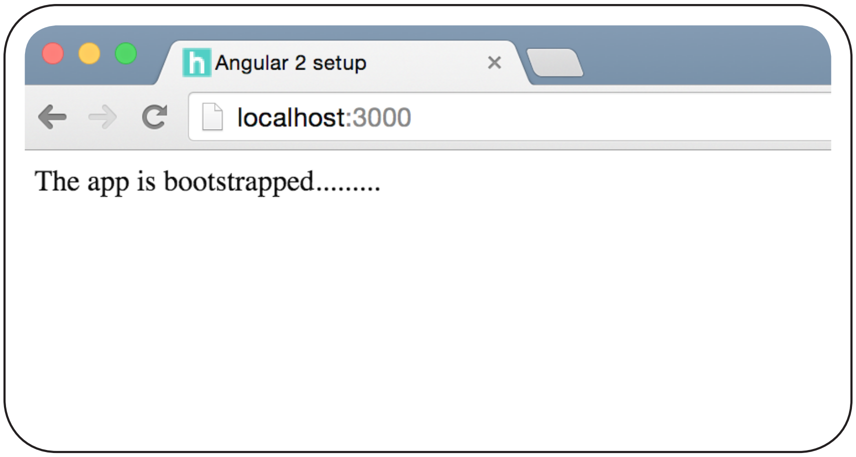
- app
- node_modules
- index.html
- package.json
- readme.md
- server.js
- systemjs.config.js
- tsconfig.json

Figure 5.7: *Files and folders in the project*

Run the following command on a command prompt to start the application:

```
> node server.js
```

Open a browser and change the URL to *http://localhost:3000*. The browser loads the page and displays the Angular 2 component on the page, as shown in Figure 5.8.



Angular 2 setup

localhost:3000

The app is bootstrapped.........

Figure 5.8: *The running application*

Note that making a change to main.ts file doesn't refresh the browser, because we are watching for changes made to the JavaScript files, and changes made to the TypeScript file are not transpiled yet. Open another command prompt, move to the folder where the application is, and run the following command:

```
> tsc -w
```

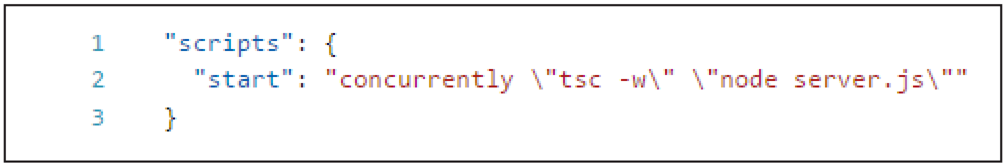
This command adds a watcher on all TypeScript files except the files mentioned in exclude section of *tsconfig.json* file. Now make a change to the TypeScript file to see the changes on the browser.

Running these two commands separately is tedious. To simplify, both of these commands can be run together using the npm package *concurrently*. Install this package globally.

```
> npm install -g concurrently
```

Add the following statement to the scripts section in the *package.json* file:

Listing 5.9: *scripts section in package.json*



```
"scripts": {
  "start": "concurrently \"tsc -w\" \"node server.js\""
}
```

Invoke this section using the following command:

```
> npm run start
```

Conclusion

Now that we are ready with an Angular 2 setup, we can explore the new features. Please note that this setup is just for development and it doesn't suffice for deploying the project. Webpack is a popular tool for setting up production ready environments. We will discuss a more sophisticated set up using Webpack in Chapter 41.

Chapter

6

Introduction to Components

COMPONENTS FORM THE BACKBONE of Angular apps and define the individual parts of an Angular 2 application. As previously mentioned, an Angular 2 application starts with a component and loads components at every level in the application.

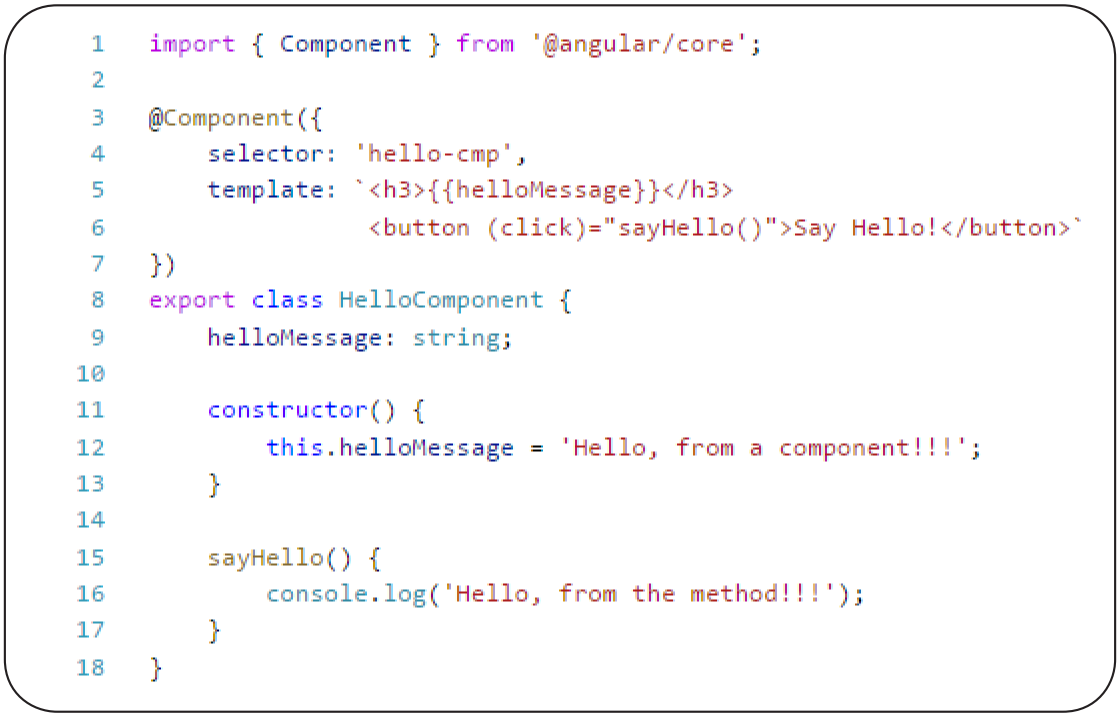
Any Angular 2 application can be described as a collection of a set of components that are created to achieve a task together.

Components are used to create new HTML elements. They are independent, reusable, and are built on the concepts of HTML5 web components, making them very compatible with all modern browsers.

Anatomy of Components

An Angular 2 application consists of a number of modules and one of these components is responsible to bootstrap the application. The module will have a set of components registered in it, one of which will be used to render the first view to the user. This component, in turn, will load other components of the application. This process makes the components the most important pieces of an Angular 2 application. This chapter will introduce you to the components and explain the basic blocks of a component.

A component is a JavaScript class with a decorator. The decorator defines how the component should look on the page, how it has to be used, and the dependencies of the component. The following snippet defines a component:



```
 1  import { Component } from '@angular/core';
 2
 3  @Component({
 4      selector: 'hello-cmp',
 5      template: `<h3>{{helloMessage}}</h3>
 6                  <button (click)="sayHello()">Say Hello!</button>`
 7  })
 8  export class HelloComponent {
 9      helloMessage: string;
10
11      constructor() {
12          this.helloMessage = 'Hello, from a component!!!';
13      }
14
15      sayHello() {
16          console.log('Hello, from the method!!!');
17      }
18  }
```

Figure 6.1: *A simple component*

Let's examine this snippet piece by piece. The numbers listed here correspond with the numbers listed in Figure 6.1.

- **Statement 1** imports the member *Component* from the *@angular/core* module using ES6 module syntax. This module is defined in the Angular 2 library.
- **Statement 8** defines the class *HelloComponent* with a field named *helloMessage* and a method *sayHello* in it.
- **Statement 3** adds the decorator *Component* to this class (@Component). The decorator tells Angular that the class has to be treated as a component.
- **Statement 4** adds the selector property to the decorator. This selector has to be used in the HTML view to render the component. Unlike Angular 1, where the selector has camel case notation in directive's definition, and dashed notation in the template; Angular 2 uses the same notation to both define and use in a template.
- **Statement 5** assigns a template to the component. Notice that the template uses the field and the method defined in the *HelloComponent* class. The parentheses surrounding the click on the button tag `(click)` indicates that we are binding an event. This syntax can be used with any HTML event. The field *helloMessage* is used with the double curly braces in the template, it is called interpolation. It binds the value of the field on the page when it is rendered.

In Angular 1, the controllers built the view model and views were wired up with their corresponding controllers while defining routes. Angular 2 combines these two pieces into a component. An object of the component class is used as a view model, and it is used for binding data in the view. The properties and methods of the component class are directly accessible in the template.

In this example, the template uses the property in the class in a binding expression. Angular does the following when a component is used in a view:

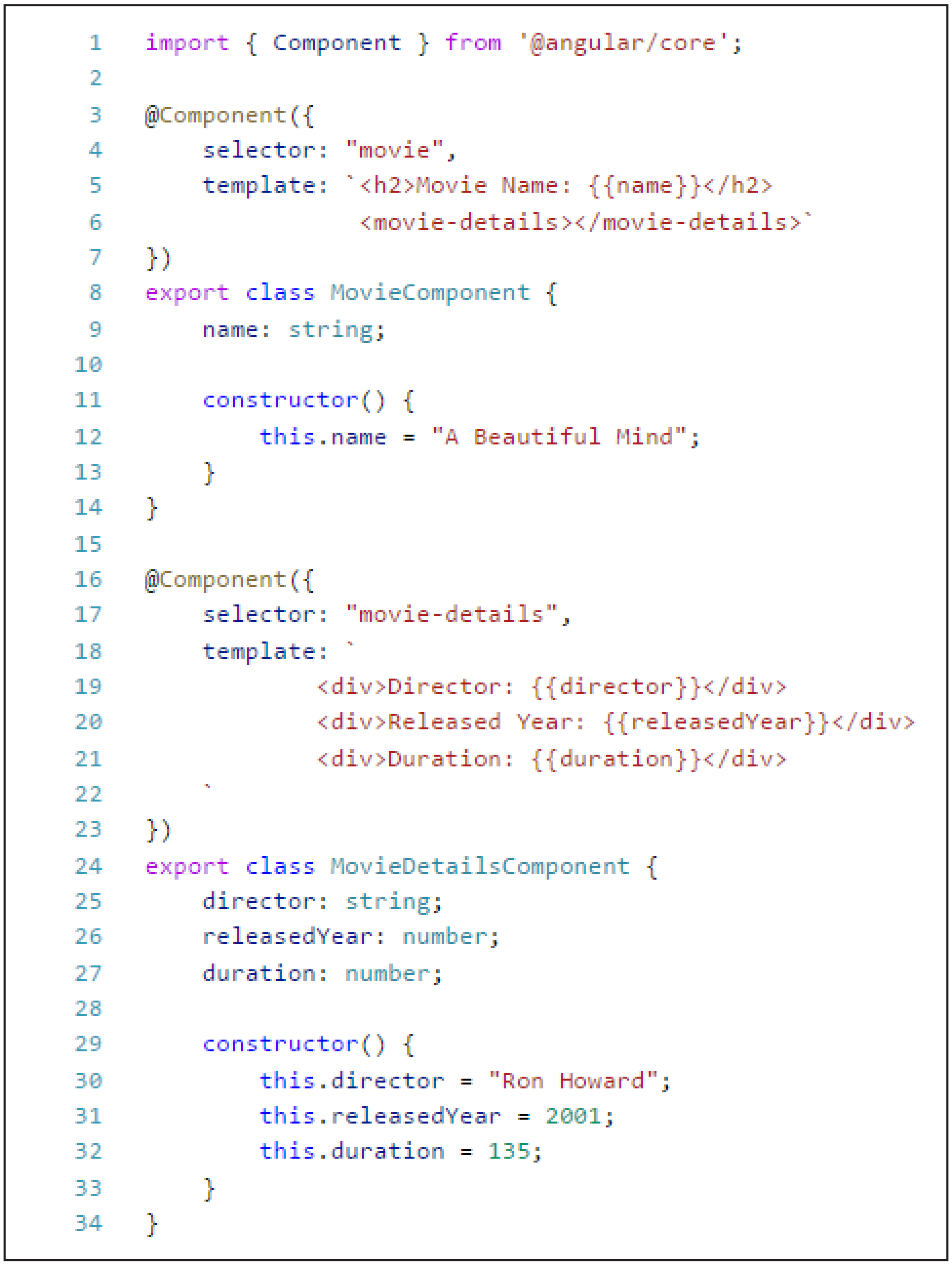
- Loads the template used by the component
- Creates an object of the component class and compiles the HTML template using this object as the source of data. All the properties and methods defined as members of this class are accessible directly inside the template
- Appends the compiled content inside the component element

If you are coming from an Angular 1 background, take note that in Angular 2, a component is actually a View Component as it encapsulates the template, data, and behavior of a view. There is no Controller in Angular 2. Component contains the view as well as the selector, thereby mimicking the behavior of a Directive.

Declaration of Components

When a component has to use another component in its template, ***the second component must be made available to the first component***. The second component has to be declared in the module of the first component. If the second component is declared in another module, the module must be imported into the module of the first component. Otherwise, the component won't be compiled with the template.

Listing 6.1: *A Component using another component*



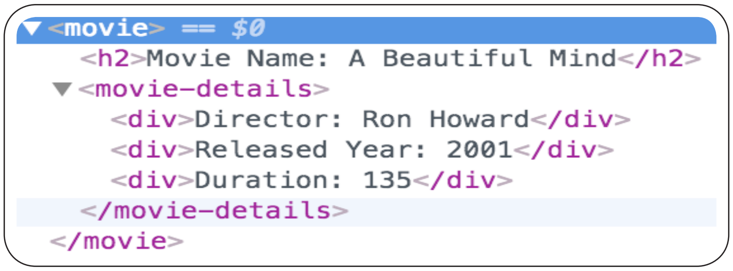
```
import { Component } from '@angular/core';

@Component({
    selector: "movie",
    template: `<h2>Movie Name: {{name}}</h2>
               <movie-details></movie-details>`
})
export class MovieComponent {
    name: string;

    constructor() {
        this.name = "A Beautiful Mind";
    }
}

@Component({
    selector: "movie-details",
    template: `
            <div>Director: {{director}}</div>
            <div>Released Year: {{releasedYear}}</div>
            <div>Duration: {{duration}}</div>
    `
})
export class MovieDetailsComponent {
    director: string;
    releasedYear: number;
    duration: number;

    constructor() {
        this.director = "Ron Howard";
        this.releasedYear = 2001;
        this.duration = 135;
    }
}
```

As seen in Listing 6.1, the component *MovieComponent* uses the component *MovieDetailsComponent* in it. However, when *MovieComponent* is rendered on the view, it doesn't show the contents of *MovieDetailsComponent*. Why is that so? Though the *MovieDetailsComponent* was used in the template, it was not declared in metadata of *MovieComponent* that it will use the component *MovieDetailsComponent*. To fix this, metadata annotation of *AppModule* has to be modified as shown in listing 6.4.

Listing 6.2: *Registering components in a module*

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { MovieComponent } from './movie.component';
import { MovieDetailsComponent } from "./moviedetails.component";

@NgModule({
  declarations: [MovieComponent, MovieDetailsComponent],
  imports: [BrowserModule],
  bootstrap: [MovieComponent]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Now the *MovieDetailsComponent* is rendered in the *MovieComponent*. Figure 6.2 shows the mark-up generated inside *MovieComponent* when it is rendered on a browser. This screenshot is taken using the developer tools of Google Chrome (Ctrl + Shift + J):



```
▼<movie> == $0
    <h2>Movie Name: A Beautiful Mind</h2>
  ▼<movie-details>
      <div>Director: Ron Howard</div>
      <div>Released Year: 2001</div>
      <div>Duration: 135</div>
    </movie-details>
  </movie>
```

Figure 6.2: *Nested components in inspect window*

Life Cycle of Components

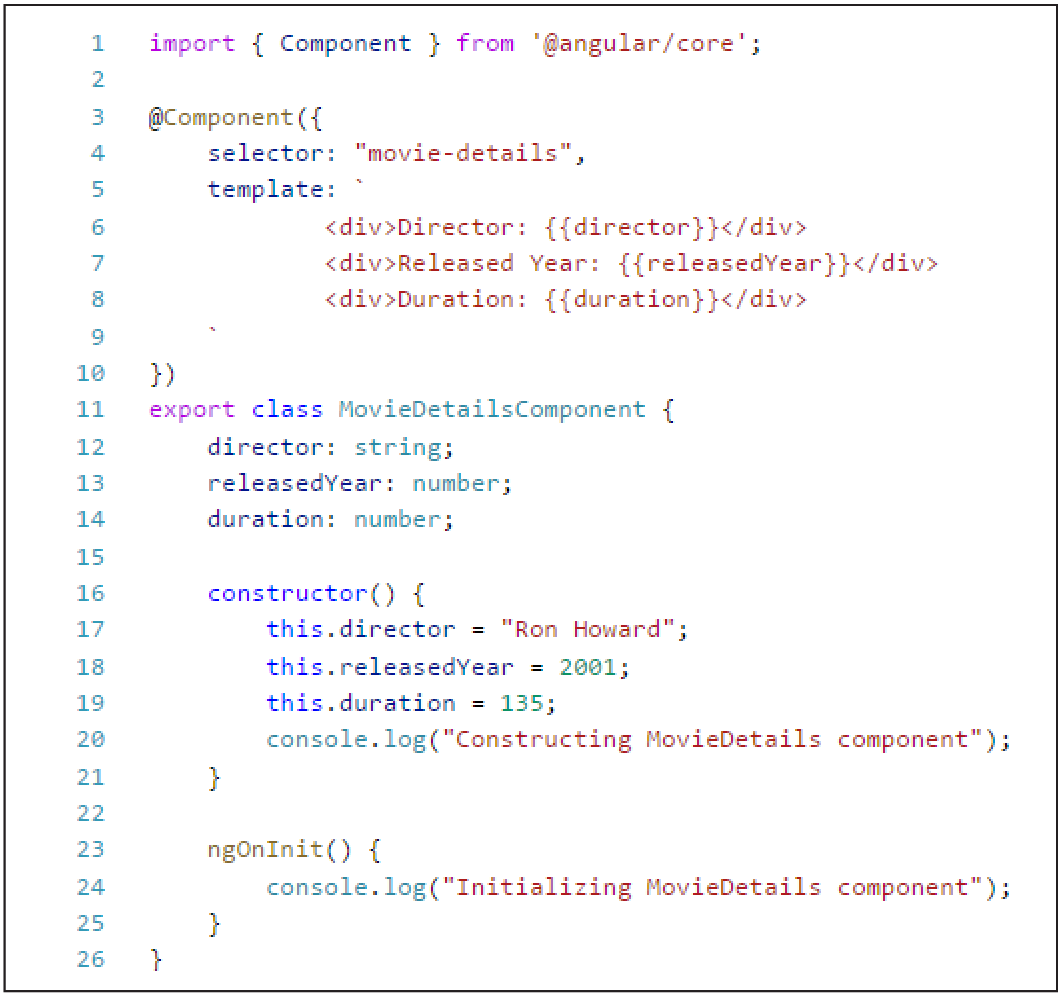
Every component follows a life cycle and Angular provides hooks to these life cycle events. These hooks can be used to detect when a certain life-cycle event occurs and help in performing certain actions during the course of the events.

A component goes through the following phases:

- It gets created when an occurrence is encountered
- Rendered on the page after compiling the template
- Follows the above steps for each child component it contains
- Processes changes when any change in data of the component is detected
- Destroyed when the component is removed from the page

Let's use one of the life cycle hooks of the components in the *MovieDetailsComponent*. Listing 6.4 shows code of the modified component:

Listing 6.3: *Component with Lifecycle hook*



```
import { Component } from '@angular/core';

@Component({
    selector: "movie-details",
    template: `
            <div>Director: {{director}}</div>
            <div>Released Year: {{releasedYear}}</div>
            <div>Duration: {{duration}}</div>
    `
})
export class MovieDetailsComponent {
    director: string;
    releasedYear: number;
    duration: number;

    constructor() {
        this.director = "Ron Howard";
        this.releasedYear = 2001;
        this.duration = 135;
        console.log("Constructing MovieDetails component");
    }

    ngOnInit() {
        console.log("Initializing MovieDetails component");
    }
}
```

When the component is used, it first calls the constructor, followed by the *ngOnInit* method because the data of the view is made available before initializing the component.

The interface *OnInit* and other interfaces for life cycle hooks are defined in the *angular2/core* module. Implementing these interfaces is optional to handle a life cycle event; the interfaces are used to guarantee the signature of the life cycle method.

Conclusion

Components, at the core of any Angular 2 application are: easy to create and use, make markup of the application look declarative, are quite powerful, and can be used to make the application highly extensible. We continue to explore additional features of Components throughout the book.

Chapter

7

Creating a Custom Component

COMPONENTS ARE THE CORE of any Angular 2 application. An Angular 2 application cannot be started without a component, nor can any subsequent operation in the application be performed without one. Chapter 6 outlined the basics of a component and explained the API to create a component. Chapter 7 will extend that knowledge by building a more meaningful component.

This chapter will demonstrate an example of a Master-Detail view to explain components. The component will list orders placed by customers and, when the user wants to view the details of each order, it will show the list of items in each order.

The new component will be added to a new folder to the development environment created in Chapter 5. The following software and packages should be installed in the development environment:

- **Node.js:** To start a web server to run the application, and for the Node.js Package Manager (npm)
- **TypeScript compiler:** To compile TypeScript
- **TypeScript typings installer:** To install TypeScript typings of the libraries used in the application
- **Angular 2 and its dependencies:** The libraries required for the project. These are to be installed in the project using npm
- **SystemJS:** To load modules. To be installed in the project using npm
- **Koa.js:** A Node.js framework to start a Node.js server

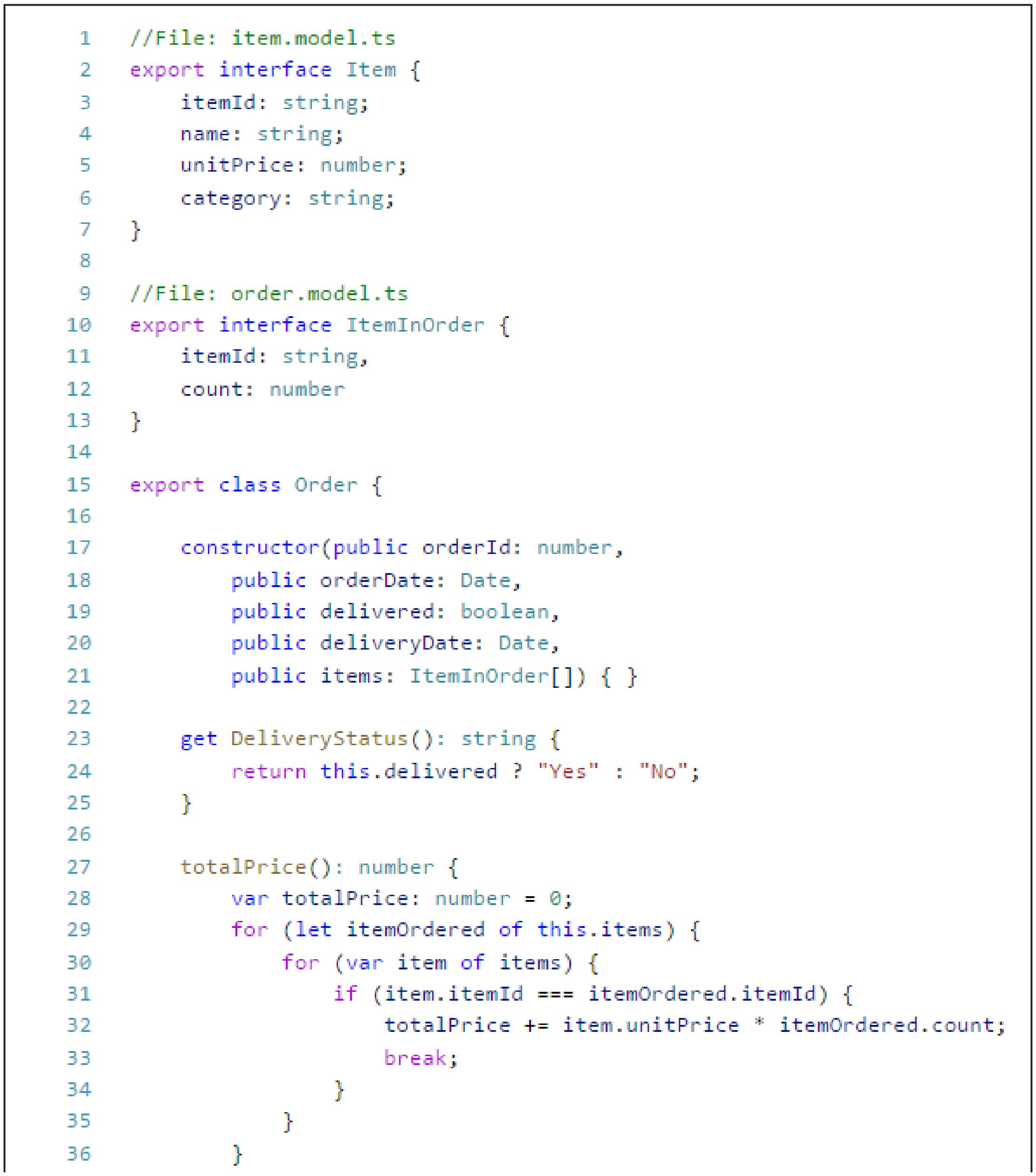
The process of installing these dependencies and starting an Angular 2 project has already been explained in **Chapter 5—Angular 2 Development Environment**.

Create a folder named **Master-Detail Component.** Initialize the folder to build an Angular 2 application. Alternatively, you can copy the files and folders of the setup project into it and install the npm packages. Open the folder in Visual Studio Code and make sure that the sample runs.

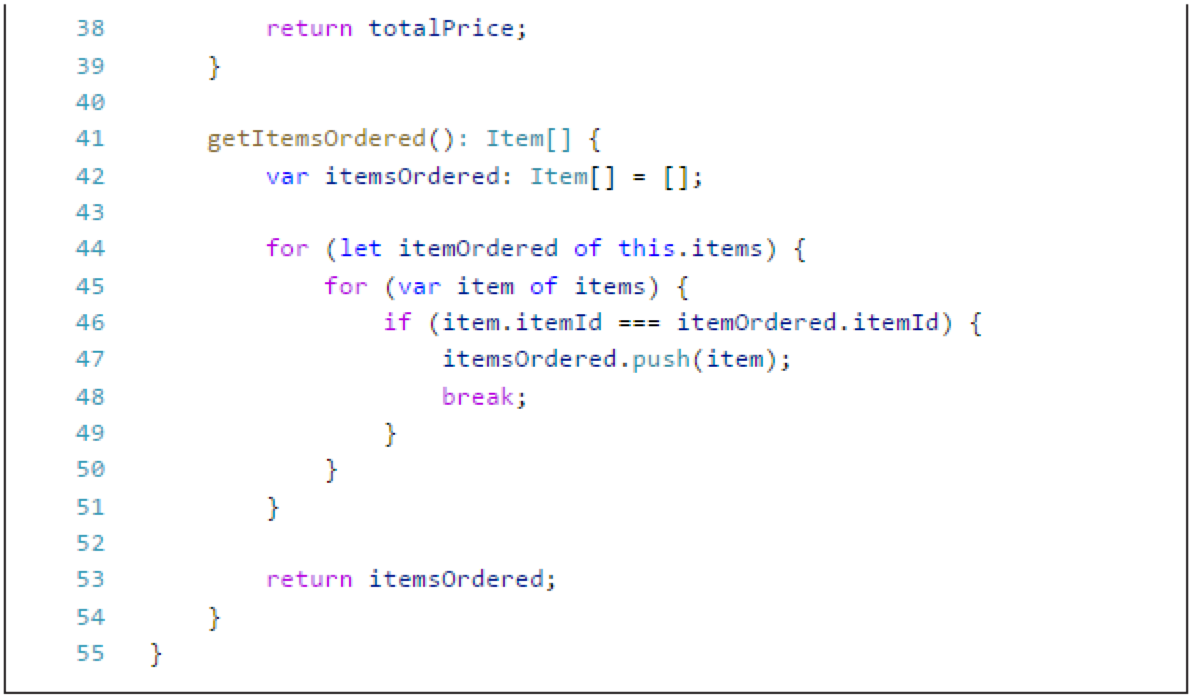
Data and Types for the Component

To embrace strong typing of TypeScript, the application needs some classes and interfaces to represent data types for orders and items. For this application, we need two interfaces to represent the model of an item, an item in order, and a class to represent order and to handle some functionalities for the order. Add two files to the project named item.model.ts and order.model.ts. The listing 7.1 shows the contents of these files:

Listing 7.1: *Model classes and interfaces*



```
//File: item.model.ts
export interface Item {
    itemId: string;
    name: string;
    unitPrice: number;
    category: string;
}

//File: order.model.ts
export interface ItemInOrder {
    itemId: string,
    count: number
}

export class Order {

    constructor(public orderId: number,
        public orderDate: Date,
        public delivered: boolean,
        public deliveryDate: Date,
        public items: ItemInOrder[]) { }

    get DeliveryStatus(): string {
        return this.delivered ? "Yes" : "No";
    }

    totalPrice(): number {
        var totalPrice: number = 0;
        for (let itemOrdered of this.items) {
            for (var item of items) {
                if (item.itemId === itemOrdered.itemId) {
                    totalPrice += item.unitPrice * itemOrdered.count;
                    break;
                }
            }
        }
```



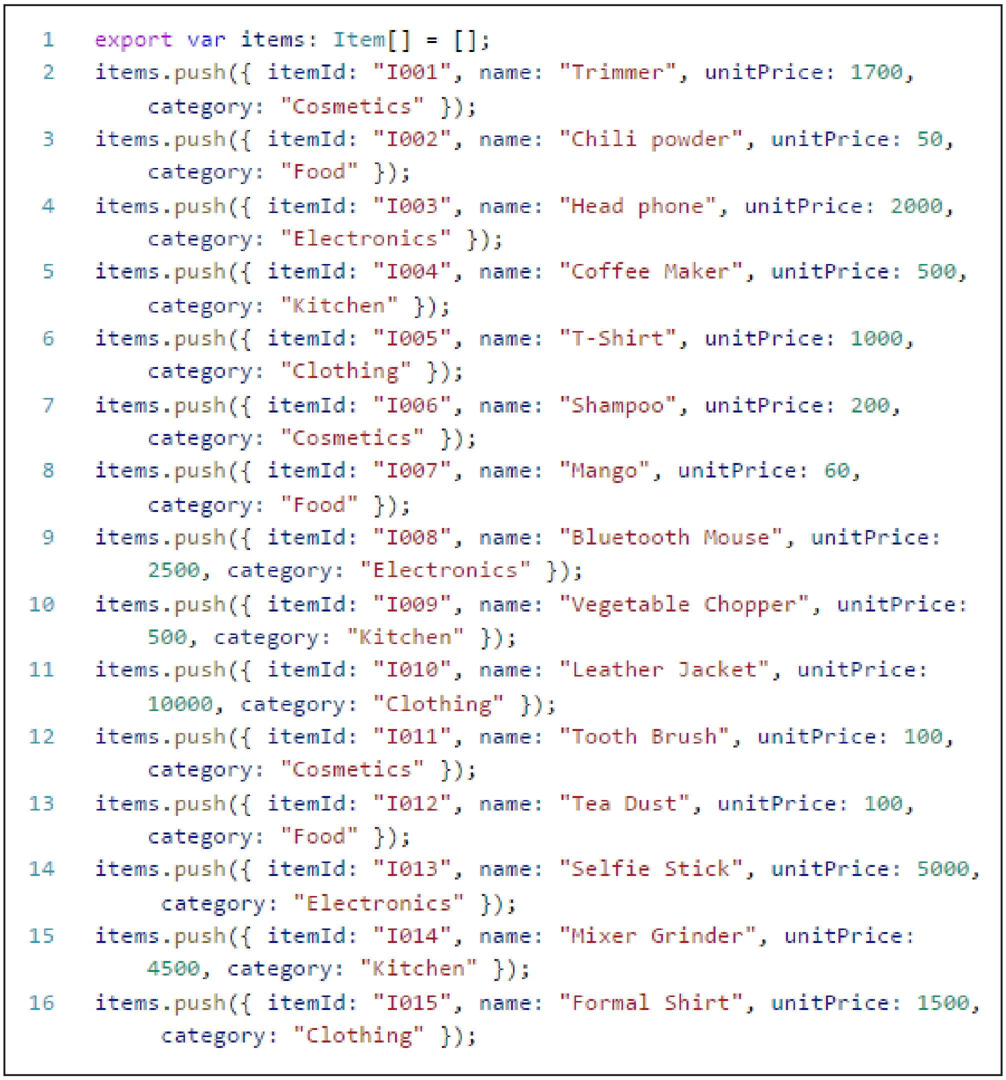
```
        return totalPrice;
    }

    getItemsOrdered(): Item[] {
        var itemsOrdered: Item[] = [];

        for (let itemOrdered of this.items) {
            for (var item of items) {
                if (item.itemId === itemOrdered.itemId) {
                    itemsOrdered.push(item);
                    break;
                }
            }
        }

        return itemsOrdered;
    }
}
```

The above code defines the following types:

- *ItemInOrder*: Interface representing model of an item added to an order
- *Item*: Interface representing model of an item
- *Order*: Class that can hold an order object. It has a method to calculate total price of an order, get details of all items added to it, and a *getter* block encapsulating status of the order

All three types are exported out of their modules, so that they can be used by other modules. The data required for the application will be stored in two static lists. The items array will be added to the file *item.model.ts*. The following listing shows the array:

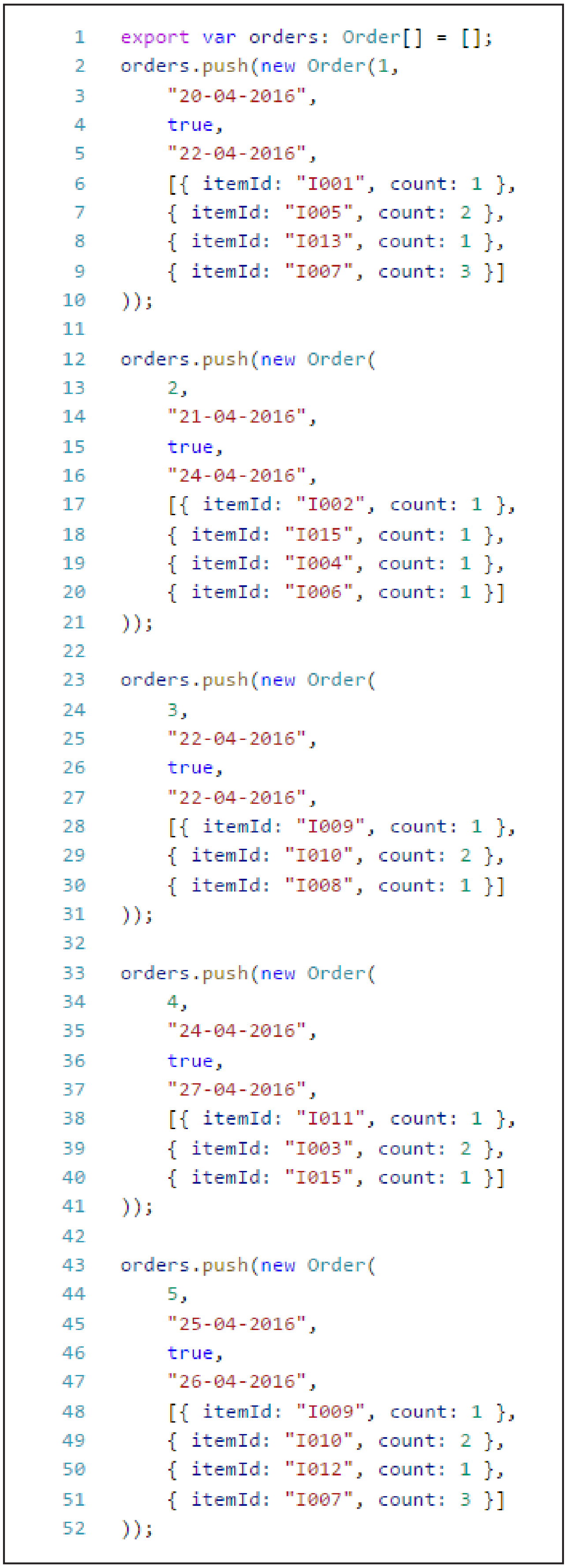
Listing 7.2: *Arrays for items*



```
export var items: Item[] = [];
items.push({ itemId: "I001", name: "Trimmer", unitPrice: 1700, category: "Cosmetics" });
items.push({ itemId: "I002", name: "Chili powder", unitPrice: 50, category: "Food" });
items.push({ itemId: "I003", name: "Head phone", unitPrice: 2000, category: "Electronics" });
items.push({ itemId: "I004", name: "Coffee Maker", unitPrice: 500, category: "Kitchen" });
items.push({ itemId: "I005", name: "T-Shirt", unitPrice: 1000, category: "Clothing" });
items.push({ itemId: "I006", name: "Shampoo", unitPrice: 200, category: "Cosmetics" });
items.push({ itemId: "I007", name: "Mango", unitPrice: 60, category: "Food" });
items.push({ itemId: "I008", name: "Bluetooth Mouse", unitPrice: 2500, category: "Electronics" });
items.push({ itemId: "I009", name: "Vegetable Chopper", unitPrice: 500, category: "Kitchen" });
items.push({ itemId: "I010", name: "Leather Jacket", unitPrice: 10000, category: "Clothing" });
items.push({ itemId: "I011", name: "Tooth Brush", unitPrice: 100, category: "Cosmetics" });
items.push({ itemId: "I012", name: "Tea Dust", unitPrice: 100, category: "Food" });
items.push({ itemId: "I013", name: "Selfie Stick", unitPrice: 5000, category: "Electronics" });
items.push({ itemId: "I014", name: "Mixer Grinder", unitPrice: 4500, category: "Kitchen" });
items.push({ itemId: "I015", name: "Formal Shirt", unitPrice: 1500, category: "Clothing" });
```

The orders array will be added to the file *order.model.ts*. The following listing shows the data in this array:

Listing 7.3: *Array for orders*



```
export var orders: Order[] = [];
orders.push(new Order(1,
    "20-04-2016",
    true,
    "22-04-2016",
    [{ itemId: "I001", count: 1 },
    { itemId: "I005", count: 2 },
    { itemId: "I013", count: 1 },
    { itemId: "I007", count: 3 }]
));

orders.push(new Order(
    2,
    "21-04-2016",
    true,
    "24-04-2016",
    [{ itemId: "I002", count: 1 },
    { itemId: "I015", count: 1 },
    { itemId: "I004", count: 1 },
    { itemId: "I006", count: 1 }]
));

orders.push(new Order(
    3,
    "22-04-2016",
    true,
    "22-04-2016",
    [{ itemId: "I009", count: 1 },
    { itemId: "I010", count: 2 },
    { itemId: "I008", count: 1 }]
));

orders.push(new Order(
    4,
    "24-04-2016",
    true,
    "27-04-2016",
    [{ itemId: "I011", count: 1 },
    { itemId: "I003", count: 2 },
    { itemId: "I015", count: 1 }]
));

orders.push(new Order(
    5,
    "25-04-2016",
    true,
    "26-04-2016",
    [{ itemId: "I009", count: 1 },
    { itemId: "I010", count: 2 },
    { itemId: "I012", count: 1 },
    { itemId: "I007", count: 3 }]
));
```

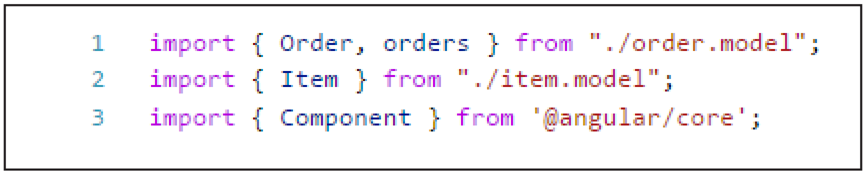
Building the Orders Component

The Orders component consists of two parts.

The first part is a table displaying the list of orders. Each row shows fields of the *Order* class. The component must import the exported members of the file *orders.data.ts*.

Add a new file to the *app* folder and name it *orders.component.ts*. First, import the required set of classes, interfaces, annotations and the arrays.

Listing 7.4: *Import statements*



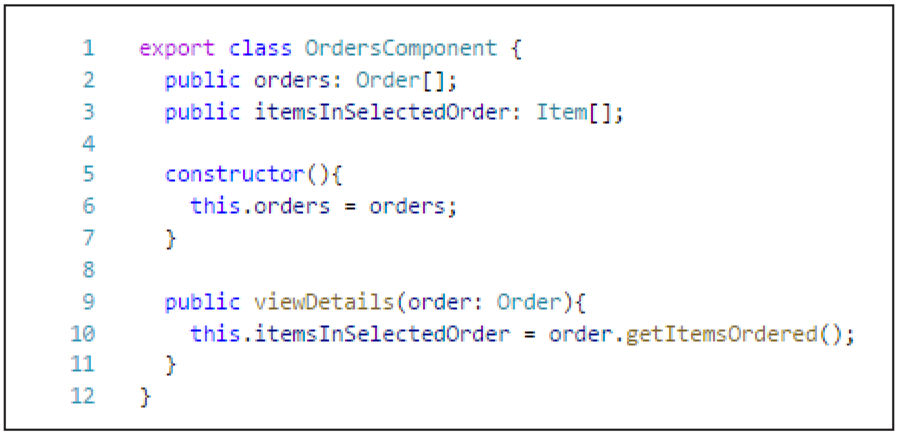
```
import { Order, orders } from "./order.model";
import { Item } from "./item.model";
import { Component } from '@angular/core';
```

The component class will consist of the following public fields and method:

- *orders***:** An array storing the list of products imported from the *orders.data.ts* file. This field will be used to show the list of orders in the view
- *itemsInSelectedOrder***:** An array to store the list of items of a given order
- *viewDetails***:** a method that takes an order and gets details of the items in the order

The component class has a constructor, which assigns the list of orders imported to the *orders* field in the class. Listing 7.5 shows the OrdersComponent class:

Listing 7.5: *OrdersComponent class*

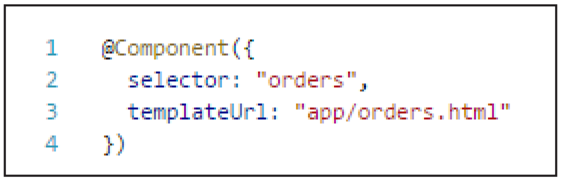


```
export class OrdersComponent {
  public orders: Order[];
  public itemsInSelectedOrder: Item[];

  constructor(){
    this.orders = orders;
  }

  public viewDetails(order: Order){
    this.itemsInSelectedOrder = order.getItemsOrdered();
  }
}
```

> **NOTE:** This is not the best way to consume data in a component. The above code can be made better using Services. Services will be discussed in Chapter 27.

At this point, it is just an exported class. Let's turn this to a component by adding the *Component* decorator to it. The component has to be set with a selector and a template. As this will be a larger template consisting of two tables (one for the master list of orders, and another to list the set of items) it would be best to put the template in a separate HTML file. Otherwise, the bulky inline HTML in the TypeScript file will make it an unmaintainable mess. The file can be referred in the component using the *templateUrl* property. Listing 7.6 shows the decorator that has to be added to the above class:

Listing 7.6: *Annotation of OrdersComponent*



```
@Component({
  selector: "orders",
  templateUrl: "app/orders.html"
})
```

Building the Template

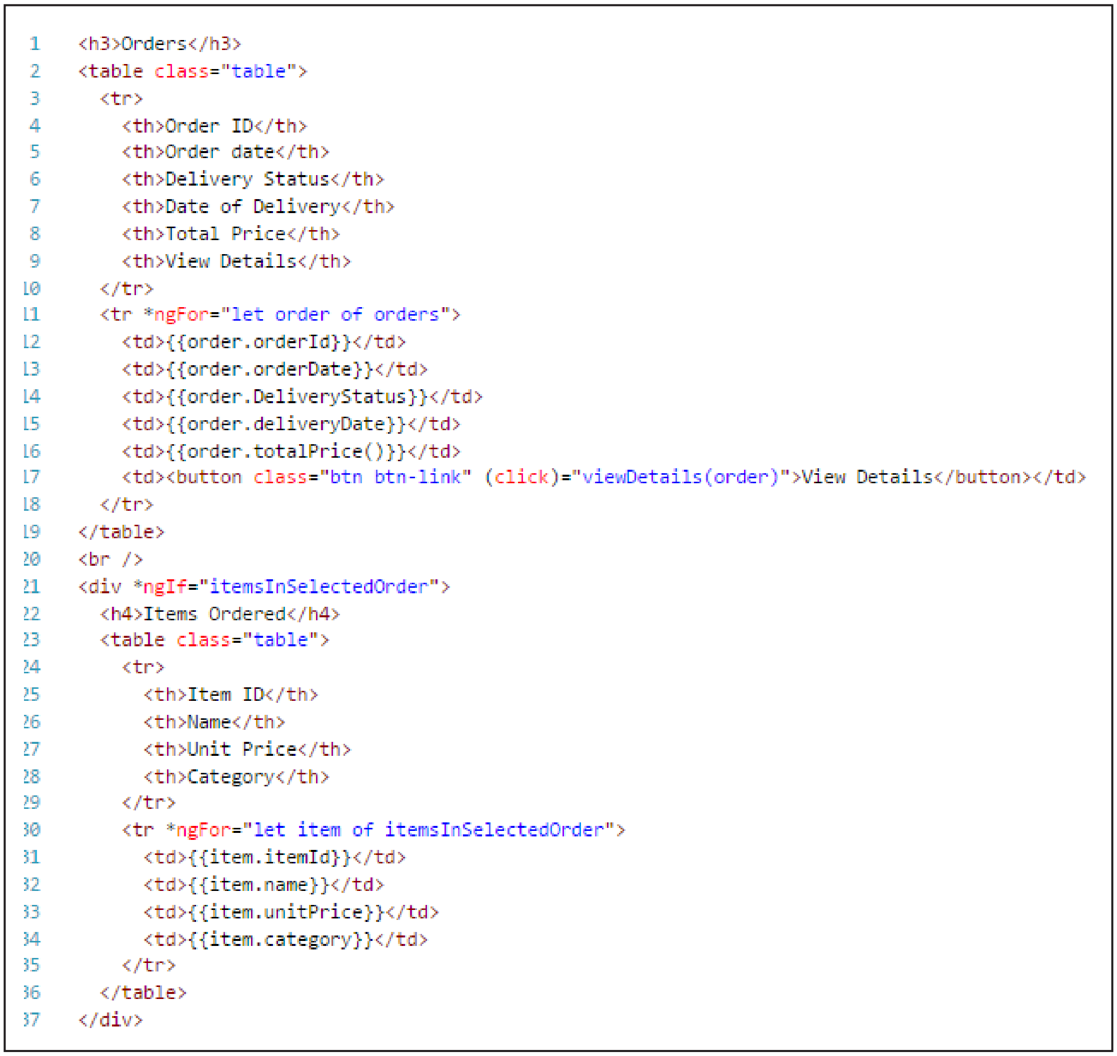
To reiterate, the template will have two tables displaying orders and items in an order. The order table runs a loop over the *orders* array in the component and displays its fields. It uses the following features of Angular 2:

- **Interpolation:** Interpolation is used to bind data from view model on the view. In Angular 2, the view model for any view is an instance of its component class. An interpolation in Angular 2 can bind a field of a component, a property of a field of a component, or even perform some simple JavaScript operations like arithmetic operations. The following is an example of using an interpolation displaying a name inside an SPAn element:
  `<span>{{name}}</span>`
- **ngFor directive:** ngFor is a directive built inside Angular 2 to support binding collections. The syntax of using *ngFor* is shown in listing 7.6:
- **`<div *ngFor="#item of items">`**
- **`{{item.name}}`**
- **`</div>`**

- The asterisk in front of the directive indicates it is a structural directive, which will manipulate the HTML template. The hash symbol in front of the record variable *item* indicates it is a local variable, and can be used in the template inside *ngFor*.
- **ngIf directive:** It is another structural directive used to add or remove a block of HTML in a view, based on the value of a condition.
- `<div *ngIf=”showTheBlock”>Bingo! I am visible!!!</div>`
- The div shown in Listing 7.8 is displayed only when the value of the field *showTheBlock* is set to a value that evaluates to *true*.
- **Event binding:** Angular 2 doesn’t define new directives to bind events in the template, thus any event can be bound to a method in the component using a special syntax. Listing 7.9 shows an example of attaching a method to click event of a button:
- `<button (click)=”doSomething()”>Click Here</button>`
- The parentheses around click in the snippet indicate that it is an event.

These features will be discussed with examples in the subsequent chapters. Listing 7.7 shows template of the component:

Listing 7.7: *Template of OrdersComponent*

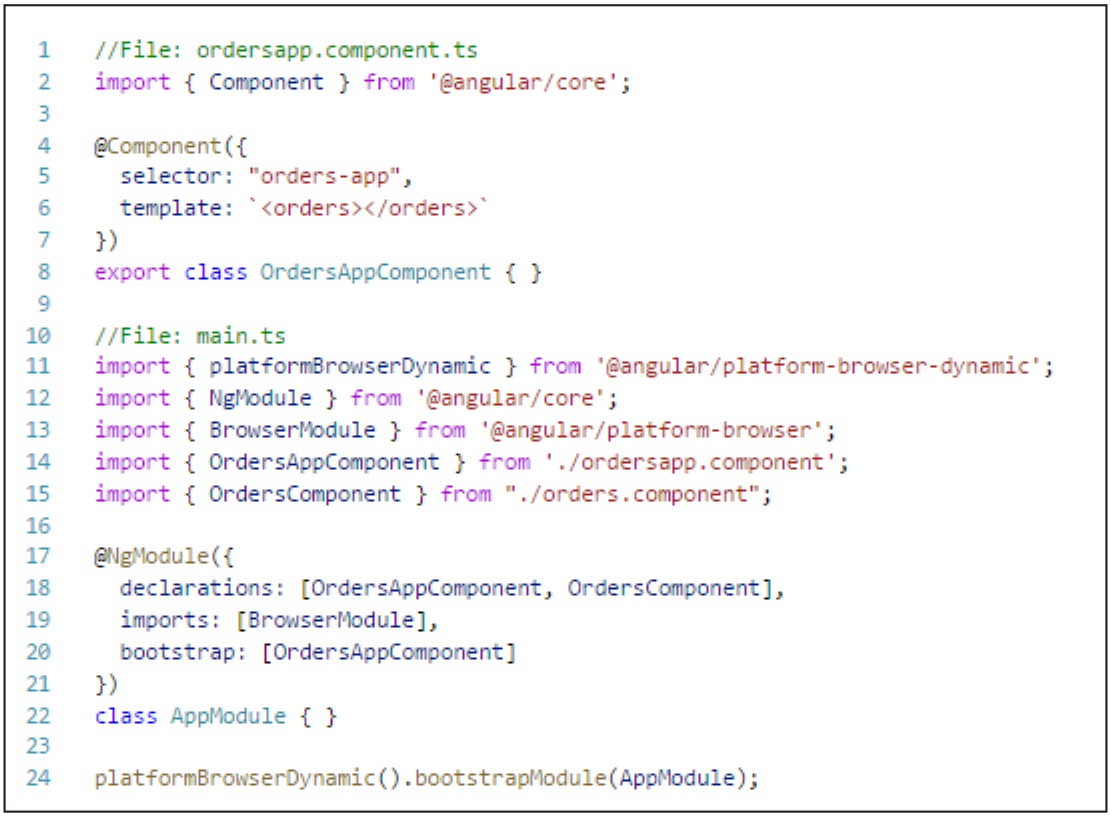


```
<h3>Orders</h3>
<table class="table">
  <tr>
    <th>Order ID</th>
    <th>Order date</th>
    <th>Delivery Status</th>
    <th>Date of Delivery</th>
    <th>Total Price</th>
    <th>View Details</th>
  </tr>
  <tr *ngFor="let order of orders">
    <td>{{order.orderId}}</td>
    <td>{{order.orderDate}}</td>
    <td>{{order.DeliveryStatus}}</td>
    <td>{{order.deliveryDate}}</td>
    <td>{{order.totalPrice()}}</td>
    <td><button class="btn btn-link" (click)="viewDetails(order)">View Details</button></td>
  </tr>
</table>
<br />
<div *ngIf="itemsInSelectedOrder">
  <h4>Items Ordered</h4>
  <table class="table">
    <tr>
      <th>Item ID</th>
      <th>Name</th>
      <th>Unit Price</th>
      <th>Category</th>
    </tr>
    <tr *ngFor="let item of itemsInSelectedOrder">
      <td>{{item.itemId}}</td>
      <td>{{item.name}}</td>
      <td>{{item.unitPrice}}</td>
      <td>{{item.category}}</td>
    </tr>
  </table>
</div>
```

The component shows the list of orders in a table and (upon clicking the View Details button on any row) the list of items in the second table. The second table is not displayed unless the field *itemsInSelectedOrder* is populated with data.

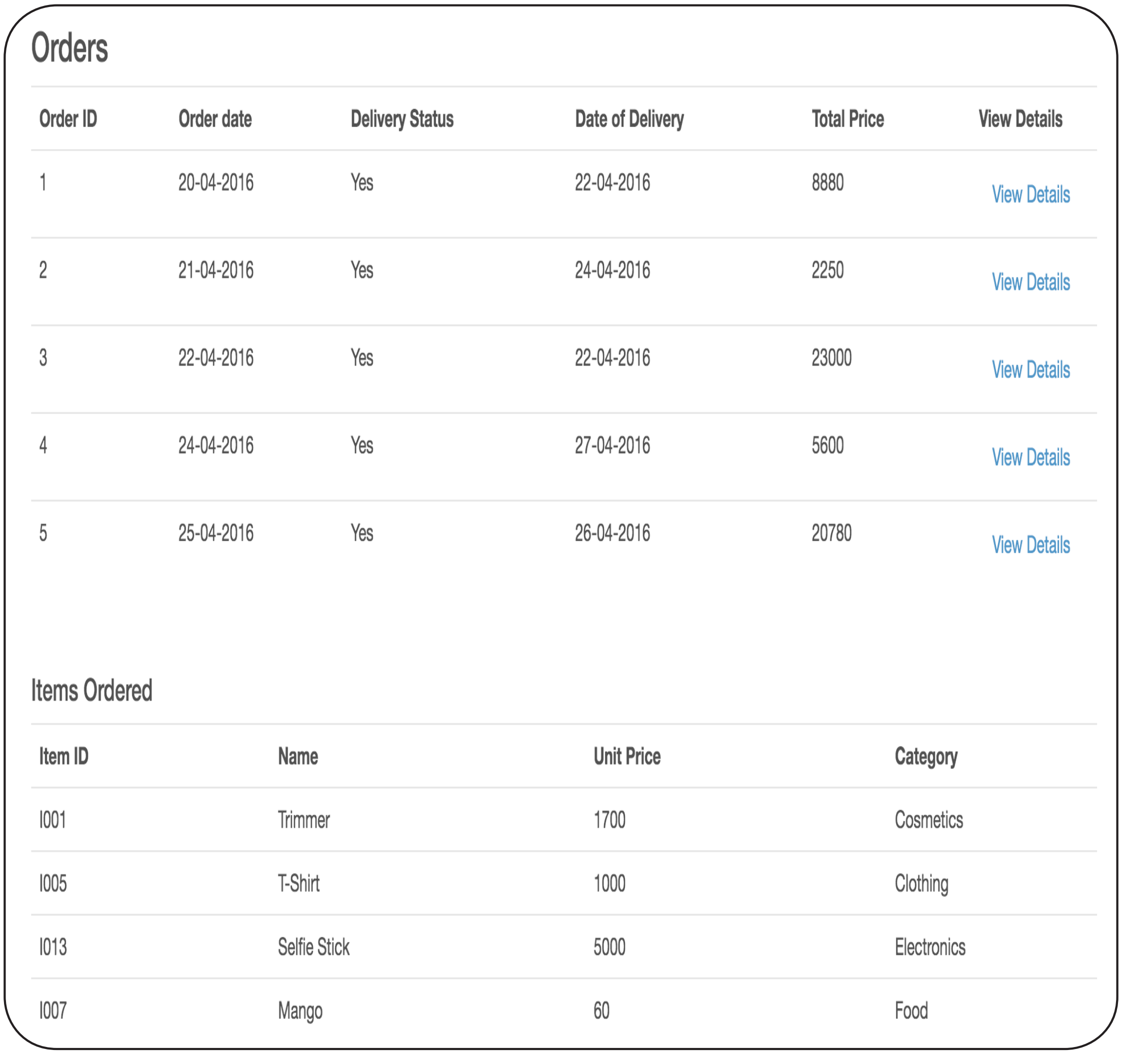
The final step is to use this component in the main application component and bootstrap the application using a module. The main component, *OrdersAppComponent* doesn't contain any logic; it is used to load the component *OrdersComponent*. Both *OrdersComponent* and *OrdersAppComponent* are registered in the module. The *OrdersAppComponent* is used. Listing 7.8 shows this:

Listing 7.8: *Contents of OrdersAppComponent and main.ts*



```
//File: ordersapp.component.ts
import { Component } from '@angular/core';

@Component({
  selector: "orders-app",
  template: `<orders></orders>`
})
export class OrdersAppComponent { }

//File: main.ts
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { OrdersAppComponent } from './ordersapp.component';
import { OrdersComponent } from "./orders.component";

@NgModule({
  declarations: [OrdersAppComponent, OrdersComponent],
  imports: [BrowserModule],
  bootstrap: [OrdersAppComponent]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Save all of the files, run the application, and open the application in a browser. The display will be similar to the following:



Orders

Order ID	Order date	Delivery Status	Date of Delivery	Total Price	View Details
1	20-04-2016	Yes	22-04-2016	8880	View Details
2	21-04-2016	Yes	24-04-2016	2250	View Details
3	22-04-2016	Yes	22-04-2016	23000	View Details
4	24-04-2016	Yes	27-04-2016	5600	View Details
5	25-04-2016	Yes	26-04-2016	20780	View Details

Items Ordered

Item ID	Name	Unit Price	Category
I001	Trimmer	1700	Cosmetics
I005	T-Shirt	1000	Clothing
I013	Selfie Stick	5000	Electronics
I007	Mango	60	Food

Figure 7.1: *Screenshot of the application in the browser*

Conclusion

As this chapter demonstrates, some of the built-in features of Angular 2 can be used with custom components to build data-bound interactive views. These features make programmers more productive and also improve usability of applications. These features will be used extensively in the upcoming chapters.

Chapter 8

Multiple Components Communication

COMPONENTS ARE BUILDING BLOCKS of a Angular 2 applications and can be used to define Domain Specific Language (DSL). Meaning, you can create components to fulfill the business needs of your application. Every significant, logical module of an application can also be viewed as a combination of a set of related components.

Components work together to achieve a single goal.

These components need to communicate with each other to exchange information. To do so, they need to have input and output points, which helps keep the components lighter and smaller, increases readability, and eases debugging of the application.

Chapter 7 demonstrated a component with a list of items and their details. The component was large, as the logic of displaying both the list and the details was stuffed into a single component. A better approach is to split it into two components: one to display the list of items, and another to display details of the selected component. The list component will pass the ID of the item to be displayed to the *details* component. The order being currently displayed in the *details* component will be highlighted in the list. On closing the *details* component, it will inform the *list* component so that the row won't be highlighted anymore.

The sample of this chapter will be built using the development environment described in Chapter 5. The following software and packages must be installed in the development environment:

- **Node.js:** To start a web server to run the application and for the Node.js Package Manager (npm)
- **TypeScript compiler:** To compile TypeScript
- **TypeScript typings installer:** To install TypeScript typings of the libraries used in the application
- **Angular 2 and its dependencies:** The libraries required for the project; to be installed in the project using npm

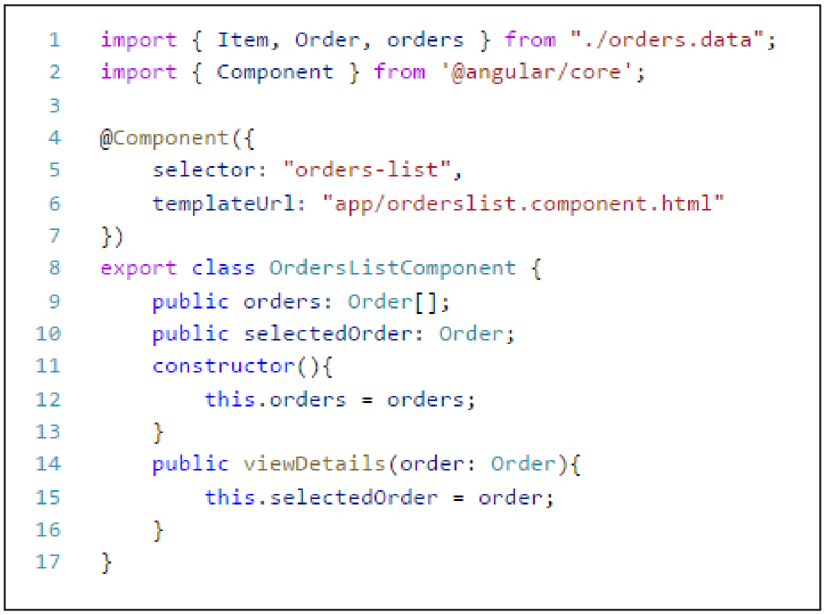
- **SystemJS:** To load modules; o be installed in the project using npm
- **Koa.js:** A Node.js framework to start a Node.js server

Create a new folder named MasterDetail-Interactive and copy the contents of the setup project into it. Run the npm packages and make sure the sample runs.

Creating an Orders List and Order Details Components

Create two components: one to display the list of orders and one to display the details of a particular order. The orders list component will use the details component as a child component in it. Add a new file in the *app* folder and name it *orderslist.component.ts.* Add the code shown in listing 8.1 to it:

Listing 8.1: *Code of OrdersListComponent*

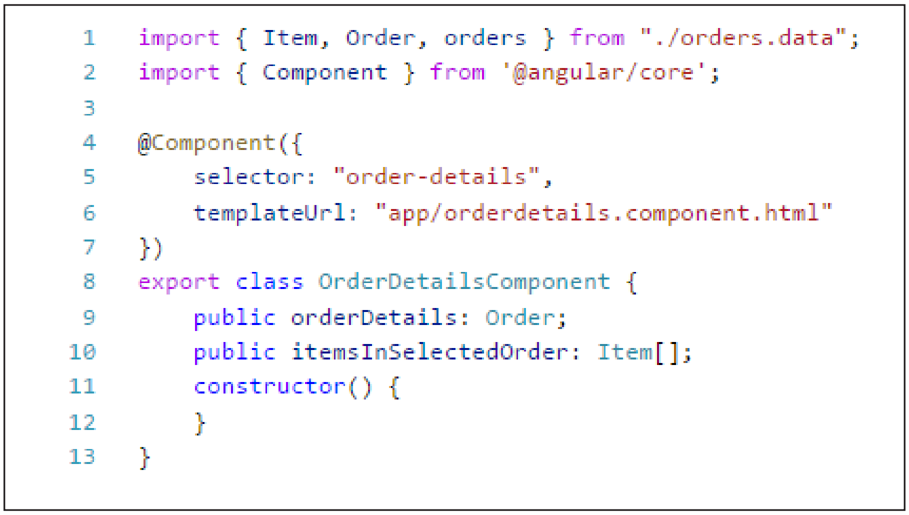


```
import { Item, Order, orders } from "./orders.data";
import { Component } from '@angular/core';

@Component({
    selector: "orders-list",
    templateUrl: "app/orderslist.component.html"
})
export class OrdersListComponent {
    public orders: Order[];
    public selectedOrder: Order;
    constructor(){
        this.orders = orders;
    }
    public viewDetails(order: Order){
        this.selectedOrder = order;
    }
}
```

The *orders.data.ts* file imported in Listing 8.1 is the file containing the model class for Order, an interface representing the structure of an item, and a static list of Orders with some items added to each of the Order. This data will be displayed in the view. You may find this file in the source code of this chapter or in Chapter 7 .

The orders list component refers to a different file for the HTML template. To create this file, add a new file in the *app* folder, and name it *orderslist.component.html.* Add the following HTML markup to this file:

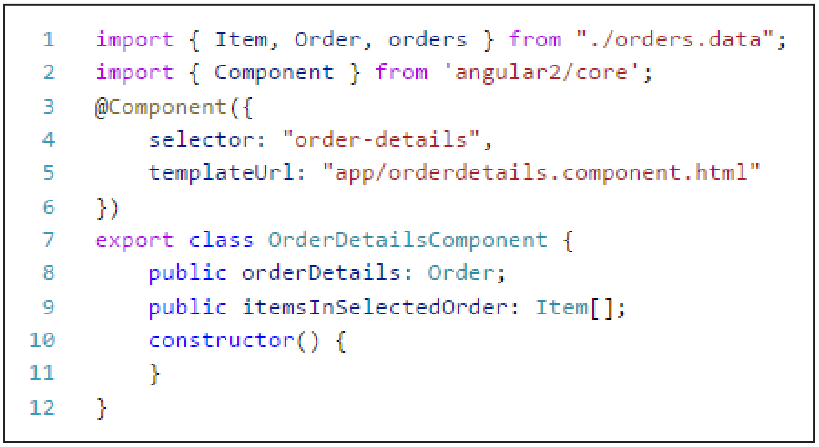
Listing 8.2: *Template of OrdersListComponent in orderlist.component.html*



```
import { Item, Order, orders } from "./orders.data";
import { Component } from '@angular/core';

@Component({
    selector: "order-details",
    templateUrl: "app/orderdetails.component.html"
})
export class OrderDetailsComponent {
    public orderDetails: Order;
    public itemsInSelectedOrder: Item[];
    constructor() {
    }
}
```

This component loads and uses the order details component, but the details component is not defined yet. Let's define it now. Add a new file in the *app* folder and name it *orderdetails.component.ts.* Add the following code to this file:

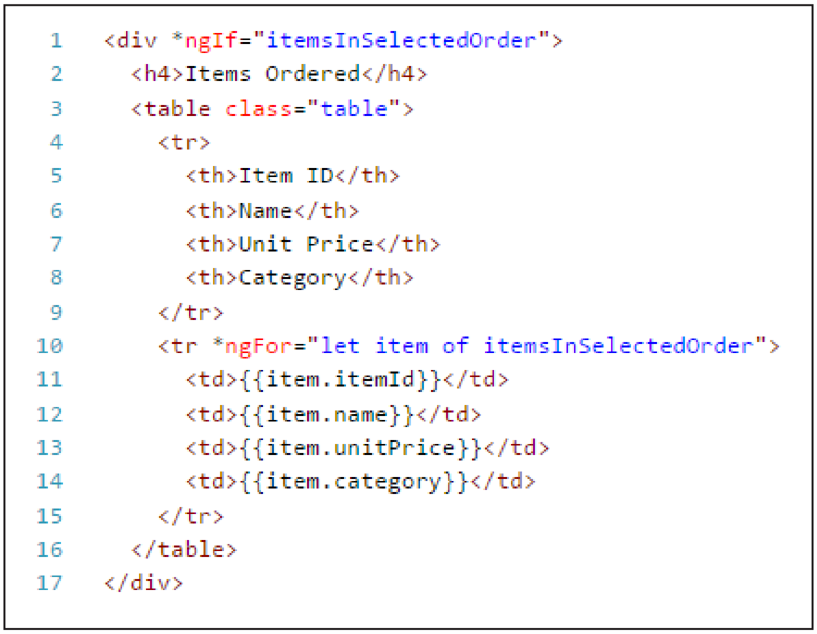
Listing 8.3: *Code of OrderDetailsComponent*



```
import { Item, Order, orders } from "./orders.data";
import { Component } from 'angular2/core';
@Component({
    selector: "order-details",
    templateUrl: "app/orderdetails.component.html"
})
export class OrderDetailsComponent {
    public orderDetails: Order;
    public itemsInSelectedOrder: Item[];
    constructor() {
    }
}
```

The HTML template of the order details component will only display the list of items in a table. Add a new file to the *app* folder and name it *orderdetails.component.html.* Add the following code to this file:

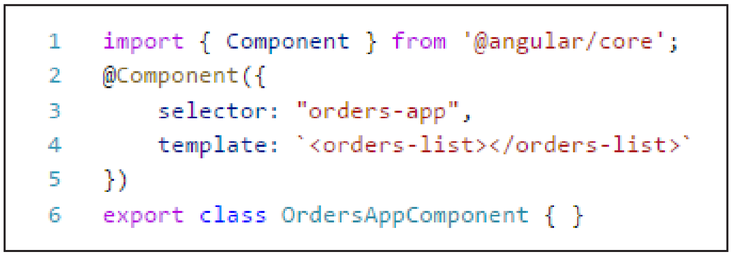
Listing 8.4: *Template of OrderDetailsComponent*



```
<div *ngIf="itemsInSelectedOrder">
  <h4>Items Ordered</h4>
  <table class="table">
    <tr>
      <th>Item ID</th>
      <th>Name</th>
      <th>Unit Price</th>
      <th>Category</th>
    </tr>
    <tr *ngFor="let item of itemsInSelectedOrder">
      <td>{{item.itemId}}</td>
      <td>{{item.name}}</td>
      <td>{{item.unitPrice}}</td>
      <td>{{item.category}}</td>
    </tr>
  </table>
</div>
```

Now that the *OrdersListComponent* is ready, let's seen how it appears on the page. Add a new file to the application to store the main application module and name it *ordersapp.component.ts*. This component doesn't contain any logic; it is used to load the component *OrdersListComponent* on the page. The following listing shows the code of this file:

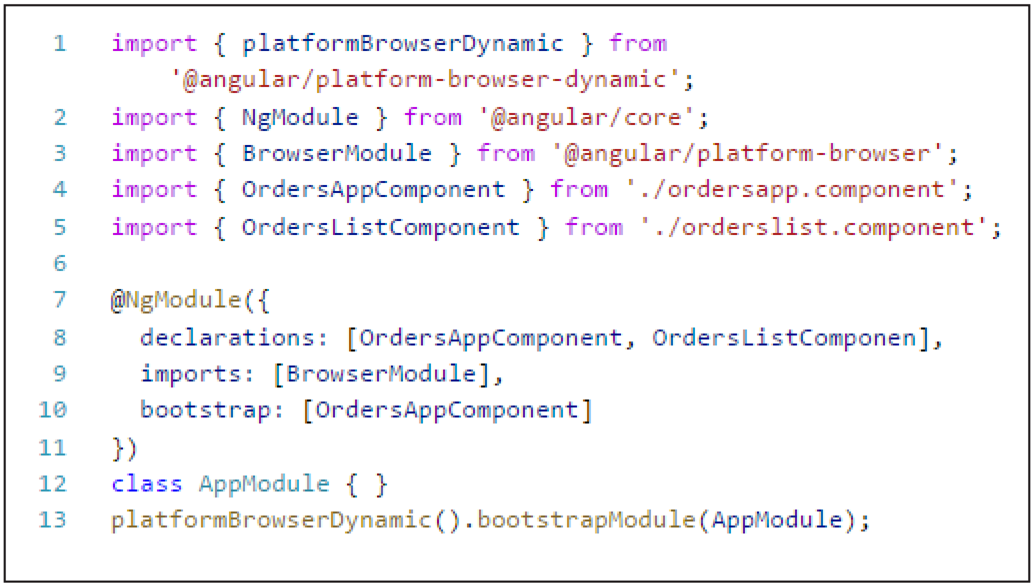
Listing 8.5: *Code of OrdersAppComponent*



```
import { Component } from '@angular/core';
@Component({
    selector: "orders-app",
    template: `<orders-list></orders-list>`
})
export class OrdersAppComponent { }
```

Both of the components created must be registered in a module by opening the file main.ts and modifying the code of the file as in the following listing:

Listing 8.6: *Code of module*

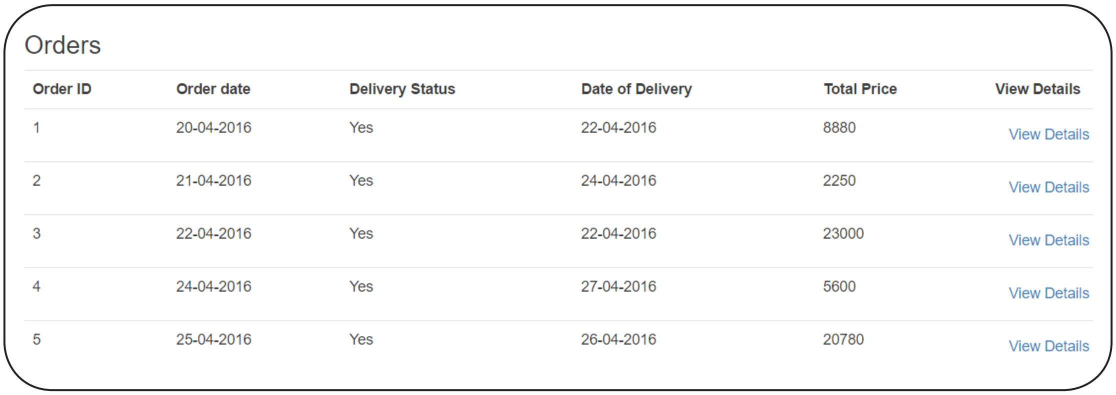


```
import { platformBrowserDynamic } from
    '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { OrdersAppComponent } from './ordersapp.component';
import { OrdersListComponent } from './orderslist.component';

@NgModule({
  declarations: [OrdersAppComponent, OrdersListComponen],
  imports: [BrowserModule],
  bootstrap: [OrdersAppComponent]
})
class AppModule { }
platformBrowserDynamic().bootstrapModule(AppModule);
```

Save all the files and run the application using the following command:

```
> npm run start
```

Open a browser and change the URL to *http://localhost:3000*. Running the page displays the list of orders. The button "View Details" doesn't perform an action, as the action hasn't been defined yet. The action will be added in the next section.



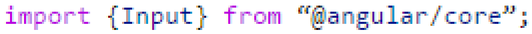
Orders

Order ID	Order date	Delivery Status	Date of Delivery	Total Price	View Details
1	20-04-2016	Yes	22-04-2016	8880	View Details
2	21-04-2016	Yes	24-04-2016	2250	View Details
3	22-04-2016	Yes	22-04-2016	23000	View Details
4	24-04-2016	Yes	27-04-2016	5600	View Details
5	25-04-2016	Yes	26-04-2016	20780	View Details

Figure 8.1: *The orders table*

Displaying Items in an Order

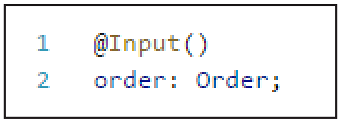
When a user clicks the "View Details" button on a row of the orders list, the selected order has to be passed to the order details component, so a field must be added in the *OrderDetailsComponent* class to accept the input. To do so, a field in the component class must be marked with the decorator *Input* on it. The decorator is defined in the *@angular/core* package. Listing 8.5 shows how to import it:



```
import {Input} from “@angular/core”;
```

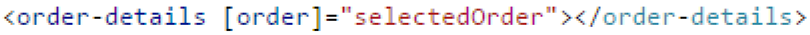
The *OrderDetailsComponent* will accept an order object from the *OrdersListComponent*. Add the field shown in listing 8.7 to the *OrderDetailsComponent* class.

Listing 8.7: *Input property to accept order*



```
@Input()
order: Order;
```

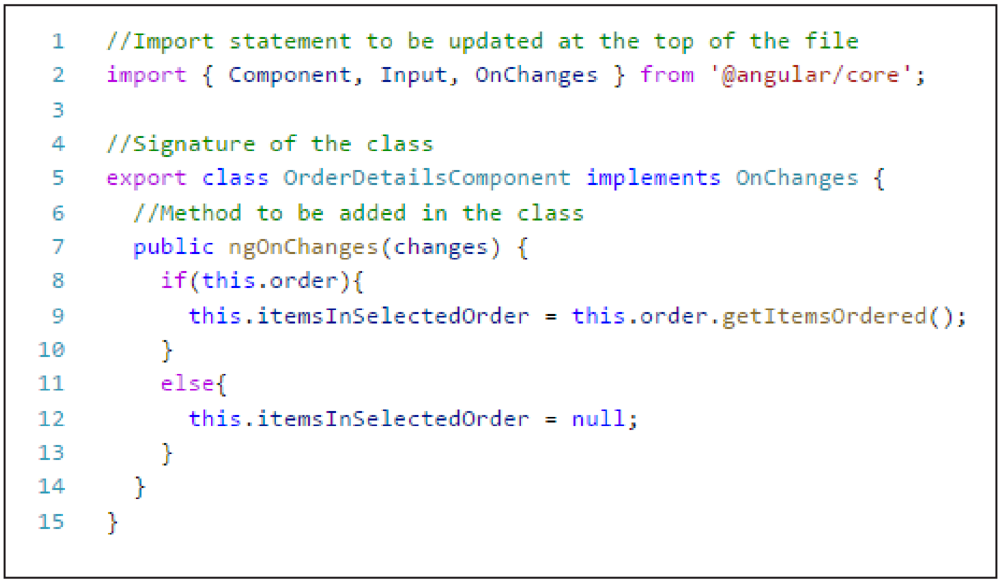
This field has to be property-bound on the component’s element used in the list component. Change the *order-details* tag in the *orderslist.component.html* file to have this property. The following snippet shows the modified tag:



```
<order-details [order]="selectedOrder"></order-details>
```

As the field *order* is property-bound, its value will change whenever the field *selectedOrder* gets a new value in the orders list component. Whenever the value of the *order* changes, the *items* added to the *order* will be retrieved using the *getItemsOrdered* method on the order object. This can be implemented using a lifecycle method of the component. The method *ngOnChanges* is called by the change detector whenever an input field of the component is modified. This method can be added to any component to listen to the changes which are happening to the input fields of the component. Angular 2 has an interface, *OnChanges*, to make the implementation type safe. The method can be implemented in the class without extending the component class from the interface as well, but it is better to implement the interface as it provides type safety for the method. Listing 8.8 demonstrates this method, which has to be added to the *OrderDetailsComponent* class:

Listing 8.8: *ngOnChange life cycle hook*



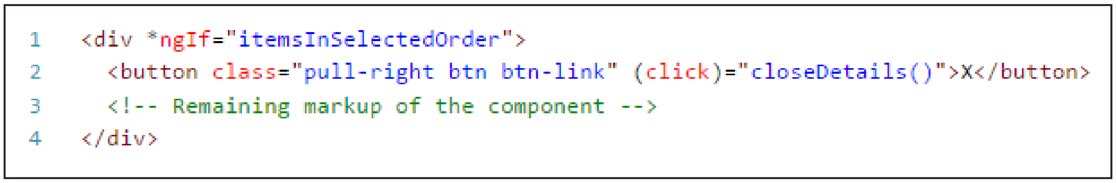
```
//Import statement to be updated at the top of the file
import { Component, Input, OnChanges } from '@angular/core';

//Signature of the class
export class OrderDetailsComponent implements OnChanges {
  //Method to be added in the class
  public ngOnChanges(changes) {
    if(this.order){
      this.itemsInSelectedOrder = this.order.getItemsOrdered();
    }
    else{
      this.itemsInSelectedOrder = null;
    }
  }
}
```

Save all the files and run the application. Click the View Details button on the orders list, randomly, to see the items of the order selected being displayed in the items list.

Closing the Items List

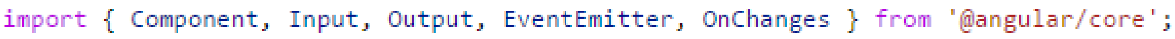
The user should be provided with an option to close the items list. Add a button in the *orderdetails.component.html* to float on the top right corner of the section. It has to be added as the first child under the root *div* element of the template. Listing 8.9 shows the markup of the button:

Listing 8.9: *ngIf to show or hide details of an item*



```
<div *ngIf="itemsInSelectedOrder">
  <button class="pull-right btn btn-link" (click)="closeDetails()">X</button>
  <!-- Remaining markup of the component -->
</div>
```

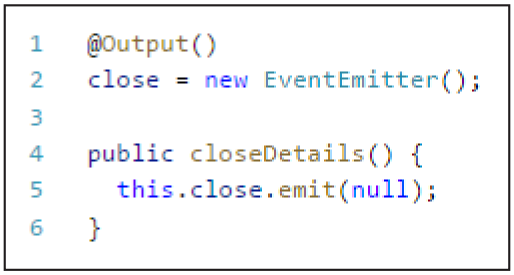
Next, the *closeDetails* method must be defined in the *OrderDetailsComponent* class to send a message to the orders list component, and the *OrdersListComponent* must set the *selectedOrder* to *null* when it receives this message. To achieve this, the *OrderDetailsComponent* has to expose an event and it should be handled in the *OrdersListComponent*. Two more objects (*Output* and *EventEmitter)* from the *@angular/core* package have to be imported for this. The import statement in the file *orderdetails.component.ts* changes to:



```
import { Component, Input, Output, EventEmitter, OnChanges } from '@angular/core';
```

An output property has to be defined in the *OrderDetailsComponent* class to emit the event in the *closeDetaills* method. Listing 8.10 defines the property and the method:

Listing 8.10: *Code for close event*

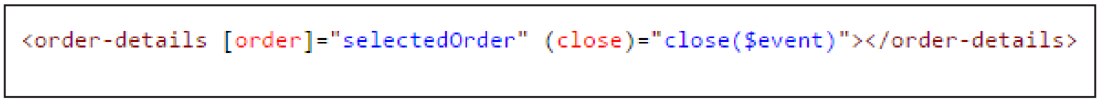


```
@Output()
close = new EventEmitter();

public closeDetails() {
  this.close.emit(null);
}
```

The call to the *emit* method on the *close* event sends a notification to the consumer of the event. Data can be passed to the consumer through the *emit* method. The *closeDetails* method didn't have to pass any data, so it emits the event with *null* value.

The property *close* is equivalent to any event on the HTML element. An event handler can be bound to it on the component's HTML tag. In the *orderslist.component.html* file, change the *order-details* tag as shown in Listing 8.11:

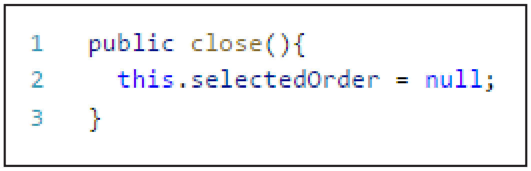
Listing 8.11: *order-details element with close event*



```
<order-details [order]="selectedOrder" (close)="close($event)"></order-details>
```

Implementing the *close* method is the only pending thing to complete this functionality. Add the following method to the *OrdersListComponent* class.

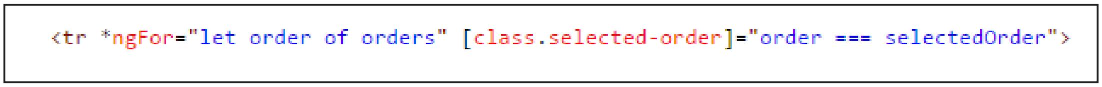
Listing 8.12: *Implementation of close event handler*



```
public close(){
  this.selectedOrder = null;
}
```

Save all the files and run the sample. Now the items section should close upon clicking the cross button. Though the functionality works, the user doesn't yet know which order is selected out of the list. Highlighting the selected row should help here. Change the row element in the *orderslist.component.html* to the following:

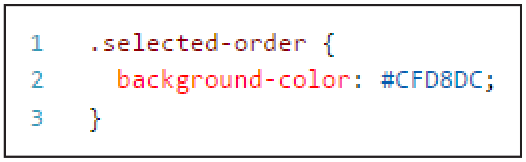
Listing 8.13: *Highlighting selected row*



```
<tr *ngFor="let order of orders" [class.selected-order]="order === selectedOrder">
```

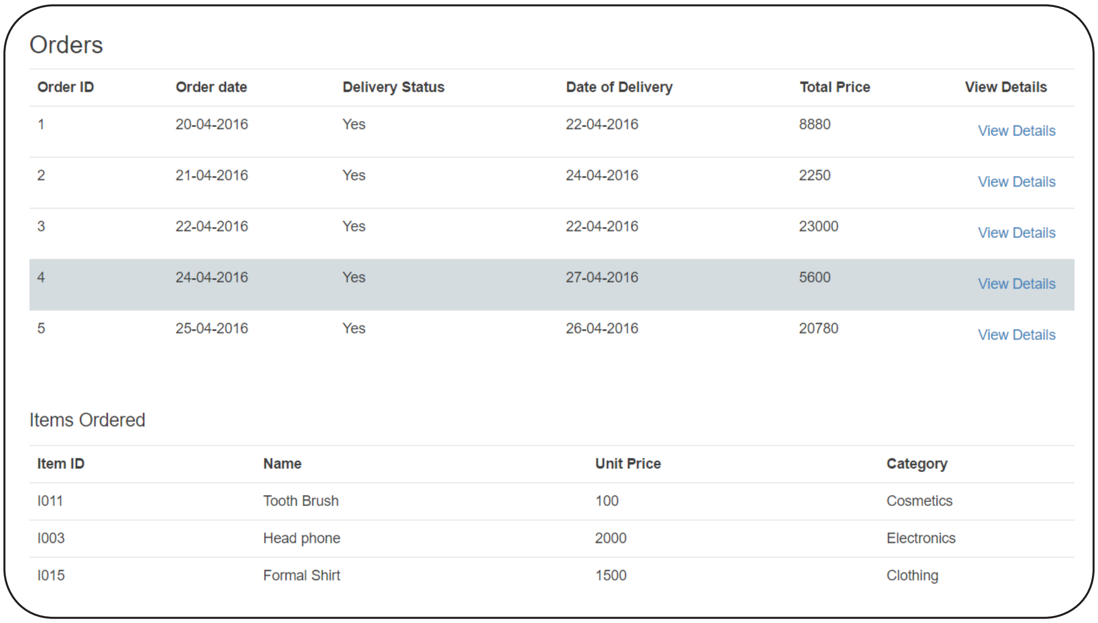
The CSS class *selected-order* will be applied if the order displayed in the row is selected. As it is property- bound, the class is removed from this row as soon as the value of *selectedOrder* is changed. The following is the CSS class, add it to a style tag in the file *index.html*:

Listing 8.14: *Style of selected row*



```
.selected-order {
  background-color: #CFD8DC;
}
```

Now the user will get an indication of the row selected. The style applied to the row element is removed via data binding when a different order is selected or when the items are closed.



Orders

Order ID	Order date	Delivery Status	Date of Delivery	Total Price	View Details
1	20-04-2016	Yes	22-04-2016	8880	View Details
2	21-04-2016	Yes	24-04-2016	2250	View Details
3	22-04-2016	Yes	22-04-2016	23000	View Details
4	24-04-2016	Yes	27-04-2016	5600	View Details
5	25-04-2016	Yes	26-04-2016	20780	View Details

Items Ordered

Item ID	Name	Unit Price	Category
I011	Tooth Brush	100	Cosmetics
I003	Head phone	2000	Electronics
I015	Formal Shirt	1500	Clothing

Figure 8.2: *Preview of master-detail view*

Conclusion

The components can communicate with each other without being aware of each other's presence. This helps in keeping the components isolated from each other,yet achieves the functionality of the application. The ability to communicate among different pieces also helps in keep the code pieces as *lean* as possible. Some additional features of components will be discussed in Chapter 9.

Chapter 9

Applying Templates and Styles to Components

COMPONENTS REPRESENT INDIVIDUAL PIECES of an Angular 2 application UI. Every component has a view that consists of a piece of HTML markup. The amount of markup varies based on the size and the responsibilities carried out by the component. The size of the template decides where a template has to be stored.

A visual design drives every software application, including Angular 2 applications. To make the application look the way the design looks, the design of every component must match its portion on the page. In CSS, it's hard to scope the styles to the block where they are applied. When the page is rendered, styles written for one portion of the page may step into the styles of other elements and make those elements look odd.

Angular 2's components have a solution to this problem.

Applying Template to a Component

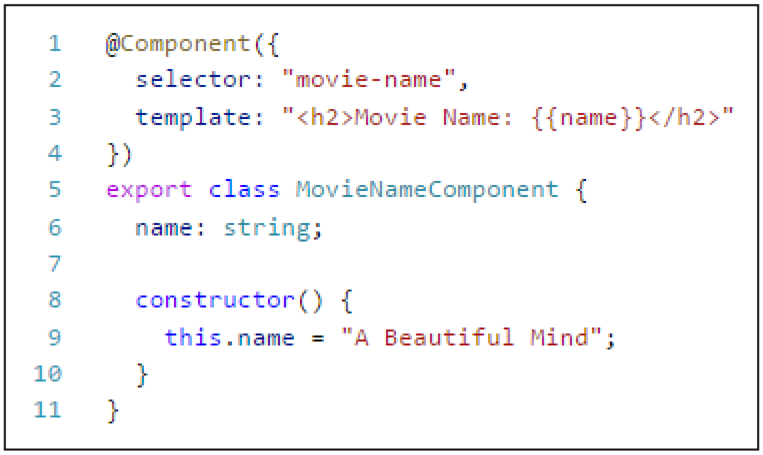
Every component consists of two significant parts: a class, containing a view model of the component, and the HTML template to be rendered in the component. The size of the template could vary from a single statement of HTML to hundreds of lines.

Let's examine a few scenarios to determine how to assign templates in these cases.

Simple Templates

Consider a simple component displaying the name of a movie. The component class will contain the name of the movie and the template will have an *h2* element displaying the name. As the component is not doing much, the template can be assigned inline. The following snippet defines this component:

Listing 9.1: *Code of MovieNameComponent:*

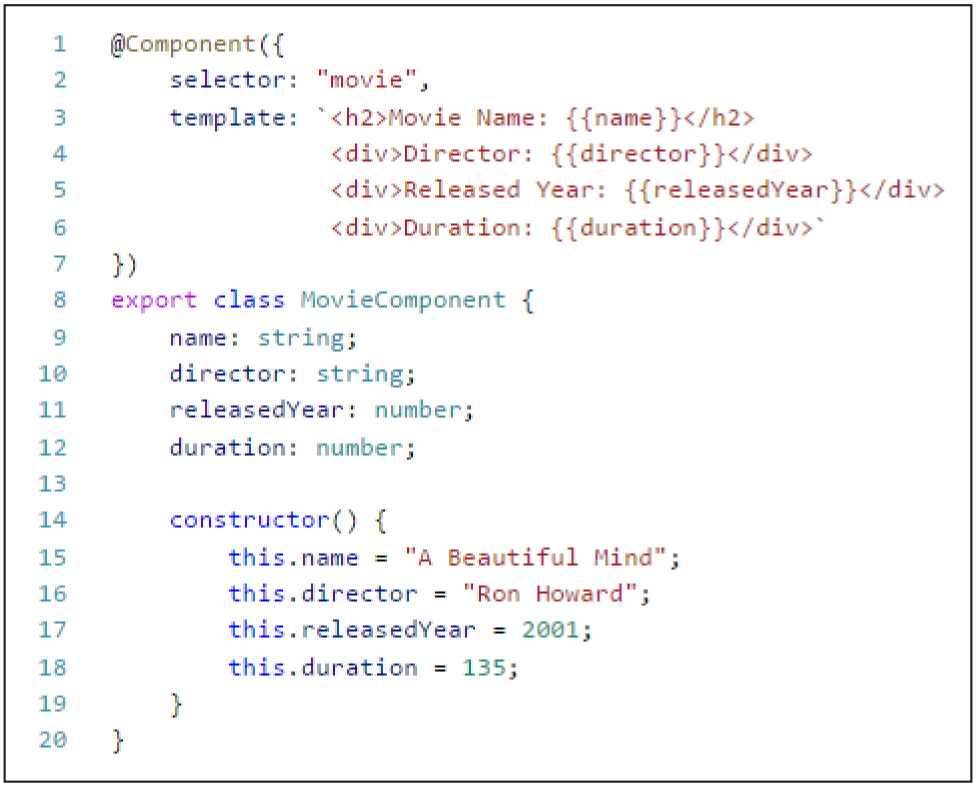


```
@Component({
  selector: "movie-name",
  template: "<h2>Movie Name: {{name}}</h2>"
})
export class MovieNameComponent {
  name: string;

  constructor() {
    this.name = "A Beautiful Mind";
  }
}
```

Multiline Templates

If a component needs to do something other than displaying a single value, it will need a template that takes more than one statement. If, for example, a component has to show additional fields of a movie, although the template of this component will span multiple statements, it is not big enough to be stored in a separate file. This template can be assigned inline in the component. As it will be a multiline string, let's use ES6 string templates to define the template. The following snippet defines the *Movie* component to display movie details:

Listing 9.2: *Code of MovieComponent*

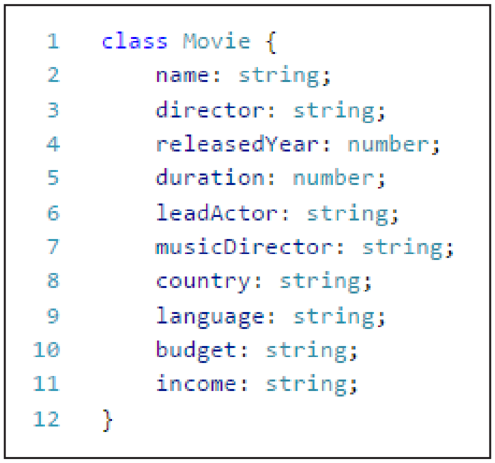


```
@Component({
    selector: "movie",
    template: `<h2>Movie Name: {{name}}</h2>
               <div>Director: {{director}}</div>
               <div>Released Year: {{releasedYear}}</div>
               <div>Duration: {{duration}}</div>`
})
export class MovieComponent {
    name: string;
    director: string;
    releasedYear: number;
    duration: number;

    constructor() {
        this.name = "A Beautiful Mind";
        this.director = "Ron Howard";
        this.releasedYear = 2001;
        this.duration = 135;
    }
}
```

Bigger Components

Let's make this component even bigger by displaying some additional fields of the movie. The model class in Listing 9.3 shows the properties to be displayed in the view:

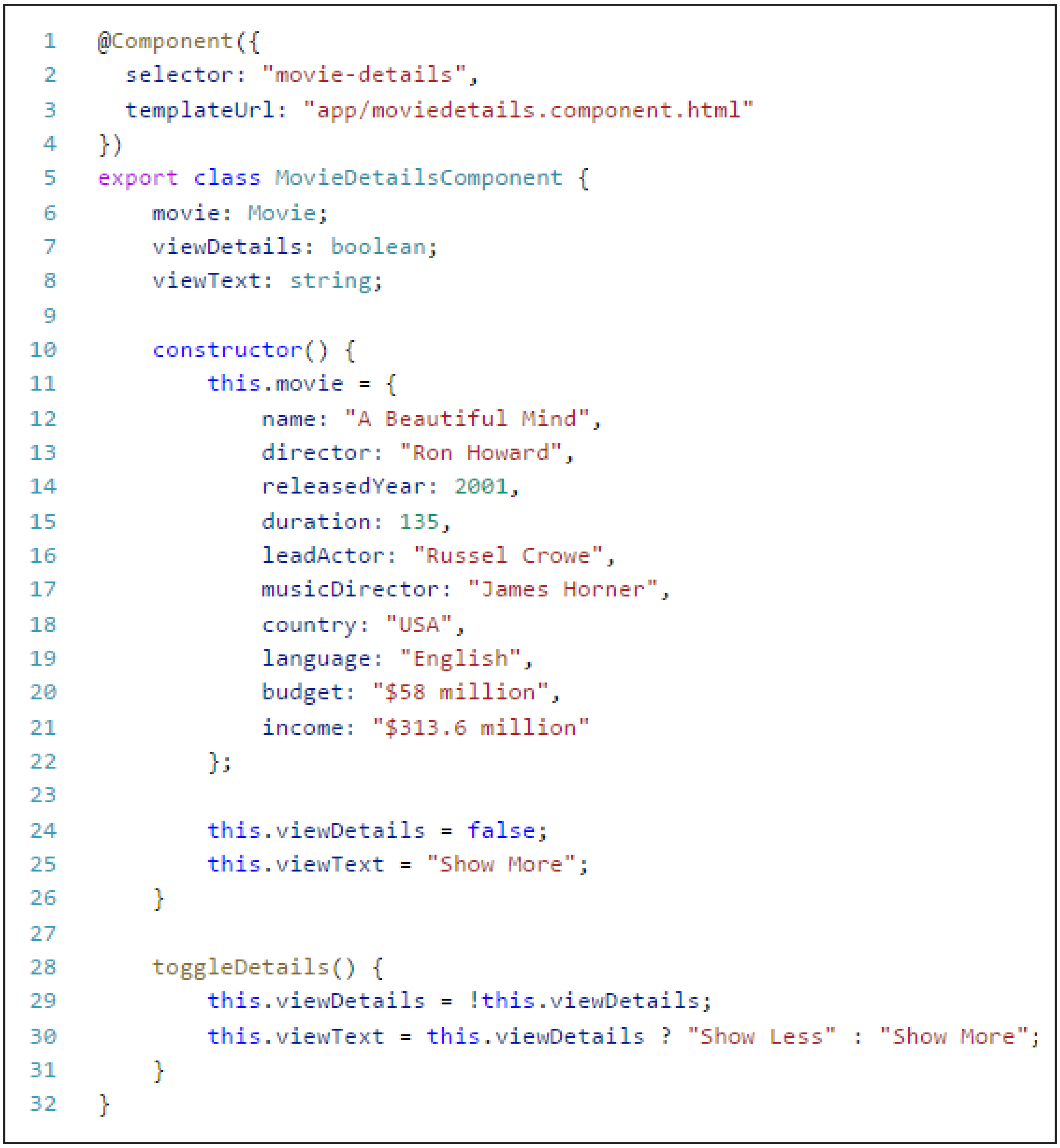
Listing 9.3: *Code of Movie class*



```
class Movie {
    name: string;
    director: string;
    releasedYear: number;
    duration: number;
    leadActor: string;
    musicDirector: string;
    country: string;
    language: string;
    budget: string;
    income: string;
}
```

Though a component displaying these fields in the view will not have a large template, it will still need an HTML template containing more than ten statements. It's better to put this template in a separate HTML file, to easily compose and edit a larger markup when read as HTML by the editors. Let's define a component for this.

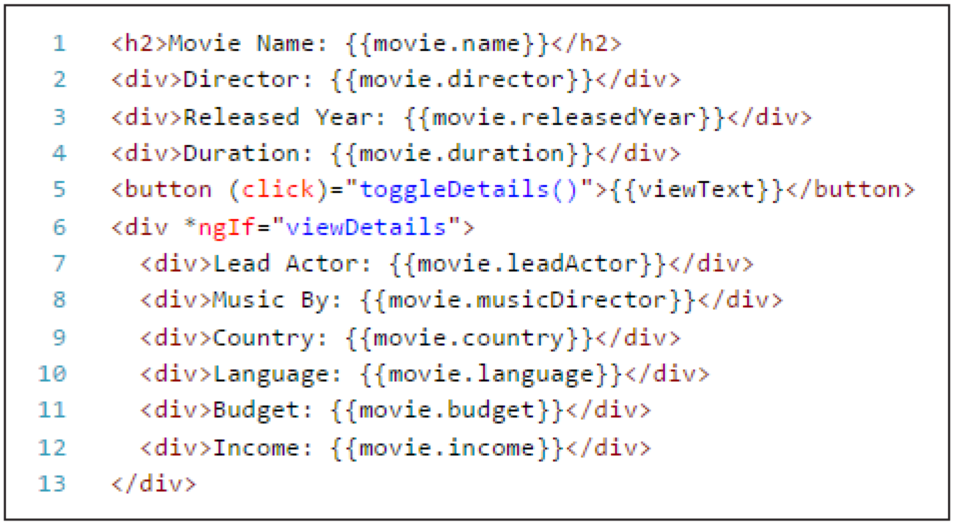
Listing 9.4: *Code of MovieDetailsComponent*



```
@Component({
  selector: "movie-details",
  templateUrl: "app/moviedetails.component.html"
})
export class MovieDetailsComponent {
    movie: Movie;
    viewDetails: boolean;
    viewText: string;

    constructor() {
        this.movie = {
            name: "A Beautiful Mind",
            director: "Ron Howard",
            releasedYear: 2001,
            duration: 135,
            leadActor: "Russel Crowe",
            musicDirector: "James Horner",
            country: "USA",
            language: "English",
            budget: "$58 million",
            income: "$313.6 million"
        };

        this.viewDetails = false;
        this.viewText = "Show More";
    }

    toggleDetails() {
        this.viewDetails = !this.viewDetails;
        this.viewText = this.viewDetails ? "Show Less" : "Show More";
    }
}
```

The property *templateUrl* in the above component holds the URL of the file which contains the HTML template to be rendered in the component. The component handles the functionality (to toggle the display of additional details of the movie) on the click of a button. Listing 9.5 shows the content of the template file:

Listing 9.5: *Template of MovieDetailsComponent*



```
<h2>Movie Name: {{movie.name}}</h2>
<div>Director: {{movie.director}}</div>
<div>Released Year: {{movie.releasedYear}}</div>
<div>Duration: {{movie.duration}}</div>
<button (click)="toggleDetails()">{{viewText}}</button>
<div *ngIf="viewDetails">
  <div>Lead Actor: {{movie.leadActor}}</div>
  <div>Music By: {{movie.musicDirector}}</div>
  <div>Country: {{movie.country}}</div>
  <div>Language: {{movie.language}}</div>
  <div>Budget: {{movie.budget}}</div>
  <div>Income: {{movie.income}}</div>
</div>
```

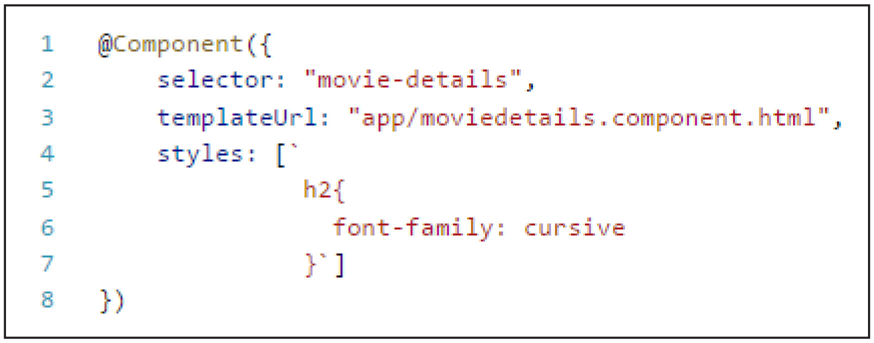
Styling Components

Applying CSS styles to an element at the nth level of an HTML document is a tiring task as it is challenging to reach the right element using CSS selectors. Any mistake in reaching the right element will not apply the style to the desired element and sometimes the style may be applied to a non-targeted element. The CSS files of the application will become bulky with many styles and complex CSS selectors. During maintenance, these files can be very frustrating for someone reading the code for the first time.

The components provide a way to define scoped styles, which can be applied to the component alone without affecting the rest of the page. Simple styles can be defined inside the components for a page where the component will be rendered.

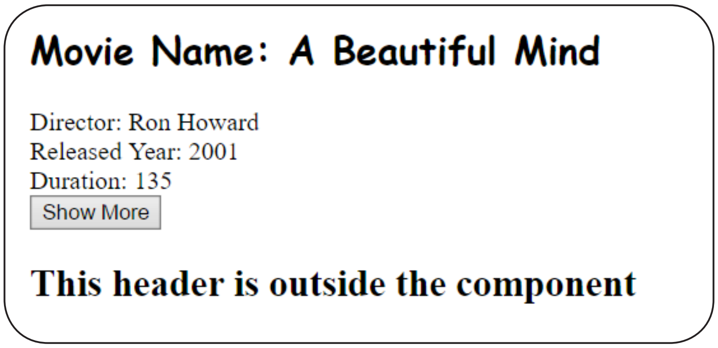
Let's apply some styles to the *MovieDetailsComponent* created in Listing 9.4. Add a new property named *styles* to the component and assign a font family to the header tag.

Listing 9.6: *Annotation of MovieDetailsComponent*



```
@Component({
    selector: "movie-details",
    templateUrl: "app/moviedetails.component.html",
    styles: [`
              h2{
                font-family: cursive
              }`]
})
```

Save it and check the page; the *h2* element in the component now has the font family applied to it. To check if the style is in scope, you may add another *h2* tag on the page and check it. The *h2* element outside the component won't get this style.



Movie Name: A Beautiful Mind

Director: Ron Howard
Released Year: 2001
Duration: 135

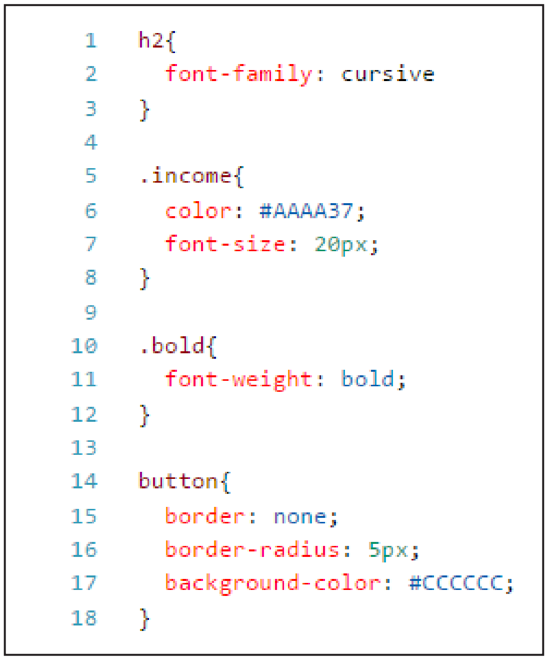
This header is outside the component

Figure 9.1: *Applying inline styles to a component*

The *styles* property accepts an array of strings so the styles can be separated into different strings, depending on: the target element, portion of the component where it has to be applied, or based on any other pattern decided for the application.

When the amount of CSS to be applied to the component grows, the group of styles can be moved into a separate file. Let's apply additional styles to the *MovieDetailsComponent* to make it look better. Add a new file and name it *moviedetails.component.css.* Note the consistent naming convention followed for all files related to the component. Add the following styles to this component:

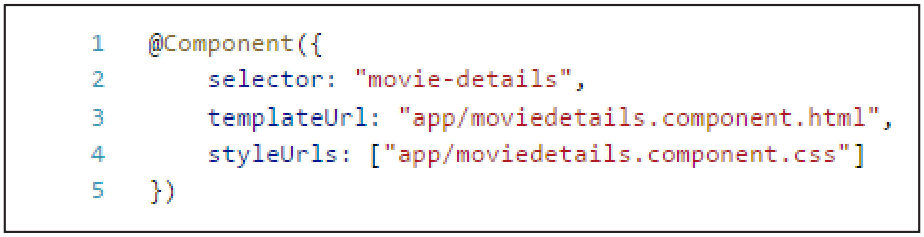
Listing 9.7: *Styles of MovieDetailsComponent*



```css
h2{
  font-family: cursive
}

.income{
  color: #AAAA37;
  font-size: 20px;
}

.bold{
  font-weight: bold;
}

button{
  border: none;
  border-radius: 5px;
  background-color: #CCCCCC;
}
```

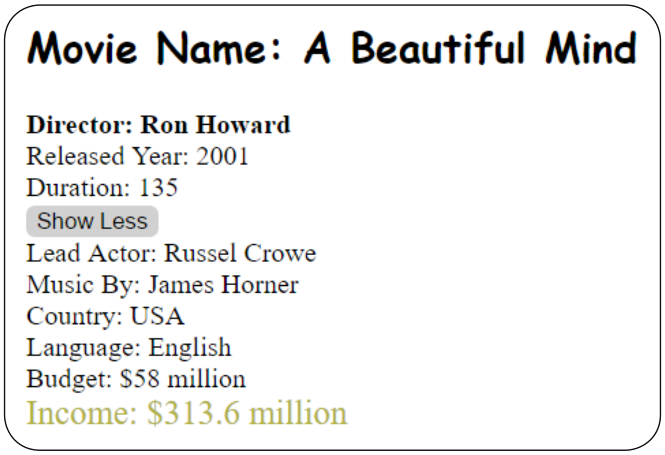
This file can be applied to the component as shown in Listing 9.8:

Listing 9.8: *Annotation of MovieDetailsComponent*



```
@Component({
    selector: "movie-details",
    templateUrl: "app/moviedetails.component.html",
    styleUrls: ["app/moviedetails.component.css"]
})
```

Save the files and run the application. The component looks similar to Figure 9.2:



Movie Name: A Beautiful Mind

Director: Ron Howard
Released Year: 2001
Duration: 135

Lead Actor: Russel Crowe
Music By: James Horner
Country: USA
Language: English
Budget: $58 million
Income: $313.6 million

Figure 9.2: *Applying styles from an external file to a component*

If a CSS style is defined in both global level and at the component level, then the style at the component level takes precedence when the component is rendered.

Conclusion

Managing views of the components is important since the customer of the application will interact with the application through the view. Angular 2 gives us a friendly interface to manage the views effectively. The feature of scoped styling in the components is a time saver and makes it easy to program the styles according to a given design.

Chapter 10
Directive Overview

ANGULAR 2 IS CREATED with Separation Of Concerns (SoC) and testability in mind. One aspect where front- end applications need distinct sections (as well as durable communication between the sections) is between the view and the model. Components and directives in Angular 2 solve this problem well.

As discussed in Chapters 6 through 9, components are used to define custom elements. Directives, on the other hand, are used to add new functionalities to the existing HTML elements and the components.

What are Directives?

Directives are used to extend HTML, teaching new tricks to HTML elements without disturbing their actual structure and implementation. They are quite useful in scenarios where the application wants certain elements to behave in a given way, depending on a value in the model.

Like components, directives make the HTML template of the application readable. Just by reading the HTML markup of an element on the page, one understands that the element will have some additional action when it gets rendered.

As the core logic of the directives is defined in JavaScript (or, TypeScript), it can be unit tested like any other block in the application. Angular 2 doesn't let a directive access its DOM element directly, rather it provides an abstraction to work with the element. This makes the directive usable across multiple platforms.

There are two types of directives in Angular 2:

- **Attribute Directive:** The attribute directives perform their task on the element on which they are applied. They get a reference of the target element and can perform tasks like applying styles, handling events, and similar operations.
- **Structural Directives:** The structural directives manipulate the HTML template of the element on which they are applied. They access the template and can perform operations like adding or removing the template, cloning the template multiple times, and anything that needs to be performed on the template as a whole.

Exploring Built-in Attribute Directives

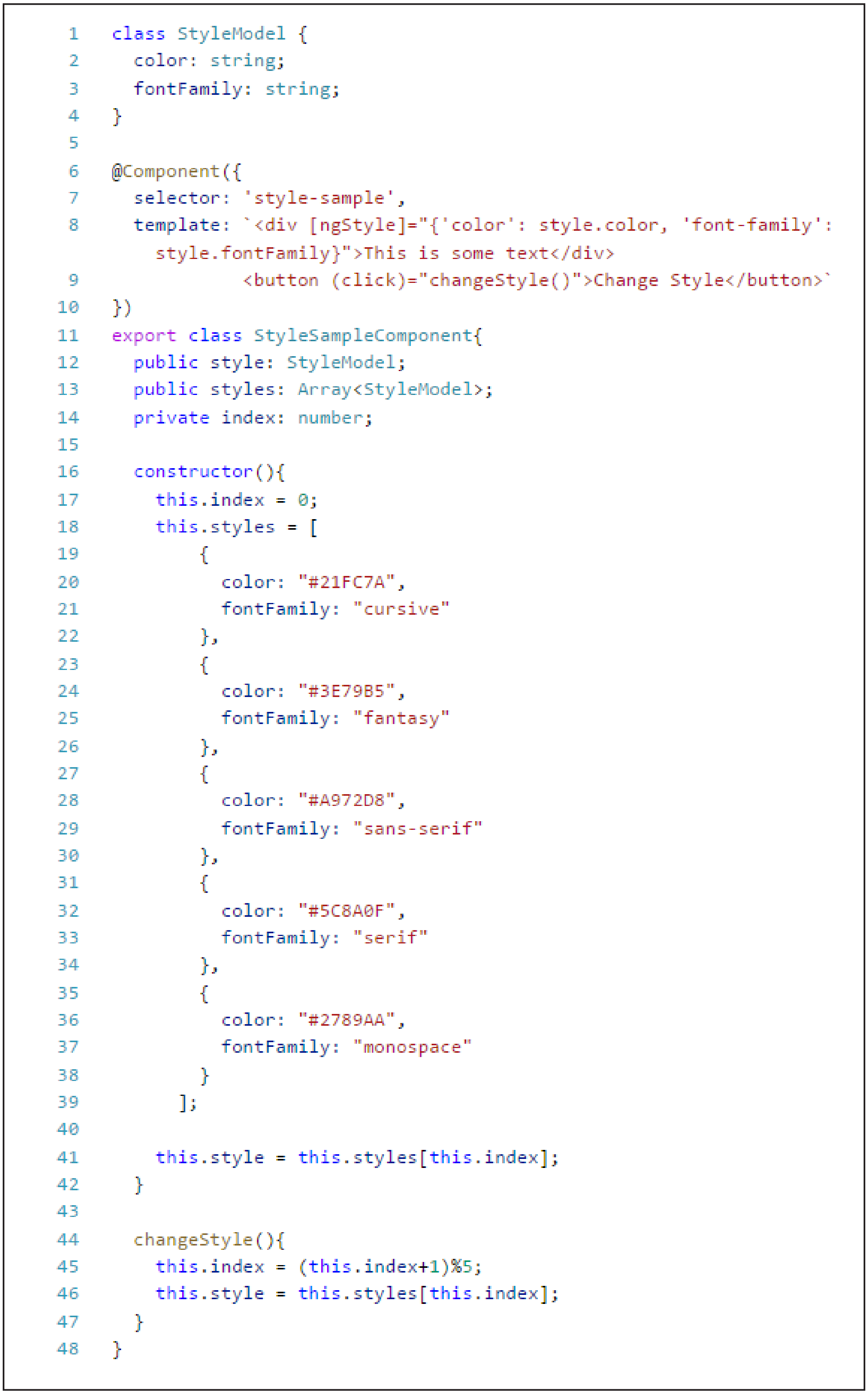
Let's see how some of the *built-in attribute* directives work in Angular 2. S*tructural* directives will be discussed in chapters 12 and 14.

ngStyle

The *ngStyle* directive is used to bind an element with dynamic inline styles. It accepts the data in the form of a JavaScript object literal, where the keys represent the CSS properties, and the values corresponding to every key are assigned to that CSS property. The values can be assigned using fields in the component's class.

Let's build a component and use the *ngStyle* directive in it to see how it works. The component will have a simple *div* and a *button*. It will have a set of five styles assigned to an array and style of the div will be changed to one of these when the button is clicked. The following snippet shows the code of the component:

Listing 10.1: *Component demonstrating ngStyle directive*



```
class StyleModel {
  color: string;
  fontFamily: string;
}

@Component({
  selector: 'style-sample',
  template: `<div [ngStyle]="{'color': style.color, 'font-family': style.fontFamily}">This is some text</div>
            <button (click)="changeStyle()">Change Style</button>`
})
export class StyleSampleComponent{
  public style: StyleModel;
  public styles: Array<StyleModel>;
  private index: number;

  constructor(){
    this.index = 0;
    this.styles = [
        {
          color: "#21FC7A",
          fontFamily: "cursive"
        },
        {
          color: "#3E79B5",
          fontFamily: "fantasy"
        },
        {
          color: "#A972D8",
          fontFamily: "sans-serif"
        },
        {
          color: "#5C8A0F",
          fontFamily: "serif"
        },
        {
          color: "#2789AA",
          fontFamily: "monospace"
        }
      ];

    this.style = this.styles[this.index];
  }

  changeStyle(){
    this.index = (this.index+1)%5;
    this.style = this.styles[this.index];
  }
}
```

In the above snippet, notice that a model class named *StyleModel* is created to represent structure of the style object, and to avoid usage of the generic *any type* in TypeScript. The *div* element in the template is applied with the first style in the array *styles*. When the button is clicked, the next color from the array *styles* is picked, and assigned to the div.

Note the way the value is assigned to the directive.

[ngStyle]="{'color': style.color, 'font-family': style.fontFamily}

The square brackets around the directive indicate that it is **property bound**. The value assigned to the directive is similar to the value assigned to the *style* attribute of HTML in general. Values assigned to the properties are inferred from the style field in the component class. Once value of the object style changes, style of the element is updated automatically by property binding.

ngClass

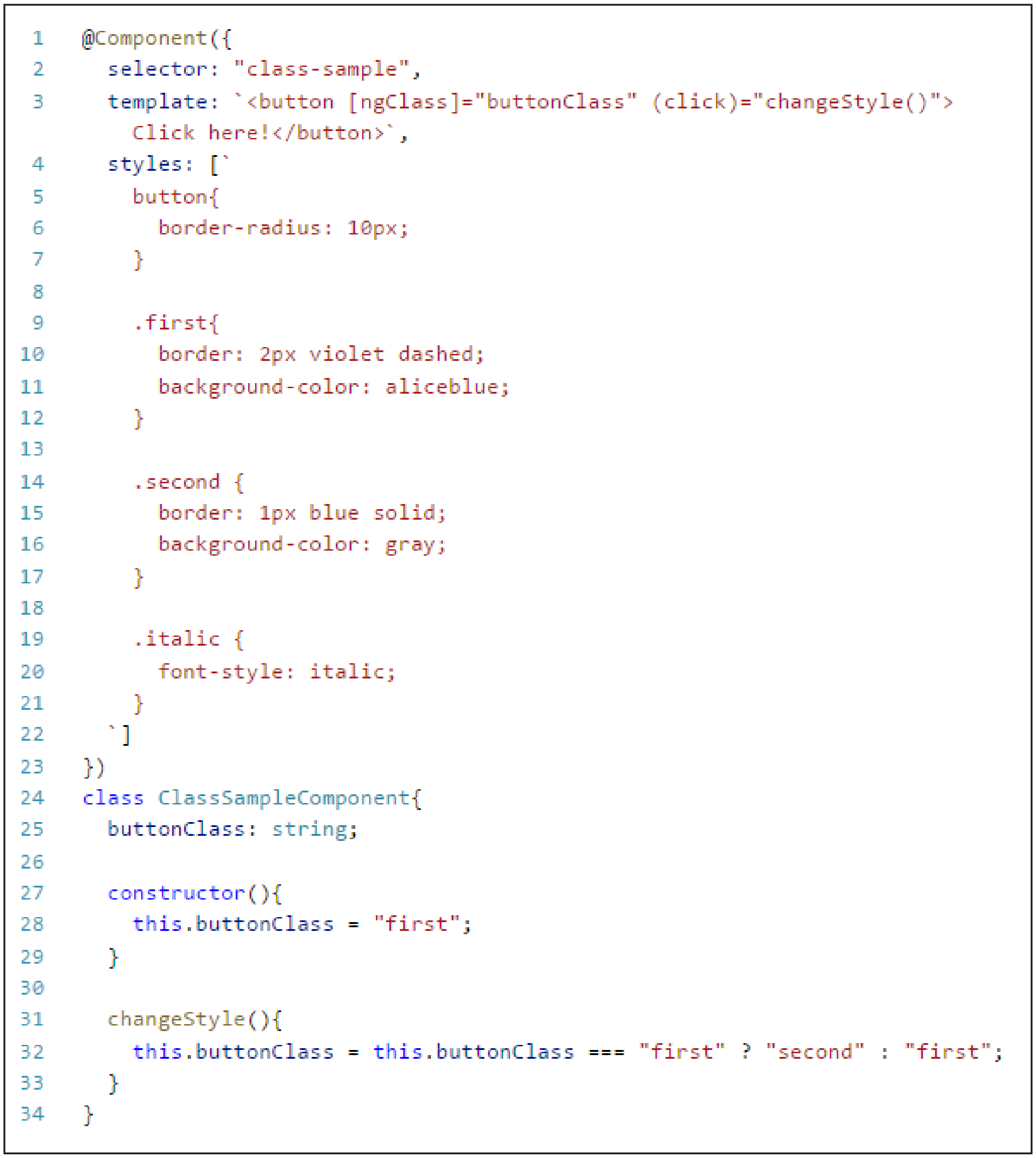
The *ngClass* directive adds or removes a CSS class on an element depending on a condition. The condition can be based on the value of a field in the component's class. Whenever the value of the field changes, the CSS class of the element also changes. There are multiple ways to use the *ngClass* directive. This section demonstrates a couple of them.

USING THE NGCLASS DIRECTIVE ON AN EVENT

Let's build a component to see how *ngClass* behaves. The component will have just one button whose CSS class toggles whenever the button is clicked. The component's *style* property will have two styles assigned to it; one will be applied to the element when the component loads and the other when the value of the class changes (to the other class) on the click of the button.

The following snippet shows code of the component:

Listing 10.2: *Component demonstrating ngClass directive*

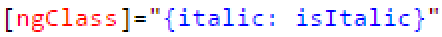


```
@Component({
  selector: "class-sample",
  template: `<button [ngClass]="buttonClass" (click)="changeStyle()">
    Click here!</button>`,
  styles: [`
    button{
      border-radius: 10px;
    }

    .first{
      border: 2px violet dashed;
      background-color: aliceblue;
    }

    .second {
      border: 1px blue solid;
      background-color: gray;
    }

    .italic {
      font-style: italic;
    }
  `]
})
class ClassSampleComponent{
  buttonClass: string;

  constructor(){
    this.buttonClass = "first";
  }

  changeStyle(){
    this.buttonClass = this.buttonClass === "first" ? "second" : "first";
  }
}
```

Here, the directive *ngClass* is assigned with the name of a field, so it has the value assigned to the field. Whenever the button is clicked, value of the field *buttonClass* changes and the button gets a new style applied.

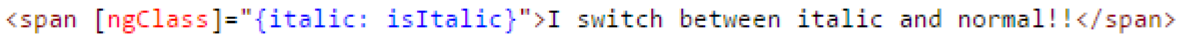
USING THE NGCLASS DIRECTIVE ON A BOOLEAN FIELD

Another way to use *ngClass* is to apply or not apply a CSS style based on a Boolean field. This requires a slightly different syntax. The following snippet shows the syntax:



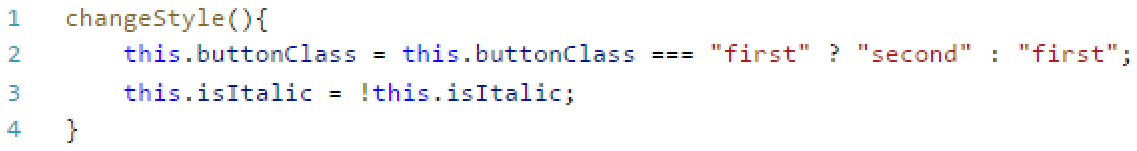
```
[ngClass]="{italic: isItalic}"
```

Let’s add a *span* element to the above component and apply this style to it.



```
<span [ngClass]="{italic: isItalic}">I switch between italic and normal!!</span>
```

The value of the Boolean property *isItalic* must be toggled to see if it works. Modify the *changeStyle* method in the class to toggle this value whenever the button is clicked.



```
changeStyle(){
    this.buttonClass = this.buttonClass === "first" ? "second" : "first";
    this.isItalic = !this.isItalic;
}
```

Conclusion

Directives provide a way to tweak the view dynamically with very few lines of code. Angular 2 has many other directives (for forms, validation, localization and routing) which will be explained in Chapters 11 through 14.

Chapter 11

Basic Custom Attribute Directive

CHAPTER 10 EXPLORED DIRECTIVES, the importance of directives, and it demonstrated two of the built-in directives in Angular 2. As discussed, Directives extend HTML elements and add additional actions to the elements without modifying their originality. This approach is quite powerful.

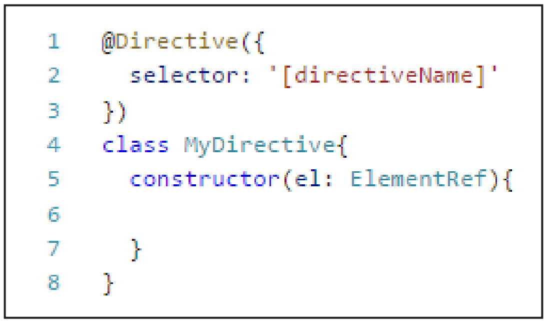
This chapter will demonstrate how to create a basic directive and use it.

While writing a full blown business application using Angular 2, sometimes we need the control of adding a business-specific meaning to the application. The dynamic nature of components and directives in Angular 2 provide us with this option. Directives can be used to add behavior such as a particular style property according to a value in the model, handling of events, and any additional behavior on the element that enhances its functionality.

Building a Custom Attribute Directive

Like a component, a directive is a TypeScript class with an annotation. The following snippet shows syntax of creating a directive:

Listing 11.1: *Syntax of creating a directive*



```
@Directive({
  selector: '[directiveName]'
})
class MyDirective{
  constructor(el: ElementRef){

  }
}
```

The directive must be applied with the *Directive* annotation, which defines the selector of the directive. It can also provide hooks to interact with the containing component, which will be discussed later in this chapter. The reference of the element on which the directive is applied must be injected into the constructor of the directive. *ElementRef* is the type of the element reference (TypeScript), which is defined in the core package of Angular 2.

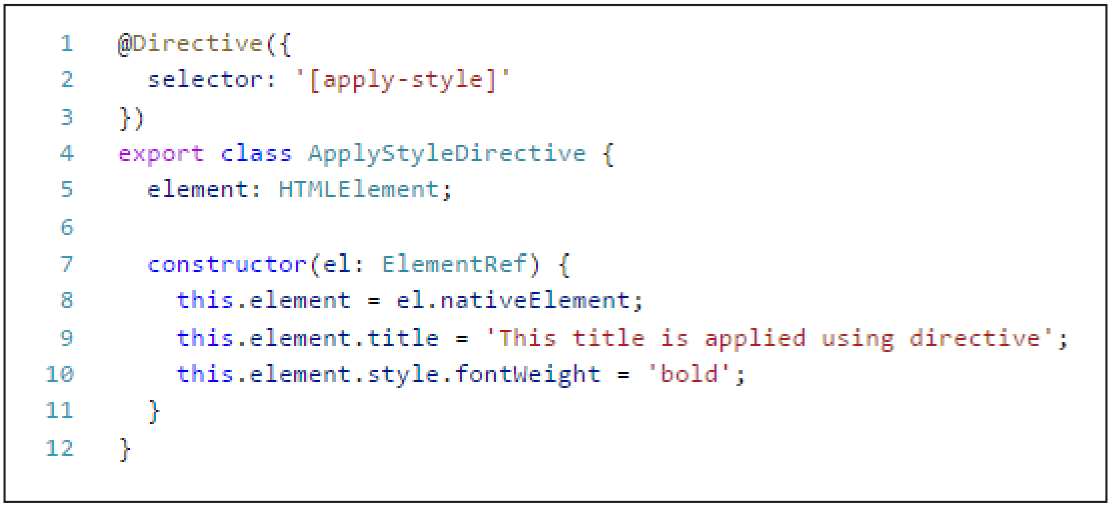
Writing a Basic Directive

Let’s write a basic directive to understand the process. The directive will make the following changes to the applied element:

- Sets a title to the element
- Changes font weight of the element to bold

Listing 11.2 shows the definition of this directive:

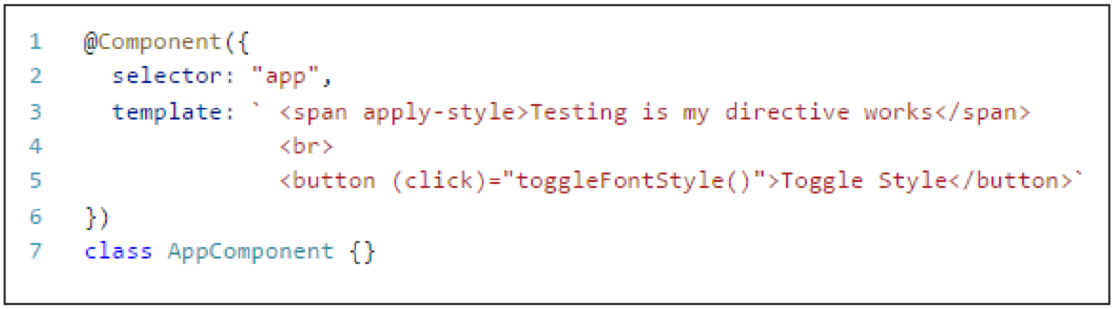
Listing 11.2: *Code of ApplyStyleDirective*



```
@Directive({
  selector: '[apply-style]'
})
export class ApplyStyleDirective {
  element: HTMLElement;

  constructor(el: ElementRef) {
    this.element = el.nativeElement;
    this.element.title = 'This title is applied using directive';
    this.element.style.fontWeight = 'bold';
  }
}
```

This directive can be used on any element on the page. Before using it, the directive has to be declared either in the component or in the module. The following component uses this directive:

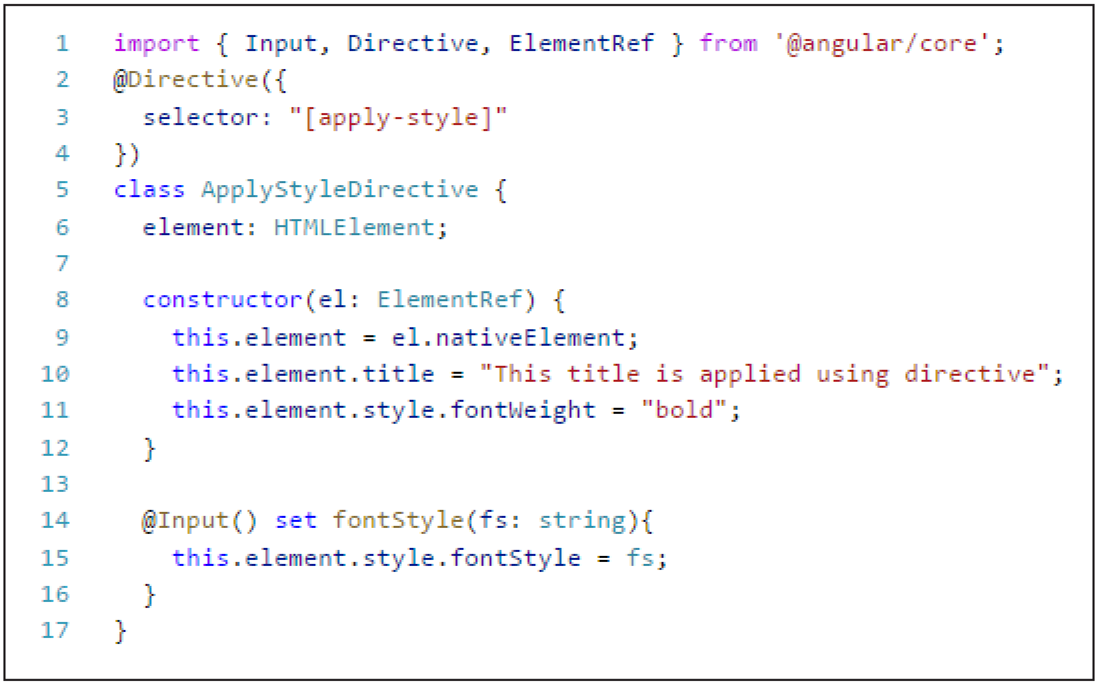
Listing 11.3: *Code of AppComponent*



```
@Component({
  selector: "app",
  template: ` <span apply-style>Testing is my directive works</span>
              <br>
              <button (click)="toggleFontStyle()">Toggle Style</button>`
})
class AppComponent {}
```

The directives can accept properties (which can be assigned with the value of a field in the component) by introducing an input property to the directive class. It is similar to the input properties discussed in Chapter 8. As an example, modify the *ApplyStyleDirective* to accept a property. The following snippet shows the modified directive:

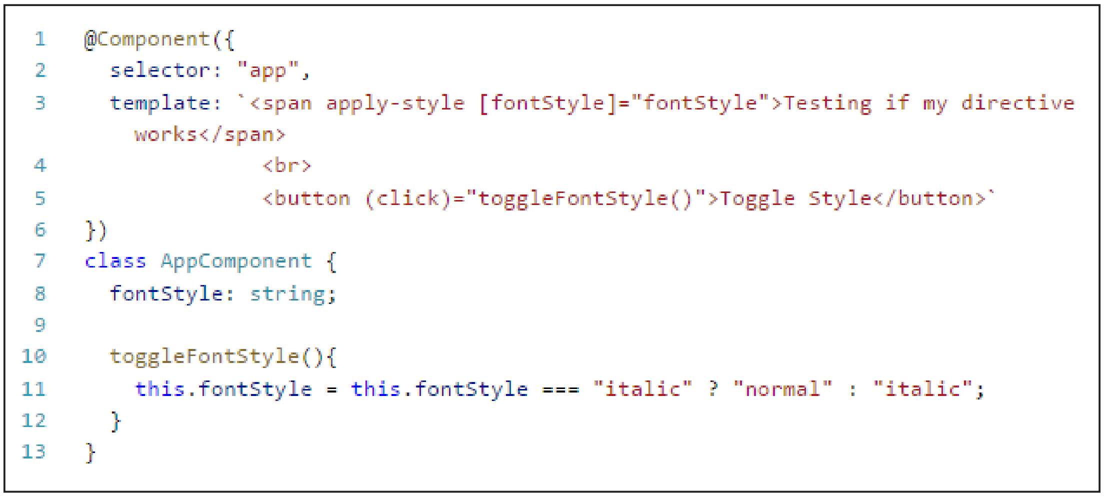
Listing 11.4: *ApplyStyleDirective with properties*



```
import { Input, Directive, ElementRef } from '@angular/core';
@Directive({
  selector: "[apply-style]"
})
class ApplyStyleDirective {
  element: HTMLElement;

  constructor(el: ElementRef) {
    this.element = el.nativeElement;
    this.element.title = "This title is applied using directive";
    this.element.style.fontWeight = "bold";
  }

  @Input() set fontStyle(fs: string){
    this.element.style.fontStyle = fs;
  }
}
```

The directive class includes a *setter* property *fontStyle*, which is invoked whenever the value of the bound property changes. This property is decorated with the *Input* decorator. The incoming value is assigned to the *fontStyle* property of style of the element. The component using this directive provides this value. Whenever the component changes the value, the *setter* property is called, and the value is updated. Let's modify the *AppComponent* to have a field for font style and change its value using an event. Listing 11.5 snippet shows the modified code of *AppComponent*:

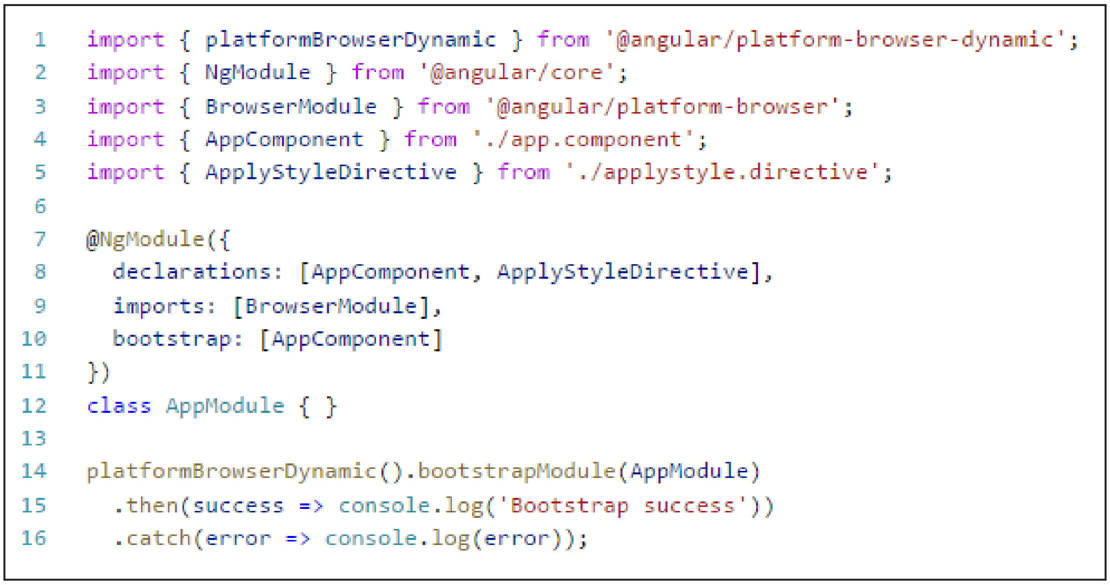
Listing 11.5: *AppComponent with property fontStyle*



```
@Component({
  selector: "app",
  template: `<span apply-style [fontStyle]="fontStyle">Testing if my directive works</span>
             <br>
             <button (click)="toggleFontStyle()">Toggle Style</button>`
})
class AppComponent {
  fontStyle: string;

  toggleFontStyle(){
    this.fontStyle = this.fontStyle === "italic" ? "normal" : "italic";
  }
}
```

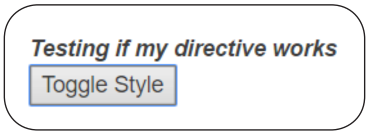
A module needs to be created to hold the directive and the component created in Listing 11.5, and the application has to be bootstrapped using the module. Listing 11.6 shows the module.

Listing 11.6: *AppModule registering AppComponent and ApplyStyleDirective*



```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { ApplyStyleDirective } from './applystyle.directive';

@NgModule({
  declarations: [AppComponent, ApplyStyleDirective],
  imports: [BrowserModule],
  bootstrap: [AppComponent]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule)
  .then(success => console.log('Bootstrap success'))
  .catch(error => console.log(error));
```

Run the example and click the Toggle Style Button. You should see the following:



Testing if my directive works

Figure 11.1: *Preview of the page*

Conclusion

Directives can be used to give minor, yet important touches to the element they are applied on. This chapter demonstrated a simple custom directive. Chapter 12 will discuss how to create a more interactive directive.

Chapter 12

Advanced Custom Attribute Directives

CHAPTERS 10 AND 11 demonstrated what directives are, explained the importance of directives, used some built-in directives in Angular 2, and also showed a demo of a simple custom directive. As discussed in these chapters, directives extend HTML elements and add actions to these elements, without modifying their originality.

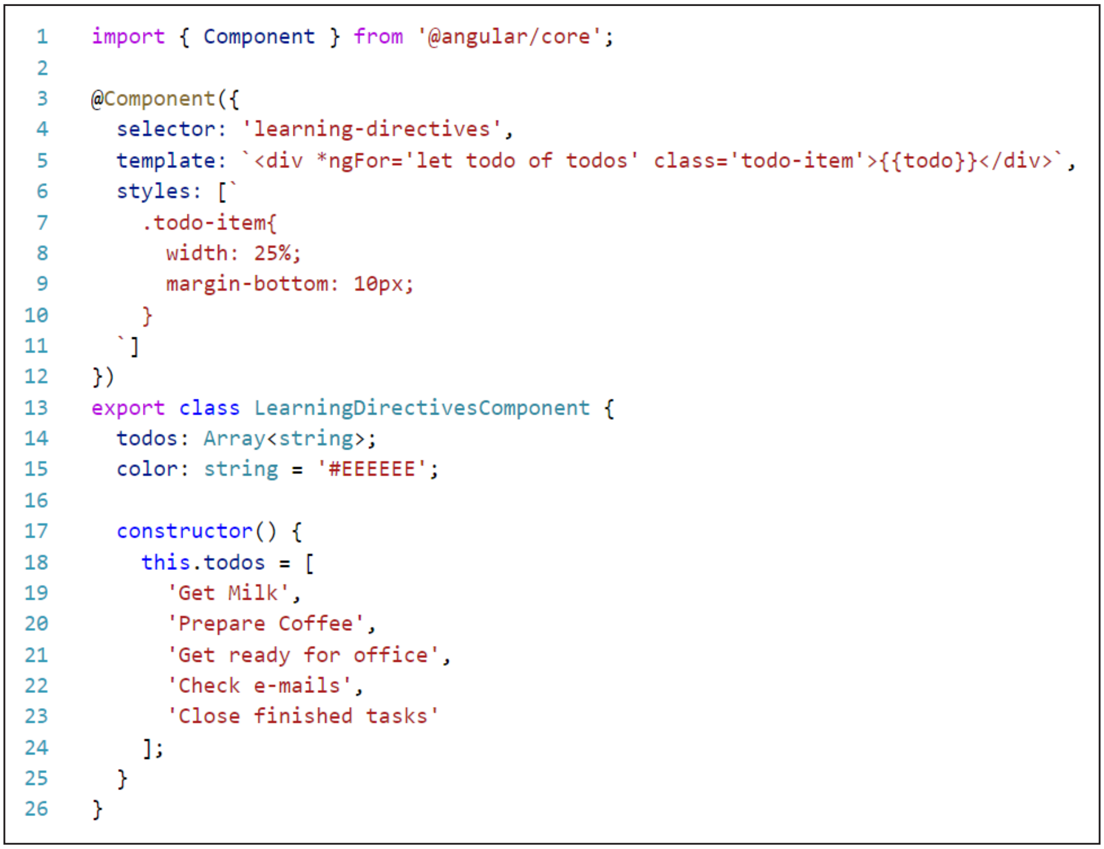
This chapter will build another custom directive to explore some additional features of custom directives.

Directive to Highlight an Element

This section will create a directive to highlight an element when it is clicked upon. The component consuming this directive will control the color to be applied in the background when the element is highlighted.

First, we need a component with a template to host the directive. Listing 12.1 shows the component and displays a list of to-do items in *div* elements. Each of these *div*s will be applied with the directive to be built. Snippet 12.1 defines this component:

Listing 12.1: *Code of the file learningdirectives.component.ts*



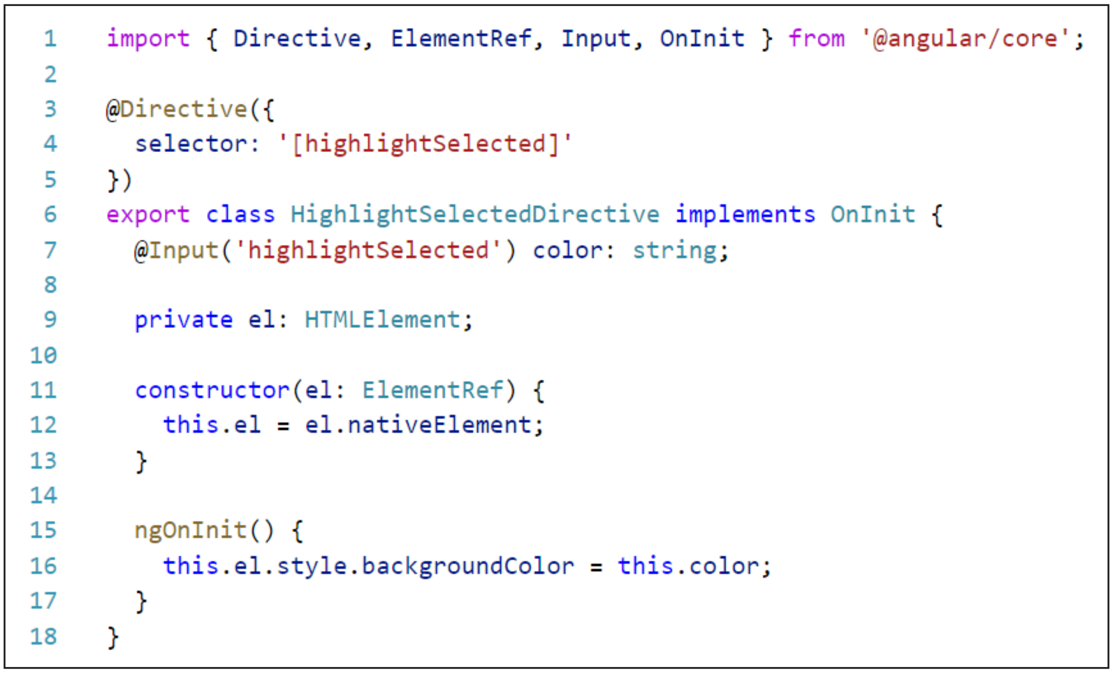
```
import { Component } from '@angular/core';

@Component({
  selector: 'learning-directives',
  template: `<div *ngFor='let todo of todos' class='todo-item'>{{todo}}</div>`,
  styles: [`
    .todo-item{
      width: 25%;
      margin-bottom: 10px;
    }
  `]
})
export class LearningDirectivesComponent {
  todos: Array<string>;
  color: string = '#EEEEEE';

  constructor() {
    this.todos = [
      'Get Milk',
      'Prepare Coffee',
      'Get ready for office',
      'Check e-mails',
      'Close finished tasks'
    ];
  }
}
```

The component has a property, *color,* that holds the value of the color to be applied to highlight the to-do items. This property is passed to the directive. Chapter 8 demonstrated how to make two components communicate with each other using the *Input* and *Output* decorators. The same principles can be used to establish communication between components and directives as well. The directive will have an *Input* field to accept the value of color from the containing component.

Like components, directives have lifecycle hooks. The color received from the component must be used in the directive in the *ngOnInit* lifecycle hook because this value wouldn't have been assigned when the directive was constructed.

Snippet 12.2 defines the directive:

Listing 12.2: *Code of the file highlightselected.directive.ts*



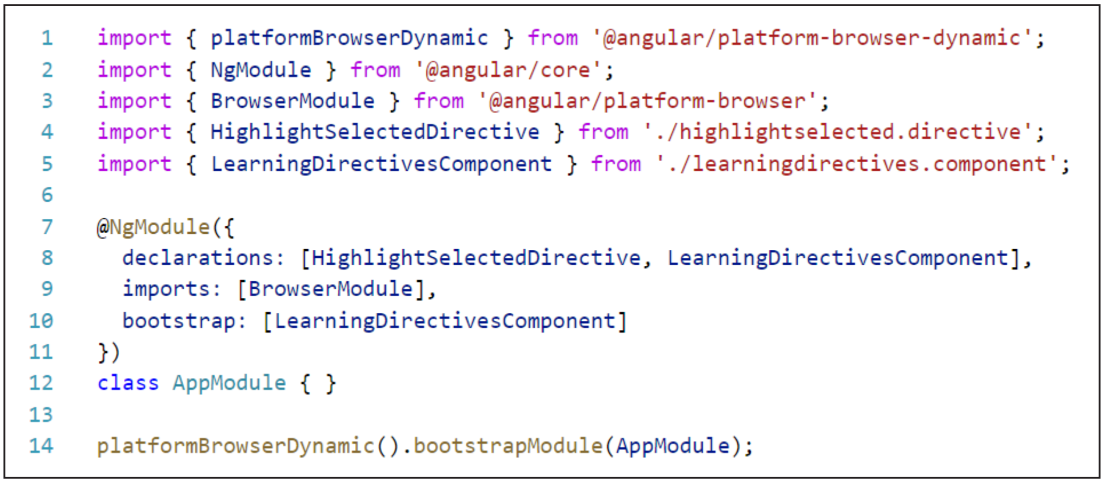
```
import { Directive, ElementRef, Input, OnInit } from '@angular/core';

@Directive({
  selector: '[highlightSelected]'
})
export class HighlightSelectedDirective implements OnInit {
  @Input('highlightSelected') color: string;

  private el: HTMLElement;

  constructor(el: ElementRef) {
    this.el = el.nativeElement;
  }

  ngOnInit() {
    this.el.style.backgroundColor = this.color;
  }
}
```

In the snippet 12.2,

- The input property *color* has the value of the property assigned to the attribute named *highlightSelected,* which is the directive itself.
- The constructor gets the reference of the DOM element on which the directive is applied. Unlike Angular 1, where the DOM elements are wrapped inside jqlite, Angular 2 provides a plain DOM object.
- The method *ngOnInit* sets the background color of the element using the color received in the input parameter.

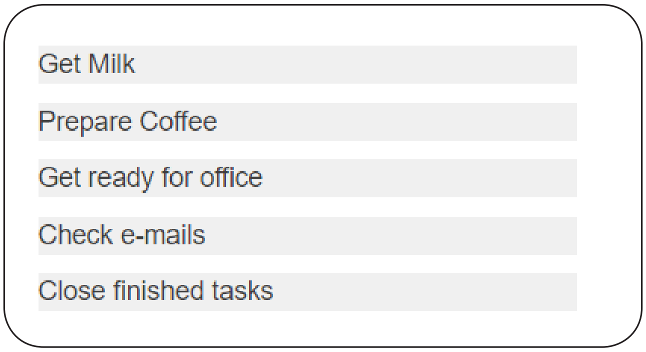
To use this directive in the component, the directive has to be declared in the *declarations* section of the module. Now apply the directive on the *div* element. Listing 12.3 shows the definition of the module:

Listing 12.3: *Module declaring the component and directive in the file main.ts*



```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HighlightSelectedDirective } from './highlightselected.directive';
import { LearningDirectivesComponent } from './learningdirectives.component';

@NgModule({
  declarations: [HighlightSelectedDirective, LearningDirectivesComponent],
  imports: [BrowserModule],
  bootstrap: [LearningDirectivesComponent]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Upon running the application, the component will have the list of to-do items with the background color.



Get Milk

Prepare Coffee

Get ready for office

Check e-mails

Close finished tasks

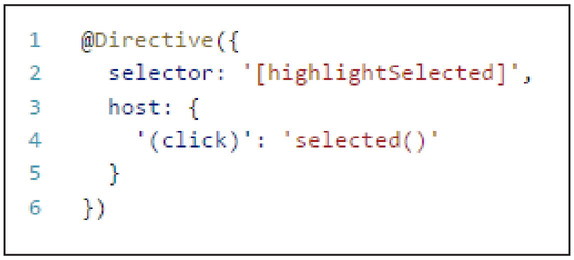
Figure 12.1: *To-do items with background color*

Highlighting all of these entries by default isn't ideal for the user. Let's modify this behavior to highlight an entry when the user clicks on it.

Highlighting on Click

To highlight divs on click, the Click event of the div element has to be handled. The previous chapters demonstrated this using a method in the component class. Now that the directive *HighlightSelectedDirective* is responsible to highlight the elements, this directive should be modified to follow the user's action. The directive can listen to the events of its container elements using the *host* property on the *Directive* annotation. The syntax of using this property is shown in listing 12.4:

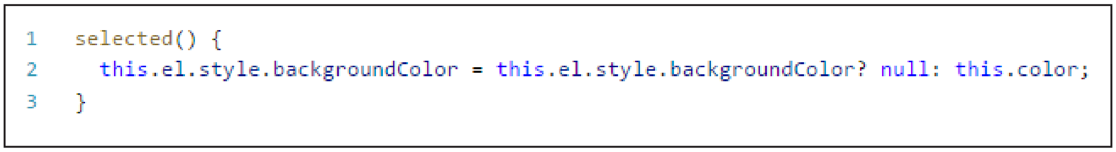
Listing 12.4: *Directive annotation with host property*



```
@Directive({
  selector: '[highlightSelected]',
  host: {
    '(click)': 'selected()'
  }
})
```

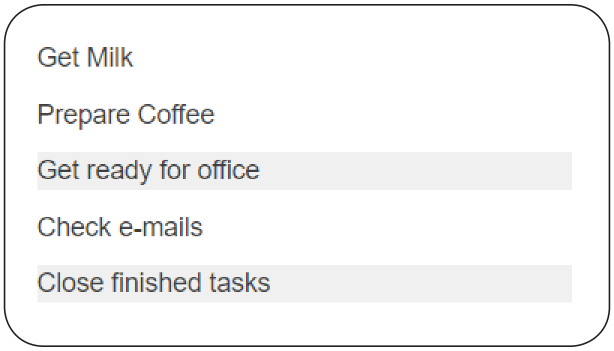
The method *selected* has to be defined in the directive class; it will toggle the background color of the element. The ternary operator ? helps achieve this in a single statement.

Listing 12.5: *Method assigning style in the directive*



```
selected() {
  this.el.style.backgroundColor = this.el.style.backgroundColor? null: this.color;
}
```

No changes are required in the component. Now if you run the application, you should be able to see the background color of the to-do items toggling when you click them. Run the application and click on the to-do items to see the background color changing.



Get Milk

Prepare Coffee

Get ready for office

Check e-mails

Close finished tasks

Figure 12.2: *Output of the final code*

Conclusion

As seen from the examples in this chapter, directives can be used to enhance the behavior of existing HTML elements according to the application's requirements. As they are not coupled to the type of element they are applied on, they are highly reusable. Upcoming chapters will define other types of directives with examples.

Chapter
13

Structural Directives

DIRECTIVES CAN BE USED to enhance the behavior of HTML; they are extremely useful in adding small, albeit important touches to the application. Directives can also be used to manipulate the template inside the element. Such directives are called **structural directives**.

Structural directives get access to the template of the element on which they are applied. They can control the way the template is displayed in the element, add the template, remove the template, and repeat this operation several times.

This chapter will demonstrate some of Angular 2's built in structural directives.

Conditionally Rendering Elements Using ngIf

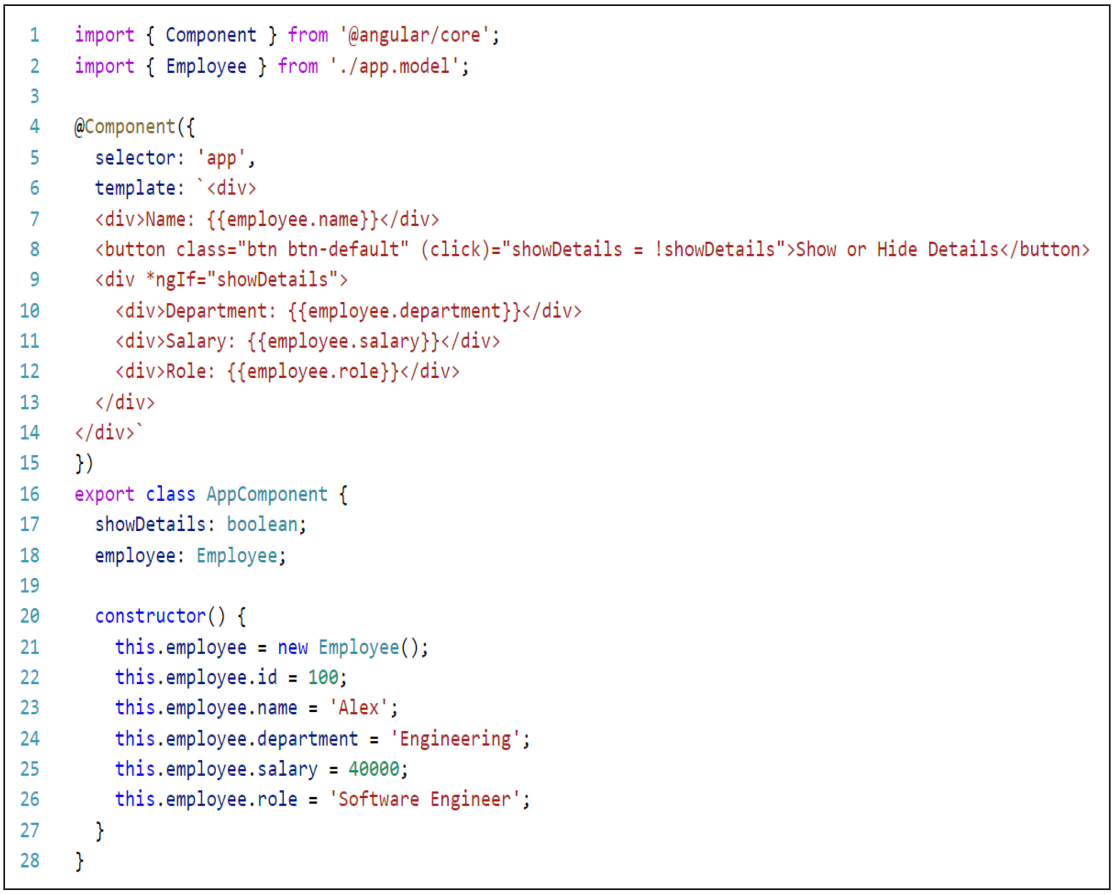
One of the common use cases of any application is to show or hide the DOM elements on the page based on a condition. This can be achieved using one of the following approaches:

- Hide the UI element from the page, and display it when required
- Completely remove the element, and add it back later

Of these two options, the second one is more difficult as it involves removing and adding elements to the page, which may include Angular 2 components and directives. Angular 2 makes it easy by providing the structural directive *ngIf*. The directive *ngIf* can be applied on any element or component in a page, and the target is added or removed from the page based on a condition. This condition can be based on the value of a field in the component class or the return value of a method.

Snippet 13.1 shows a component using *ngIf:*

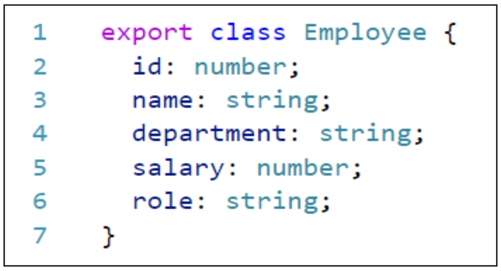
Listing 13.1: *Component using ngIf in the file app.component.ts*



```
import { Component } from '@angular/core';
import { Employee } from './app.model';

@Component({
  selector: 'app',
  template: `<div>
  <div>Name: {{employee.name}}</div>
  <button class="btn btn-default" (click)="showDetails = !showDetails">Show or Hide Details</button>
  <div *ngIf="showDetails">
    <div>Department: {{employee.department}}</div>
    <div>Salary: {{employee.salary}}</div>
    <div>Role: {{employee.role}}</div>
  </div>
</div>`
})
export class AppComponent {
  showDetails: boolean;
  employee: Employee;

  constructor() {
    this.employee = new Employee();
    this.employee.id = 100;
    this.employee.name = 'Alex';
    this.employee.department = 'Engineering';
    this.employee.salary = 40000;
    this.employee.role = 'Software Engineer';
  }
}
```

The listing 13.2 shows the Employee class to be added to a new file app.model.ts:

Listing 13.2: *Employee class in app.model.ts*



```
export class Employee {
  id: number;
  name: string;
  department: string;
  salary: number;
  role: string;
}
```

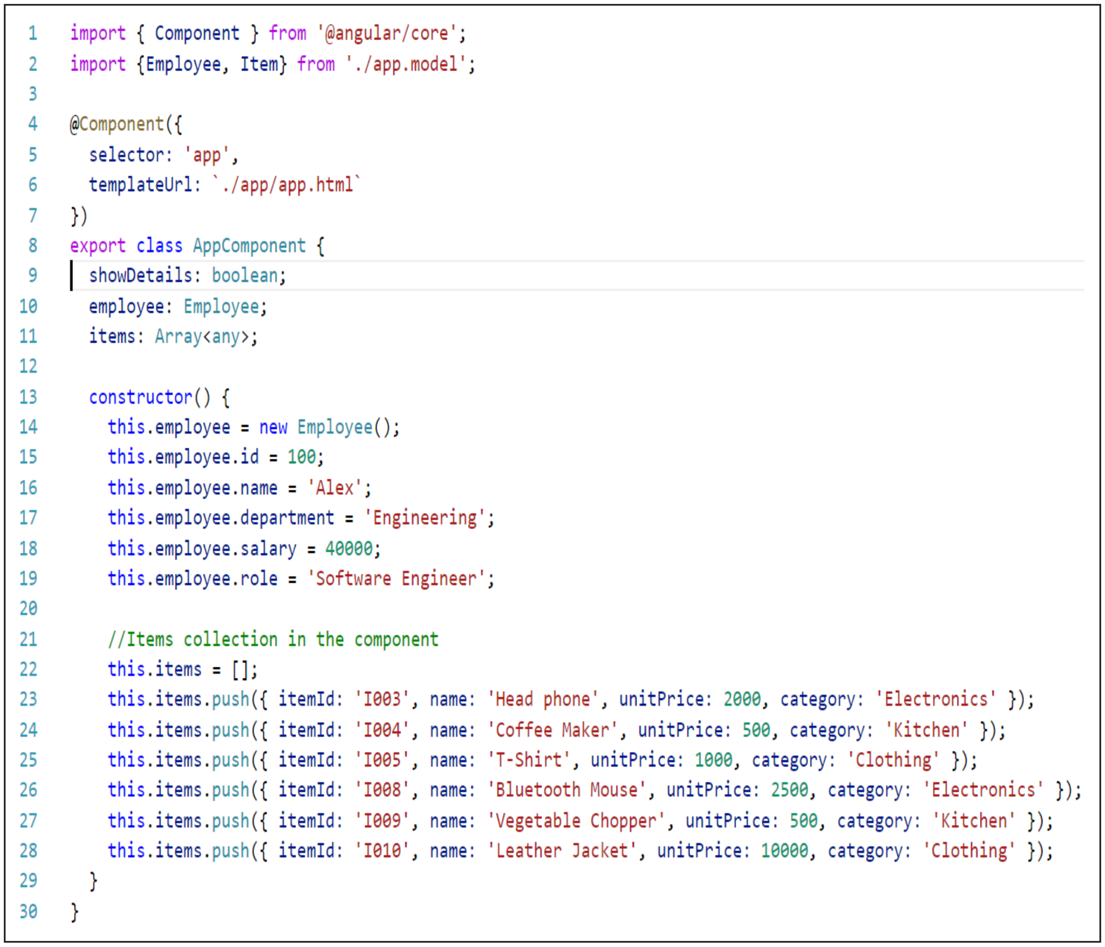
In Listing 13.1, the content inside the *div,* containing details of the element, is toggled based on the value of *showDetails*. If you inspect the content of the page, you will see that the *div* element is added to the page when value of the variable *showDetails* is set to true. Similarly, the content is removed from the page when the value is set to false.

Notice the asterisk (*) symbol in front of the directive *ngIf, indicating* that it is a structural directive and Angular 2 processes it differently than attribute directives. The directive gets a reference of the template to be rendered **inside** the target element, and so it can manipulate the template before rendering it on the UI.

Repeating a Template Using ngFor

Typical web applications can receive collections from the server to display to the user. The application repeats the content for the length of the collection. Angular 2 provides the structural directive *ngFor* to repeat the template. Suppose there is a list of items to be displayed in a component as shown in Listing 13.3. The component AppComponent has to be modified to add this list.

Listing 13.3: *Modified app.component.ts*

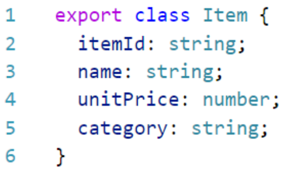


```
import { Component } from '@angular/core';
import {Employee, Item} from './app.model';

@Component({
  selector: 'app',
  templateUrl: `./app/app.html`
})
export class AppComponent {
  showDetails: boolean;
  employee: Employee;
  items: Array<any>;

  constructor() {
    this.employee = new Employee();
    this.employee.id = 100;
    this.employee.name = 'Alex';
    this.employee.department = 'Engineering';
    this.employee.salary = 40000;
    this.employee.role = 'Software Engineer';

    //Items collection in the component
    this.items = [];
    this.items.push({ itemId: 'I003', name: 'Head phone', unitPrice: 2000, category: 'Electronics' });
    this.items.push({ itemId: 'I004', name: 'Coffee Maker', unitPrice: 500, category: 'Kitchen' });
    this.items.push({ itemId: 'I005', name: 'T-Shirt', unitPrice: 1000, category: 'Clothing' });
    this.items.push({ itemId: 'I008', name: 'Bluetooth Mouse', unitPrice: 2500, category: 'Electronics' });
    this.items.push({ itemId: 'I009', name: 'Vegetable Chopper', unitPrice: 500, category: 'Kitchen' });
    this.items.push({ itemId: 'I010', name: 'Leather Jacket', unitPrice: 10000, category: 'Clothing' });
  }
}
```

Notice that the template of the component has been moved to a separate file app.component.html, as the template file will be updated with some more html later.

The listing 13.4 shows the Item class to be added to the file app.model.ts:

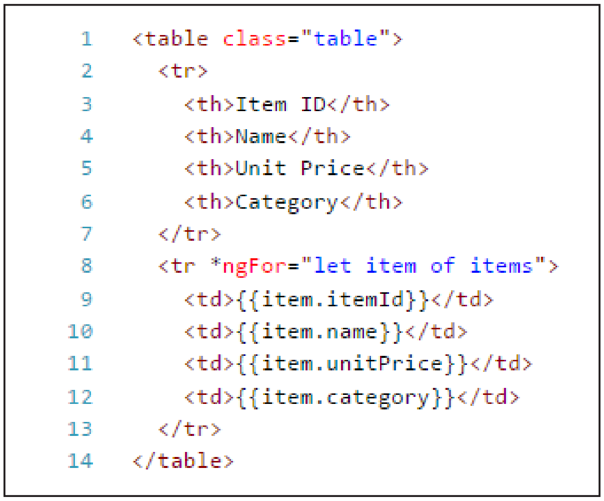
Listing 13.4: *Table with ngFor*



```
export class Item {
  itemId: string;
  name: string;
  unitPrice: number;
  category: string;
}
```

Now display this list in a table. The rows of the table will have the same template and need to be repeated for every entry in the collection. This can be done using the *ngFor* directive as shown in the snippet below, this snippet has to be added to the file app.component.html:

Listing 13.5: *Table with ngFor*



```
<table class="table">
  <tr>
    <th>Item ID</th>
    <th>Name</th>
    <th>Unit Price</th>
    <th>Category</th>
  </tr>
  <tr *ngFor="let item of items">
    <td>{{item.itemId}}</td>
    <td>{{item.name}}</td>
    <td>{{item.unitPrice}}</td>
    <td>{{item.category}}</td>
  </tr>
</table>
```

Notice the way the directive *ngFor* is used in this snippet. As a structural directive, it is prefixed with an asterisk(*). The value assigned to the directive is similar to the way a for-of loop is written in ES6. The *let* keyword is used to declare the local variable to hold the value of the current record and the *of* keyword is used before the list of items to be iterated. The variable remains private to the portion of the template where *ngFor* is used; it is not available anywhere outside the *ngFor* directive.

Conclusion

As discussed, structural directives in Angular 2 are used when the template in the element must behave based on the data available. Angular 2 has several more built-in structural directives which can be found in its official documentation. It is possible to build your own structural directives, which will be explored in the next chapter.

Chapter 14 Custom Structural Directives

CHAPTERS 10 TO 13 explained what directives are, different types of directives, using them, and creating some functional directives.

This chapter will teach you how to create custom structural directives.

As seen in Chapter 13, some of the built-in structural directives work with the template in the element and control the way the template is presented to the users. An application may challenge the developers to control the behavior of the templates, depending upon certain conditions. In such cases, a structural directive will be optimal.

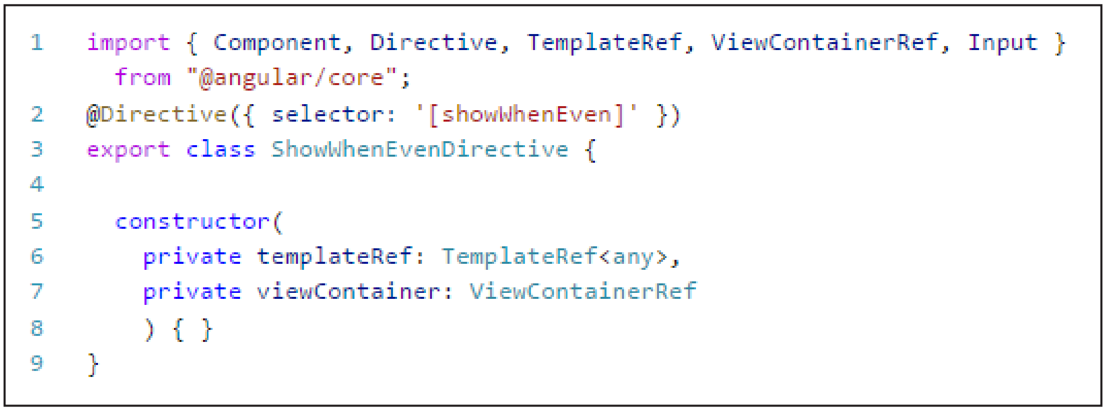
This chapter will build a simple custom structural directive to explain the process.

Building a showWhenEven Directive

Let's build a custom structural directive that shows the content, when the assigned model property is an even number, and removes the content, when the value is an odd number.

The signature of the directive will be the same as the directives discussed in the previous chapters. The only difference is in the dependencies*TemplateRef* and *ViewContainerRef* injected in the directive's constructor. The *TemplateRef* represents the template inside the element. The injected object would contain a property holding the DOM object inside the element on which the directive is applied. The *ViewContainerRef* is a container where a view can be attached. It gets the reference of the DOM content inside the element containing the structural directive. Both of these types are defined in the *@angular/core* package. Snippet 14.1 shows the directive with its constructor:

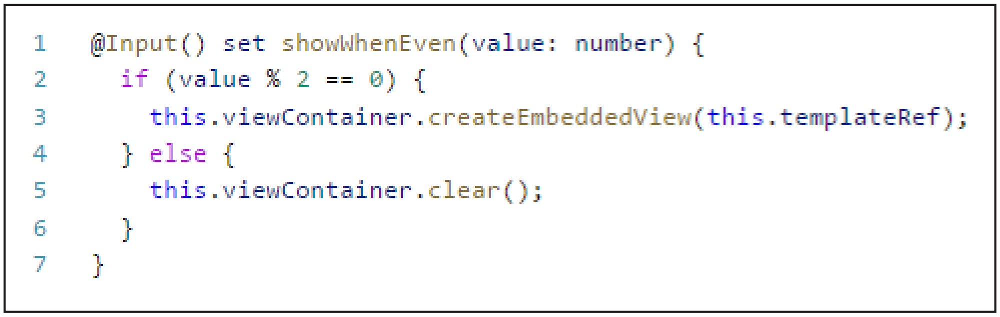
Listing 14.1: *Skeleton of custom structural directive, to be added to file showwheneven.directive.ts*



```
import { Component, Directive, TemplateRef, ViewContainerRef, Input }
  from "@angular/core";
@Directive({ selector: '[showWhenEven]' })
export class ShowWhenEvenDirective {

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef
    ) { }
}
```

The directive accepts a numeric field from the component and, based on the value of the input, content will be added or removed. The *TemplateRef* and *ViewContainerRef* objects injected into the directive will aide in this task. There must be a *setter* property in the directive to update the view whenever the value of the input changes. The name of the setter property is going to be same as the selector of the directive. Assigning the same name to the setter property allows the consumer to assign the input to the directive itself, thus makes the consumption easier. Listing 14.2 shows the setter property:

Listing 14.2: *Setter property in custom structural directive, to be added to the class in the file showwheneven.directive.ts*

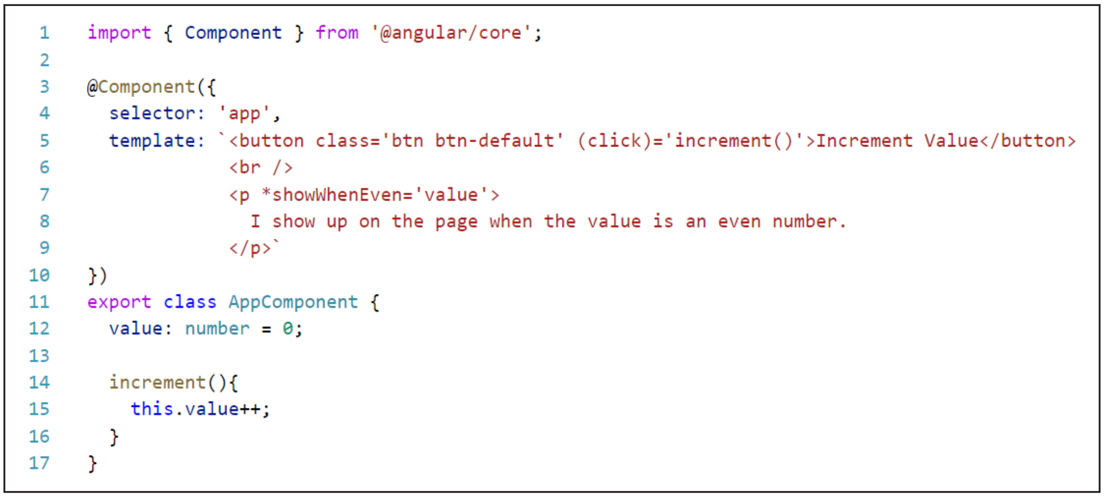


```
@Input() set showWhenEven(value: number) {
  if (value % 2 == 0) {
    this.viewContainer.createEmbeddedView(this.templateRef);
  } else {
    this.viewContainer.clear();
  }
}
```

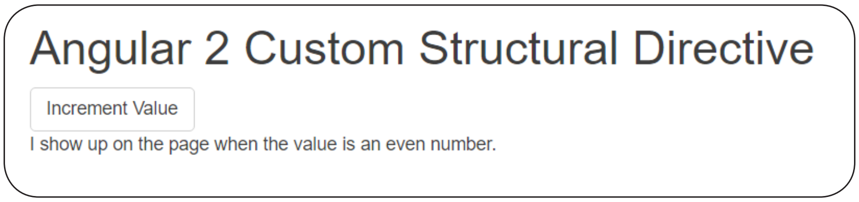
As the value of the property comes from the component, it must be marked with the annotation *Input*. Listing 14.2 checks if the value set to the property is even and, if it passes, it adds the content to the DOM using the *createEmbeddedView* method on the *viewContainer*. Otherwise, it calls the *clear* method on the *viewContainer* to remove the content inside the directive element.

Now define a component to use this directive: the component will have a numeric field and a method to increment value of this field. This method will be called from the *click* event of a button. The content inside the structural directive will be added to the view on alternate clicks of the button. Snippet 14.3 shows the component using the directive:

Listing 14.3: *Component using the custom structural directive*



```
import { Component } from '@angular/core';

@Component({
  selector: 'app',
  template: `<button class='btn btn-default' (click)='increment()'>Increment Value</button>
             <br />
             <p *showWhenEven='value'>
               I show up on the page when the value is an even number.
             </p>`
})
export class AppComponent {
  value: number = 0;

  increment(){
    this.value++;
  }
}
```



Angular 2 Custom Structural Directive

I show up on the page when the value is an even number.

Figure 14.1: *Custom structural directive in action*

Conclusion

Angular 2 provides us with different directives for different needs, which are easy to create, and make the application look more declarative. Structural directives make the page extremely dynamic and interactive by conditionally displaying the content. Now that you've learned how to create custom structural directives, use them to make your applications even better!

Chapter 15

Getting Started with Databinding

DATABINDING IS ONE OF the most appreciated features of AngularJS as it creates a connection between the application's UI and the data/business logic/model. This enables automatic synchronization of data between the model and the view. When the data changes, the UI elements bound to that data change automatically. Similarly, when the data in the UI elements changes, the underlying model is updated to reflect the changes.

Data binding eliminates writing thousands of lines of code, and the need to write tedious, boilerplate code around DOM events.

In Angular 2, the Databinding provides the following features:

- One-Way Databinding or Interpolation
- Databinding with UI Element Properties
- Databinding with UI Element Events

The following diagram gives an overview of Angular 2 Databinding

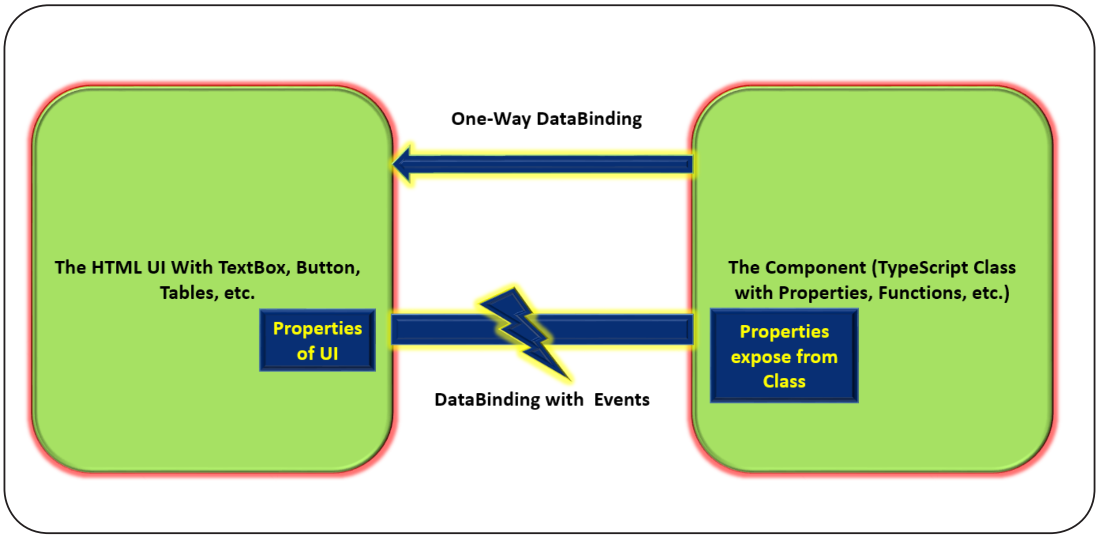

One-Way DataBinding
The HTML UI With TextBox, Button, Tables, etc.
Properties of UI
The Component (TypeScript Class with Properties, Functions, etc.)
Properties expose from Class
DataBinding with Events

Figure 15.1: *Angular 2 DataBinding*

Typically in Line-Of-Business (LOB) applications, data received from the server is exposed to the UI elements on a page. The received data is stored in the Data-Model on the client-side.

NOTE: Two-Way databinding is not explicitly available in Angular 2. It can be implemented using a combination of Property and Event Bindings.

What is Data-Model and how is it used in Databinding?

As discussed in the previous chapters, an Angular 2 application is a combination of several components. Every component has its own data model, which is an object of the **component's class.** This class contains public properties (or fields) and methods (or function/operations).

Data-Model properties are used to expose data to the UI elements; the View can use these fields for DataBinding. **One-Way** databinding is used to display data on UI, but changes to the data in UI element will not update the data-model property. In case of **Two-Way** Databinding, when the UI element changes its value, the value of the property bound with it *will* be updated with the new values from the UI element. We can bind methods from the Data-Model with the events exposed from the UI element.

How does DataBinding benefit our application?

The advantage of DataBinding is that it eliminates the UI level event handling code (remember **`document.getElementById('btn').addEventListener`** in JavaScript). Two-Way binding helps automatically update the UI when the DataModel properties are updated from the data received from server. This eliminates the code for *explicitly* updating values of UI elements. The UI will automatically generate the DOM, based on the data collection received from an external server.

Getting Started with the DataBinding

This chapter will implement a sample DataBinding application using Visual Studio Code (VS Code). Please refer to Chapter 5 to set up the development environment.

CREATING PROJECT AND INSTALLING REQUIRED PACKAGES

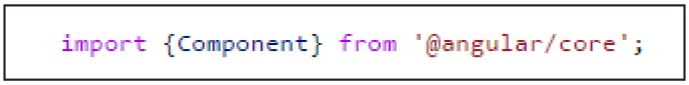
Create a folder on your drive with the name **NG2_Databinding** to be used as an **application folder**. Open VS Code and using **File > Open Folder** option, navigate to the application folder and open it. Please follow the steps mentioned in **NG Developer Environment** chapter to install necessary dependencies for the current section.

Creating Components for DataModel

In the **app** folder, add a new file of the name **employee.component.ts.** In this file, define the component *EmployeeComponent*.

The Angular 2 core modules are imported using the following statement:

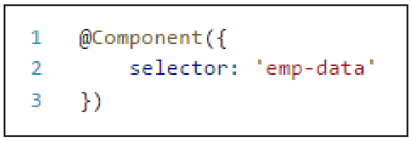
Listing 15.1: *Importing the Component*



```
import {Component} from '@angular/core';
```

The new component is created with the **selector** as **emp-data**, which will be used as an HTML tag in the View. Listing 15-2 shows the decorator to be applied on the component:

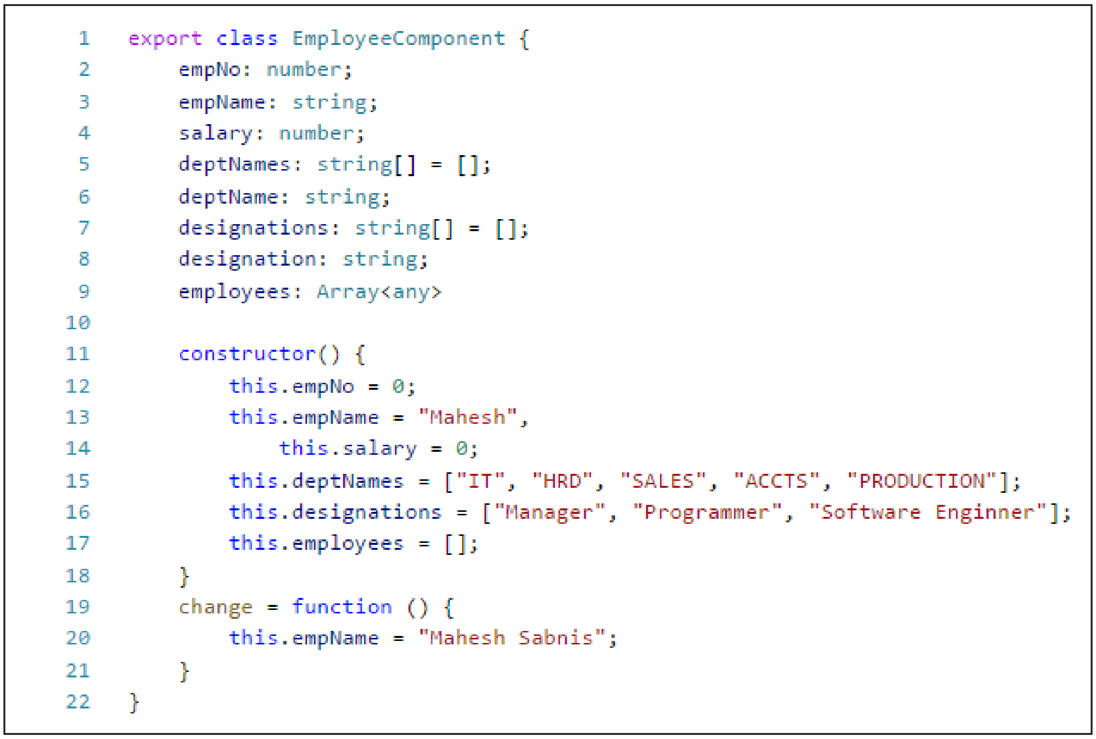
Listing 15.2: *The selector for defining HTML Tag*



```
@Component({
    selector: 'emp-data'
})
```

The **EmployeeComponent** class is defined with public properties in it for Employees.

Listing 15.3: *The EmployeeComponent class*

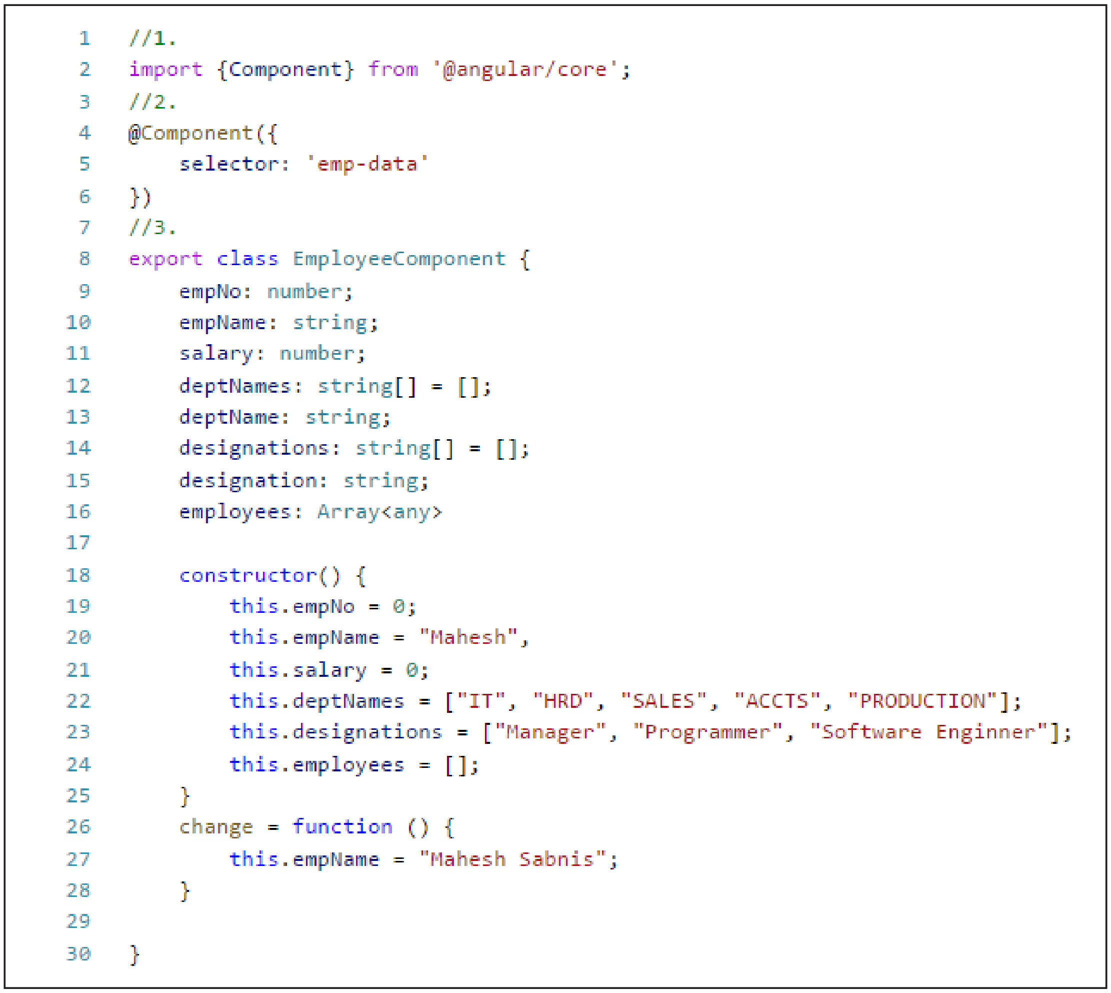


```
export class EmployeeComponent {
    empNo: number;
    empName: string;
    salary: number;
    deptNames: string[] = [];
    deptName: string;
    designations: string[] = [];
    designation: string;
    employees: Array<any>

    constructor() {
        this.empNo = 0;
        this.empName = "Mahesh",
            this.salary = 0;
        this.deptNames = ["IT", "HRD", "SALES", "ACCTS", "PRODUCTION"];
        this.designations = ["Manager", "Programmer", "Software Enginner"];
        this.employees = [];
    }
    change = function () {
        this.empName = "Mahesh Sabnis";
    }
}
```

The EmployeeComponent has the default value for EmpName as 'Mahesh'. The **change()** function is responsible for updating the EmpName value.

The entire EmployeeComponent is as shown in Listing 15-4:

Listing 15.4: *The complete code*

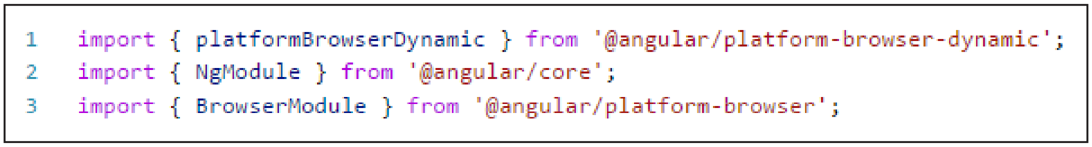


```
//1.
import {Component} from '@angular/core';
//2.
@Component({
    selector: 'emp-data'
})
//3.
export class EmployeeComponent {
    empNo: number;
    empName: string;
    salary: number;
    deptNames: string[] = [];
    deptName: string;
    designations: string[] = [];
    designation: string;
    employees: Array<any>

    constructor() {
        this.empNo = 0;
        this.empName = "Mahesh",
        this.salary = 0;
        this.deptNames = ["IT", "HRD", "SALES", "ACCTS", "PRODUCTION"];
        this.designations = ["Manager", "Programmer", "Software Enginner"];
        this.employees = [];
    }
    change = function () {
        this.empName = "Mahesh Sabnis";
    }

}
```

Once the component is created, it needs to be loaded using Angular 2's bootstrap function. This function is present in the module @angular/platform-browser-dynamic path.

In the main.ts file of the **app** folder, make sure that the modules shown in listing 15-5 are imported .

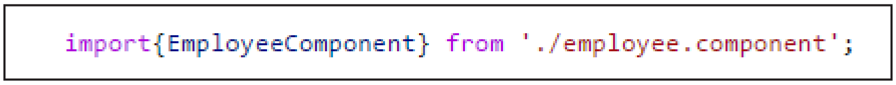
Listing 15.5: *Importing modules for bootstrapping*



```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
```

Now import the EmployeeComponent:

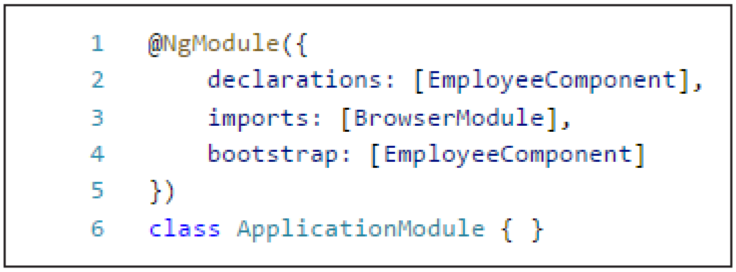
Listing 15.6: *Importing EmployeeComponent*



```
import{EmployeeComponent} from './employee.component';
```

Next the component EmployeeComponent has to be declared in the module as shown in Listing 15.7:

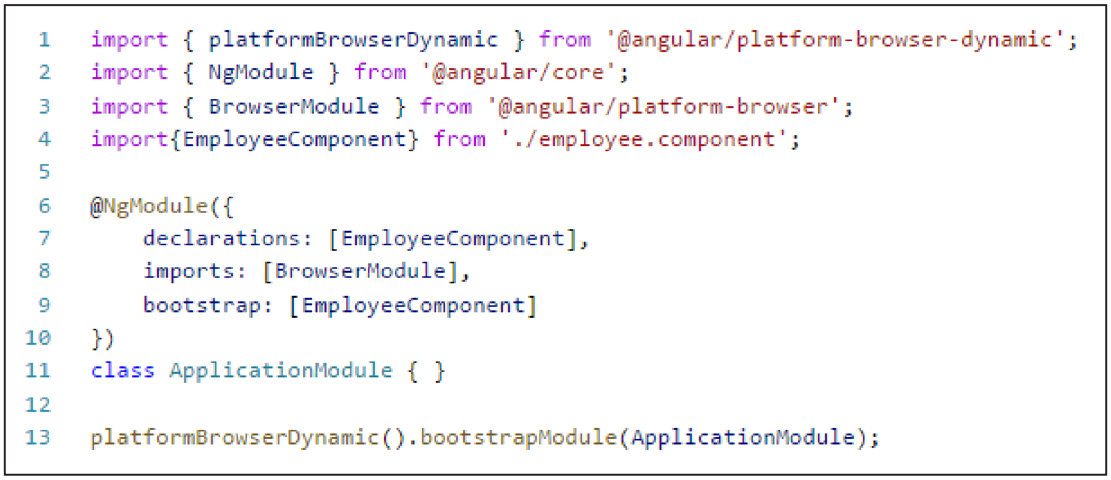
Listing 15.7: *Defining NgModule*



```
@NgModule({
    declarations: [EmployeeComponent],
    imports: [BrowserModule],
    bootstrap: [EmployeeComponent]
})
class ApplicationModule { }
```

The entire code in main.ts is shown in Listing 15-8:

Listing 15.8: *The Complete code of main.ts*



```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import{EmployeeComponent} from './employee.component';

@NgModule({
    declarations: [EmployeeComponent],
    imports: [BrowserModule],
    bootstrap: [EmployeeComponent]
})
class ApplicationModule { }

platformBrowserDynamic().bootstrapModule(ApplicationModule);
```

Once the code is complete, compile the code. Refer to Chapter 5—Angular Development Environment chapter in case you are not familiar with the process.

Now the application is **configured** with the required packages and the data needed has been provided to the component. Chapters 17 through 20 will use this data to explore different features of data binding.

Chapter 16

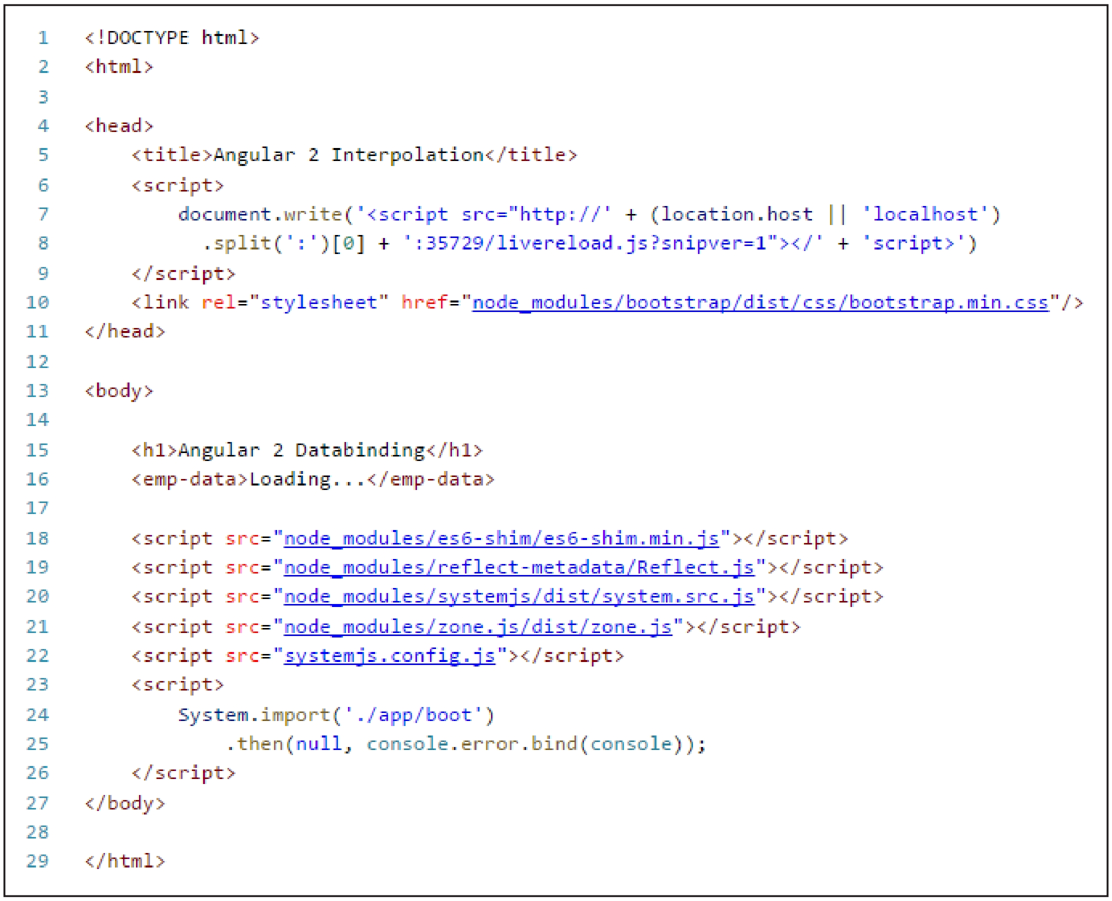
Angular 2 DataBinding Interpolation

IN CHAPTER 15, WE took an overview of Databinding and created a project with the necessary packages. This chapter will use the Data-Model Component named **EmployeeComponent** and the bootstrap code in **main.ts** for implementing **Interpolation**.

Angular 2 Interpolation (one of the biggest features of the framework) is implemented using double-curly braces '{{}}'. The Interpolation syntax is just an alternate syntax for property binding. When the Data-Model component's property is changed, then the UI element's property also gets changed; however, changes from the UI element do not update the Data-Model property. In such cases, Interpolation is used to update the UI element when any changes occur in the component's properties.

In the application folder of the previous chapter, add an HTML file and name it as Employee.html. In the index.html, make sure that the following code is present:

Listing 16.1: *The index.html*

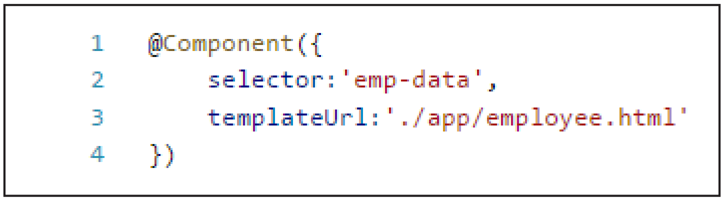


```
<!DOCTYPE html>
<html>

<head>
    <title>Angular 2 Interpolation</title>
    <script>
        document.write('<script src="http://' + (location.host || 'localhost')
          .split(':')[0] + ':35729/livereload.js?snipver=1"></' + 'script>')
    </script>
    <link rel="stylesheet" href="node_modules/bootstrap/dist/css/bootstrap.min.css"/>
</head>

<body>

    <h1>Angular 2 Databinding</h1>
    <emp-data>Loading...</emp-data>

    <script src="node_modules/es6-shim/es6-shim.min.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="systemjs.config.js"></script>
    <script>
        System.import('./app/boot')
            .then(null, console.error.bind(console));
    </script>
</body>

</html>
```

Listing 16.1 contains the required references for Angular 2. Make sure that the script references are in the same order as in Listing 16-1 (especially if you are running the app in Internet Explorer). The module of the application, which is main, is loaded first. The emp-data component starts the application. This component is defined in employee.component.ts file.

Modify the employee.component.ts file. Set the **templateUrl** property of Component to employee.html as shown in Listing 16-2.

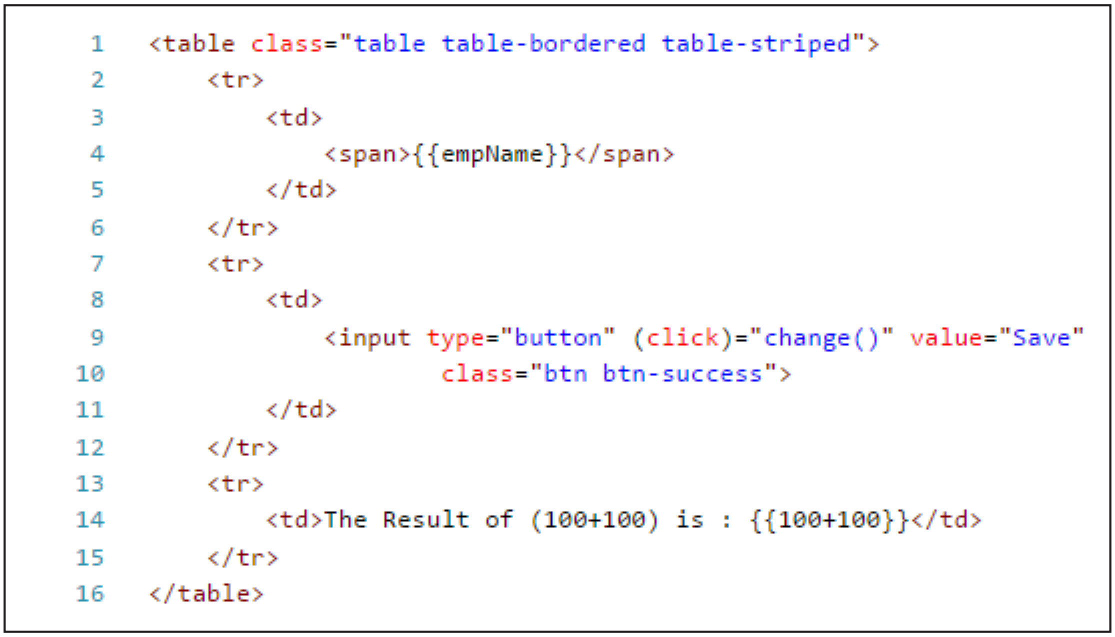
Listing 16.2: *Adding the templateUrl property*



```
@Component({
    selector:'emp-data',
    templateUrl:'./app/employee.html'
})
```

In employee.html, add the following markup.

Listing: *16-3: The code employee.html*



```
<table class="table table-bordered table-striped">
    <tr>
        <td>
            <span>{{empName}}</span>
        </td>
    </tr>
    <tr>
        <td>
            <input type="button" (click)="change()" value="Save"
                   class="btn btn-success">
        </td>
    </tr>
    <tr>
        <td>The Result of (100+100) is : {{100+100}}</td>
    </tr>
</table>
```

The html markup in Listing 16-3 contains the <span> element with the Component's property ***empName*** set in braces**.** Angular 2 will replace this interpolation with the property value from the Component. The expression in the curly braces is known as **template expression.** This is evaluated by Angular, and converted into a string. The markup in Listing 16-3 also contains mathematical expression which will be evaluated to its result.

Note the code also uses event binding `(click)="change()"` which will be discussed in detail in Chapter 18—Event binding chapter.

Running the Application

Right-click on index.html and select option **Open in Command Prompt,** to open the command prompt.

On this command prompt, run the following command:-

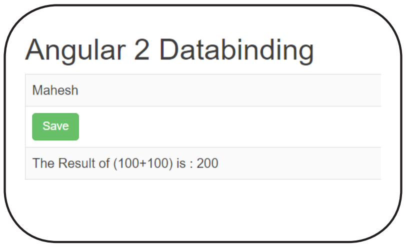
Listing 16.6: *Command to run the application*

```
npm run start
```

Open the browser and enter the following URL

```
http://localhost:3000
```

This loads the browser with the UI as shown in Figure 16-1.



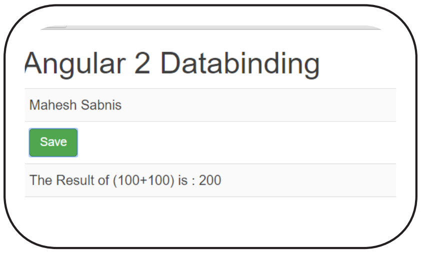
Angular 2 Databinding

Mahesh

The Result of (100+100) is : 200

Figure 16.1: *The First-Time running of the application*

The **span** shows the value of the empName property as 'Mahesh'. Click on the **Save** button and the empName property will be updated in the Data-Model to **Mahesh Sabnis**. The end result is shown in Figure 16-2:



Angular 2 Databinding

Mahesh Sabnis

The Result of (100+100) is : 200

Figure 16.2: *Result after clicking the Save button*

Conclusion

Interpolation is a special syntax that allows for a richer template HTML. Angular 2 evaluates the template expression in the interpolation braces, converts it into a string, and assigns the result to a HTML element or directive property.

Chapter

17

Angular 2 Property Binding

CHAPTER 15 PROVIDED AN overview of Databinding and demonstrated how to create a Data-Model. Chapter 16 showed Interpolation in action. This chapter will discuss about **Property Binding** in Angular 2.

A View consists of several HTML UI elements with various property attributes which can control the behavior of the UI Element. In Angular 2, Property Binding can be used to set the values of these attributes. In the case of **Property Binding,** we can bind attributes like **href, style**, etc. using values of properties from the Data-Model, especially in scenarios that do not *explicitly* want any value updates from View to Data-Model, and back. This also means that Property Binding is a type of One-Way data binding.

To demonstrate Property Binding, this chapter uses the same Data-Model from Chapter 15.

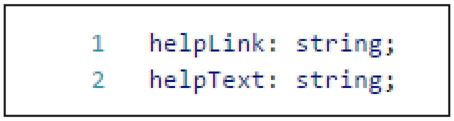
Open main.ts and for property binding to work, import the FormsModule from the @angular/forms package. The following listing shows the code for importing the FormsModule:

Listing 17.1

```
import { FormsModule } from '@angular/forms';
```

Open employee.component:

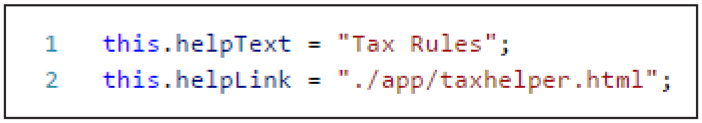
Listing 17.2



```
helpLink: string;
helpText: string;
```

In the constructor, set values for the properties as shown in Listing 17-2.

Listing 17.3

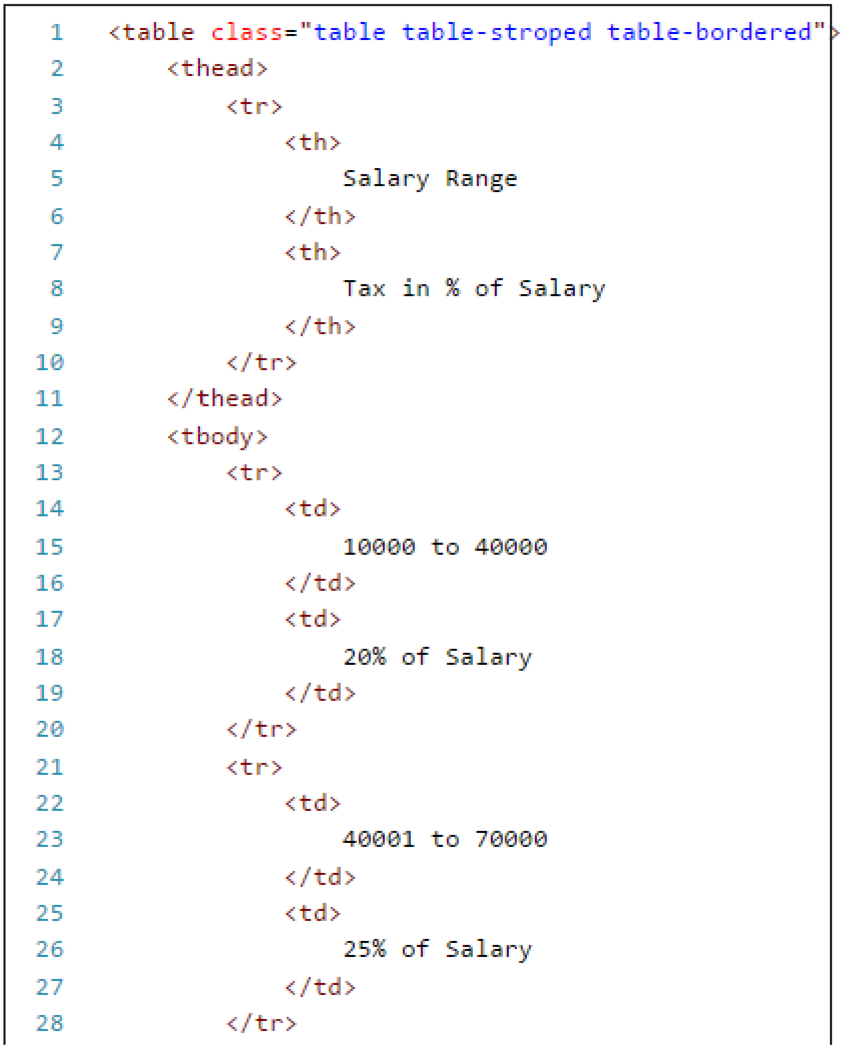


```
this.helpText = "Tax Rules";
this.helpLink = "./app/taxhelper.html";
```

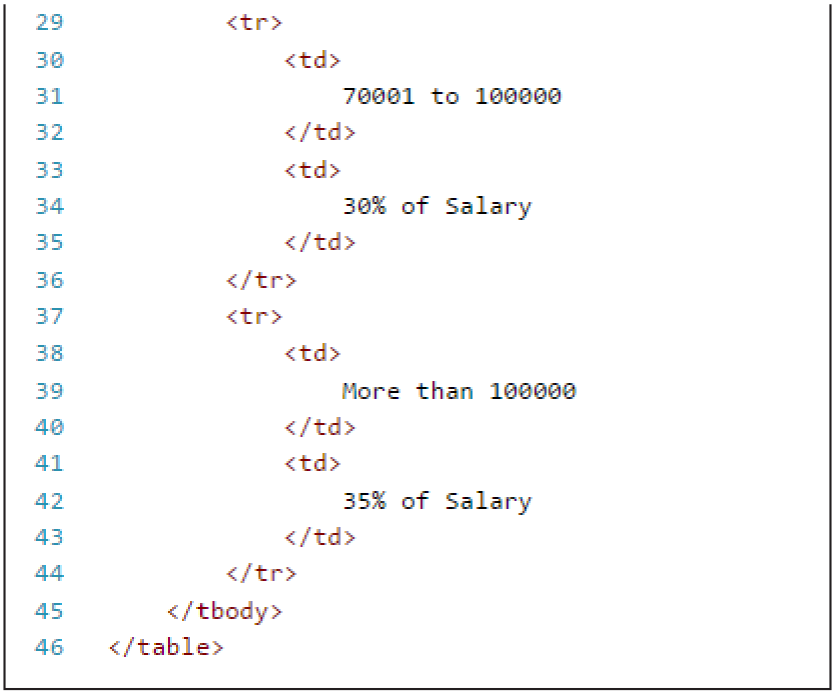
Here the values are set for **`helpText`** and **`helpLink,`** which will be used for property binding with HTML.

In the app folder of the project, add a new html file with the name taxhelper.html. This Html page will be used to display tax information for the salary entered for the employee.

Listing 17.4: *The Markup in taxhelper.html*



```
<table class="table table-stroped table-bordered">
    <thead>
        <tr>
            <th>
                Salary Range
            </th>
            <th>
                Tax in % of Salary
            </th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>
                10000 to 40000
            </td>
            <td>
                20% of Salary
            </td>
        </tr>
        <tr>
            <td>
                40001 to 70000
            </td>
            <td>
                25% of Salary
            </td>
        </tr>
```

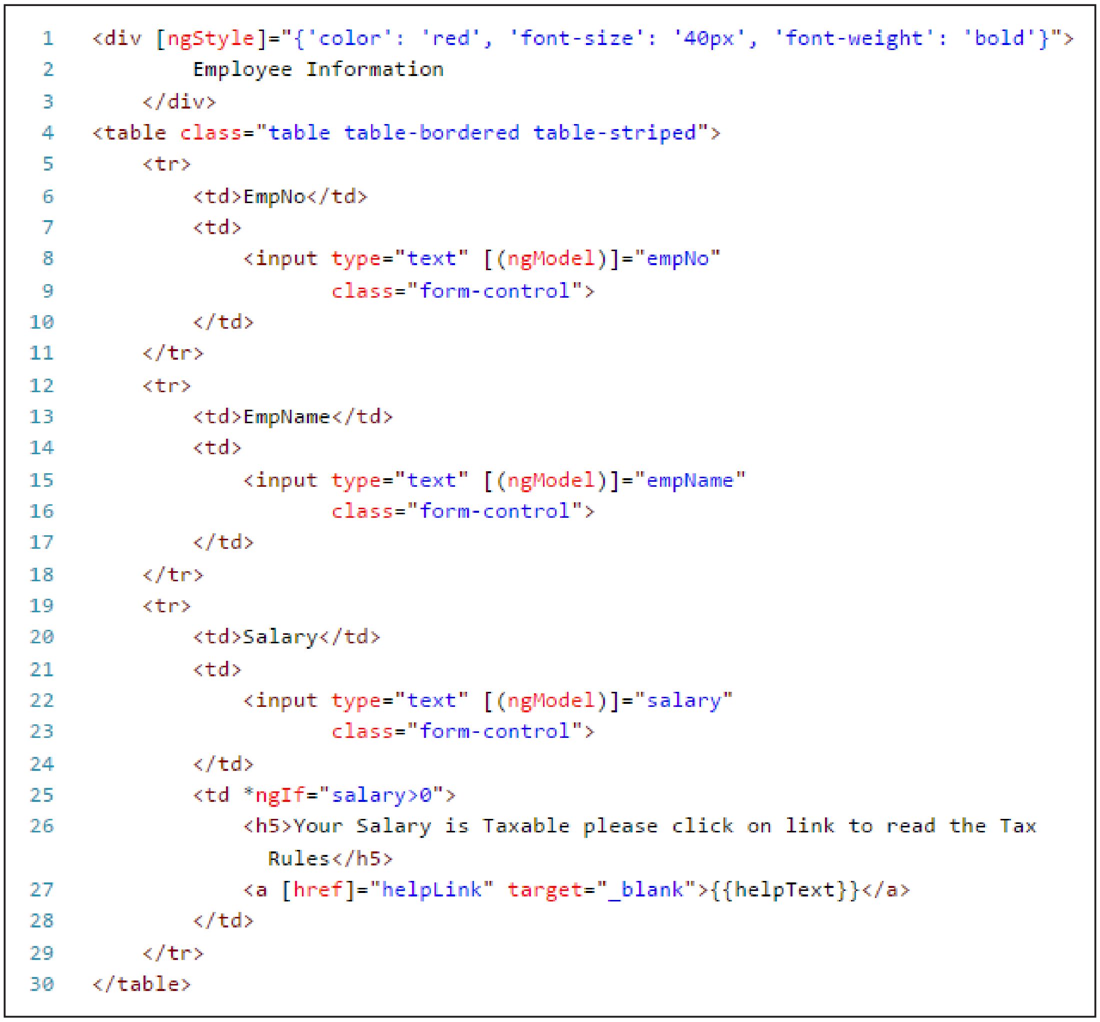


```
        <tr>
            <td>
                70001 to 100000
            </td>
            <td>
                30% of Salary
            </td>
        </tr>
        <tr>
            <td>
                More than 100000
            </td>
            <td>
                35% of Salary
            </td>
        </tr>
    </tbody>
</table>
```

The HTML page in Listing 17-4 will show the Tax for Salary ranges.

Modify the employee.html by adding a text Box for binding the property salary and a hyperlink for binding the **helpLink** properties declared in the EmployeeComponent class.

Listing 17.5: *The style in employee.html*



```
<div [ngStyle]="{'color': 'red', 'font-size': '40px', 'font-weight': 'bold'}">
        Employee Information
    </div>
<table class="table table-bordered table-striped">
    <tr>
        <td>EmpNo</td>
        <td>
            <input type="text" [(ngModel)]="empNo"
                   class="form-control">
        </td>
    </tr>
    <tr>
        <td>EmpName</td>
        <td>
            <input type="text" [(ngModel)]="empName"
                   class="form-control">
        </td>
    </tr>
    <tr>
        <td>Salary</td>
        <td>
            <input type="text" [(ngModel)]="salary"
                   class="form-control">
        </td>
        <td *ngIf="salary>0">
            <h5>Your Salary is Taxable please click on link to read the Tax
              Rules</h5>
            <a [href]="helpLink" target="_blank">{{helpText}}</a>
        </td>
    </tr>
</table>
```

In Listing 17-4, the <div> tag is bound to **ngStyle** for setting the CSS style for the text in it. The **href** property of the hyperlink element is bound to the **helpLink** property using property binding syntax **[href].**

NOTE: There is another syntax that can be used called bind-href <a bind-href="helpLink" target="_blank">{{helpText}}</a> but we will use the bracket [] syntax for the rest of this book.

The hyperlink will be visible if the salary entered is more than zero. Note that the code in Listing 17-5 uses **ngModel directive** which is a feature of the Two-Way binding, and will be discussed in a forthcoming chapter.

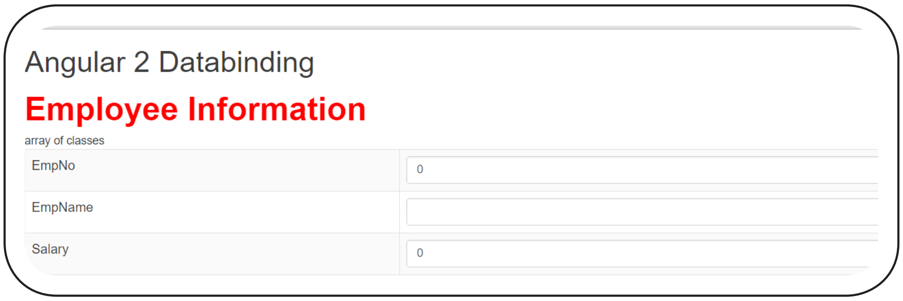
RUNNING THE APPLICATION

Right-click on index.html and select **Open in Command Prompt,** enter the following command in the command prompt:

Listing 17.6: *Command to run application*

```
npm run start
```

This command will show index.html in the browser as shown in Figure 17-1



Angular 2 Databinding

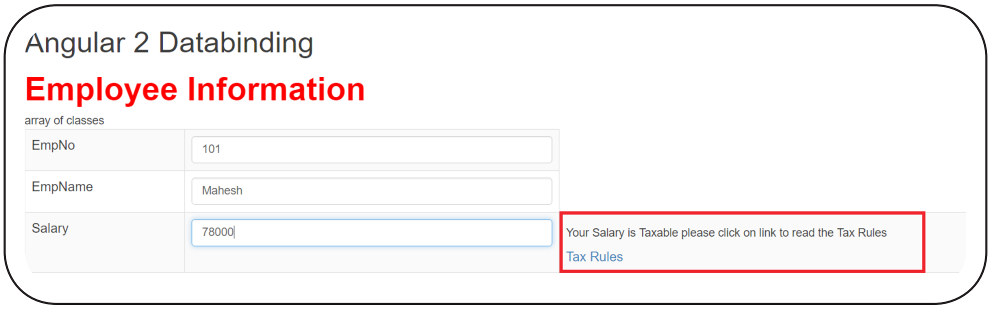
Employee Information

array of classes

EmpNo	0
EmpName	
Salary	0

Figure 17.1: *The first running the application*

Enter EmpNo, EmpName and Salary. If the Salary is greater than 0, the Link will be visible as shown in Figure 17-2.



Angular 2 Databinding

Employee Information

array of classes

EmpNo	101	
EmpName	Mahesh	
Salary	78000	Your Salary is Taxable please click on link to read the Tax Rules Tax Rules

Figure 17.2: *The property binding execution*

Once the link is visible, click on it and the page TaxHelper.html will be opened in a separate browser tab.

The **href** property of the HTML Hyperlink element has been successfully bound with the **helpLink** property from the Data-Model.

Conclusion

Property Binding can be used in cases of binding attributes of Html elements, in order to control its behavior during runtime.

Chapter 18

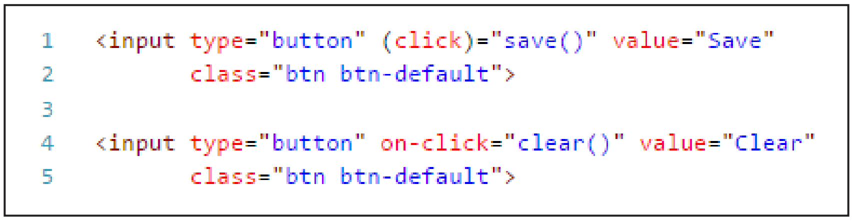
Angular 2 Event Binding

CHAPTERS 15 THROUGH 17 focused on various features of Angular 2 Databinding. Databinding is one of the most important features of Angular 2 and is heavily used in Line-of-Business (LOB) applications. The Data-Model properties are bound with the HTML UI Elements for data display and updates.

The Data-Model may have functions containing code for business rules and for processing the property values. These functions are executed using the events of HTML elements e.g. Button Click, TextBox Blur, keypress, keyup etc. These events execute a function defined in the component as a response to the action the end-user takes on the UI element, e.g. clicking on the button or pressing a key on the keyboard

The event-binding features in Angular 2 can be used to bind the functions defined in the Data-Model with the events exposed by HTML elements. The event uses parenthesis ( ) notation in HTML. The syntax for event binding is shown in Listing 18-1:

Listing 18.1: *Adding event binding in employee.html*



```
<input type="button" (click)="save()" value="Save"
       class="btn btn-default">

<input type="button" on-click="clear()" value="Clear"
       class="btn btn-default">
```

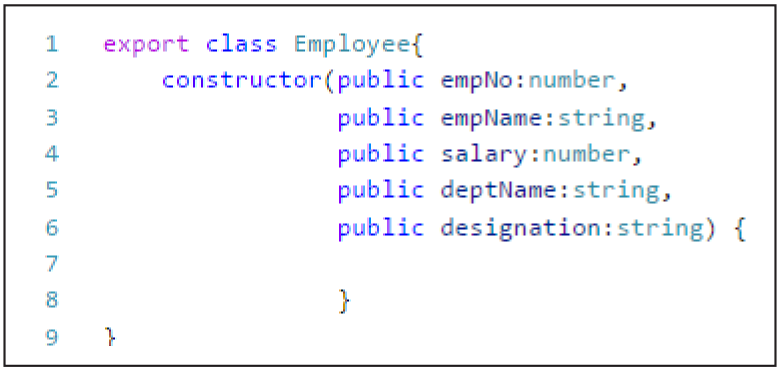
There are two syntaxes to add event binding. The event of the HTML element is enclosed in the parentheses with the function name passed to it. For example, in the above listing, the **click** event of the button is bound to the **save()** function. Alternatively, we can also use the **'on-'** prefix for the click event, as used for binding the **clear()** function in the Clear button.

The Implementation

This chapter will modify the Data-Model used in Chapter 17 by creating a separate Employee class with all of the properties in it.

In the project, add a new file of the **employee.model.ts.** Create an Employee class in this file as shown in listing 18-2.

Listing 18.2: *The Employee model class*

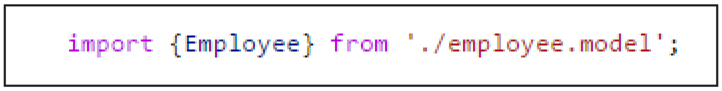


```
export class Employee{
    constructor(public empNo:number,
                public empName:string,
                public salary:number,
                public deptName:string,
                public designation:string) {

                }
}
```

The class contains constructor with public properties for storing Employee information.

Since a separate Employee class now exists, create an instance of the Employee class in the EmployeeComponent class. Modify the employee.component.ts file as shown in the listing 18-3.

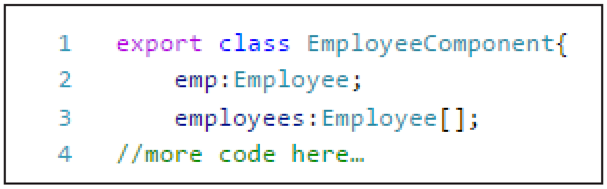
Listing 18.3: *Importing Employee Model in employee.component.ts*



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

This statement will use Employee class as a **type** to declare the Employee property in the EmployeeComponent class in employee.component.ts, as shown in listing 18-4.

Listing 18.4: *Declaring instance of Employee model class in EmployeeComponent class*

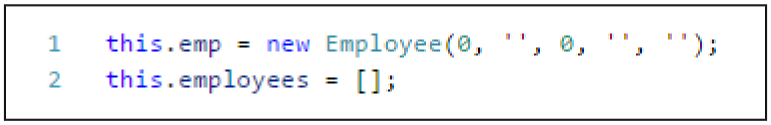


```
export class EmployeeComponent{
    emp:Employee;
    employees:Employee[];
//more code here…
```

As shown in Listing 18.4, the **emp** property and **employees** array of type **employee** have been declared.

Now instantiate the **emp** object and **employees** array in the constructor as shown in listing 18.5:

Listing 18.5: *Initializing the Employee Model object and Array declaration*

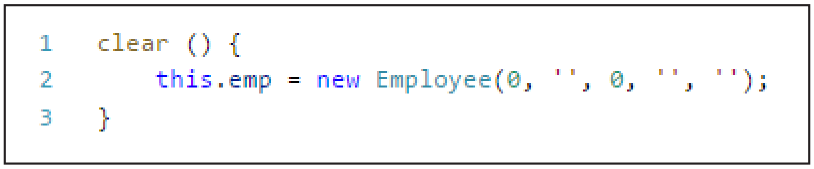


```
this.emp = new Employee(0, '', 0, '', '');
this.employees = [];
```

The **emp** instance will set values for public properties of the Employee class.

Add a function in the **EmployeeComponent** class to reset the Employee instance.

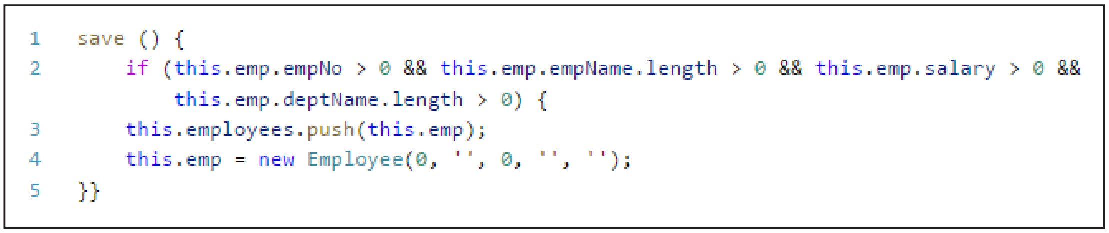
Listing 18.6: *Clearing the Employee object*



```
clear () {
    this.emp = new Employee(0, '', 0, '', '');
}
```

Create a function to **push** the Employee instance in the employees array.

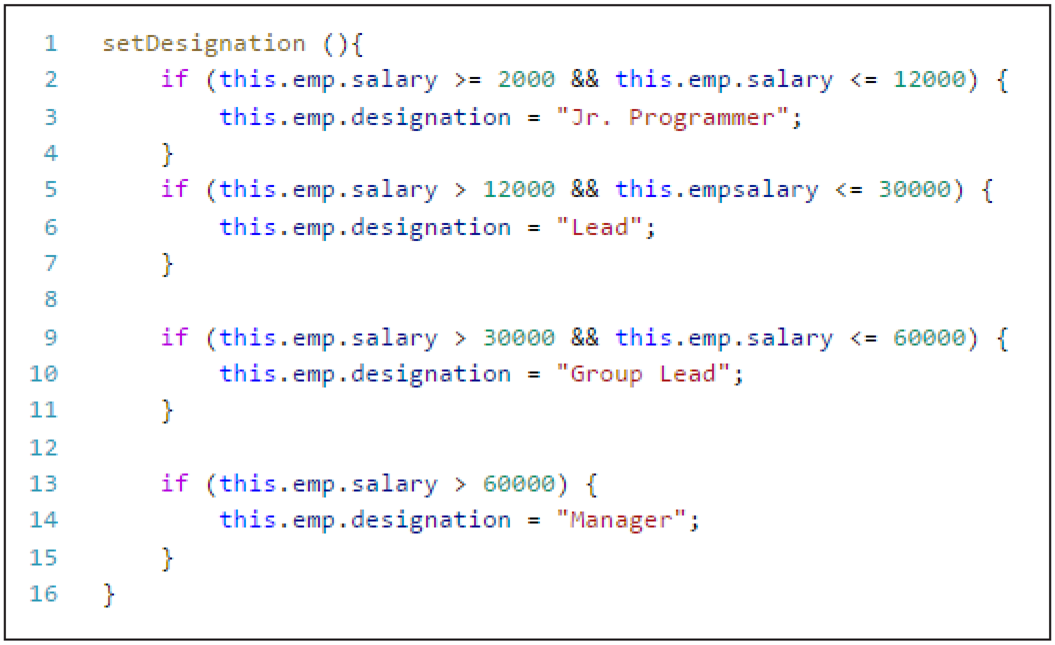
Listing 18.7: *Adding the Employee object in Employees array and clearing the Employee object*



```
save () {
    if (this.emp.empNo > 0 && this.emp.empName.length > 0 && this.emp.salary > 0 && this.emp.deptName.length > 0) {
    this.employees.push(this.emp);
    this.emp = new Employee(0, '', 0, '', '');
}}
```

Now add a function in the EmployeeComponent class to set the designation of the Employee based on the Salary entered for the Employee.

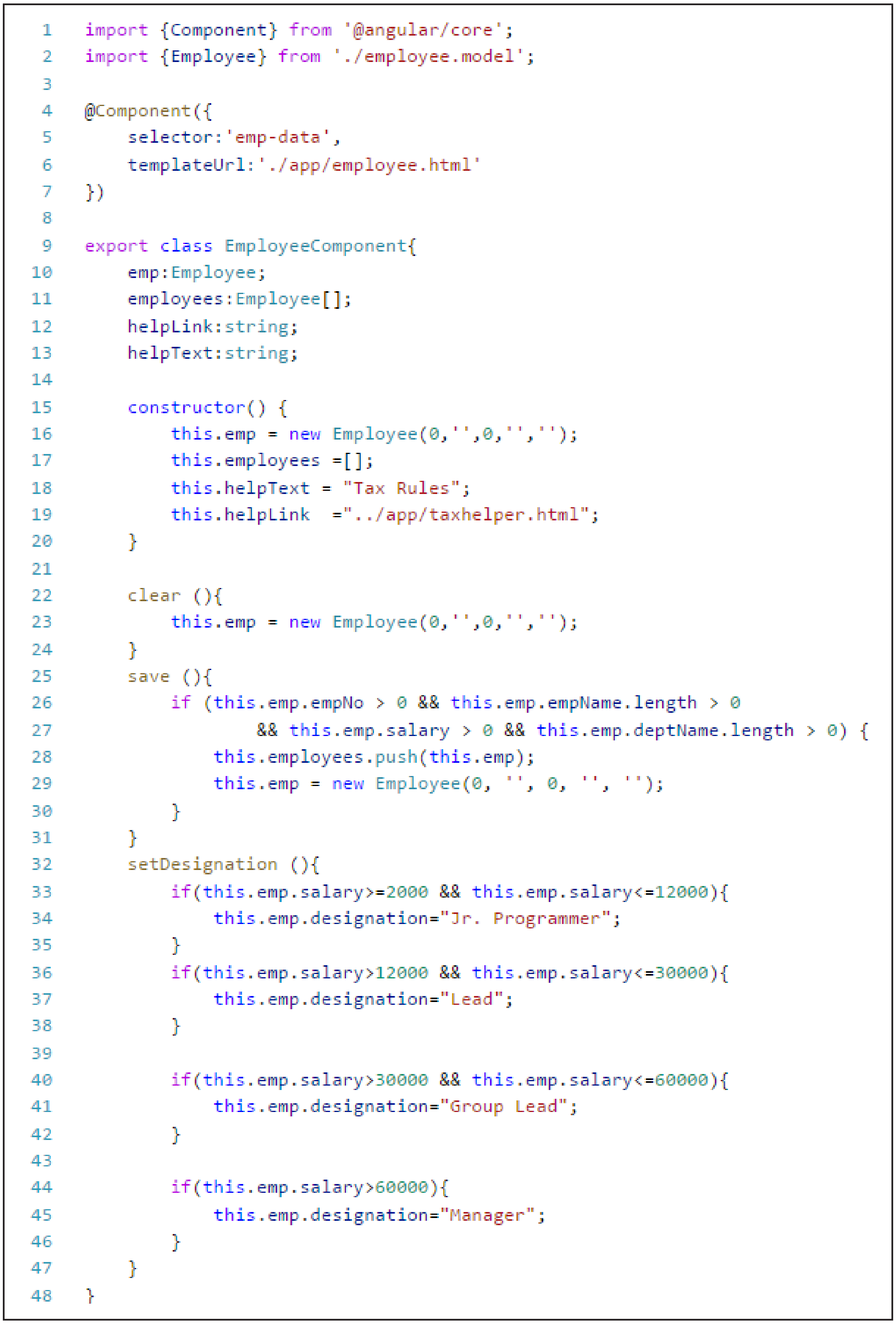
Listing 18.8: *Setting value of the designation based on salary*



```
setDesignation (){
    if (this.emp.salary >= 2000 && this.emp.salary <= 12000) {
        this.emp.designation = "Jr. Programmer";
    }
    if (this.emp.salary > 12000 && this.empsalary <= 30000) {
        this.emp.designation = "Lead";
    }

    if (this.emp.salary > 30000 && this.emp.salary <= 60000) {
        this.emp.designation = "Group Lead";
    }

    if (this.emp.salary > 60000) {
        this.emp.designation = "Manager";
    }
}
```

The entire code of the employee.component.ts is as shown in listing 18.9.

Listing 18.9: *The complete code of employee.component.ts*

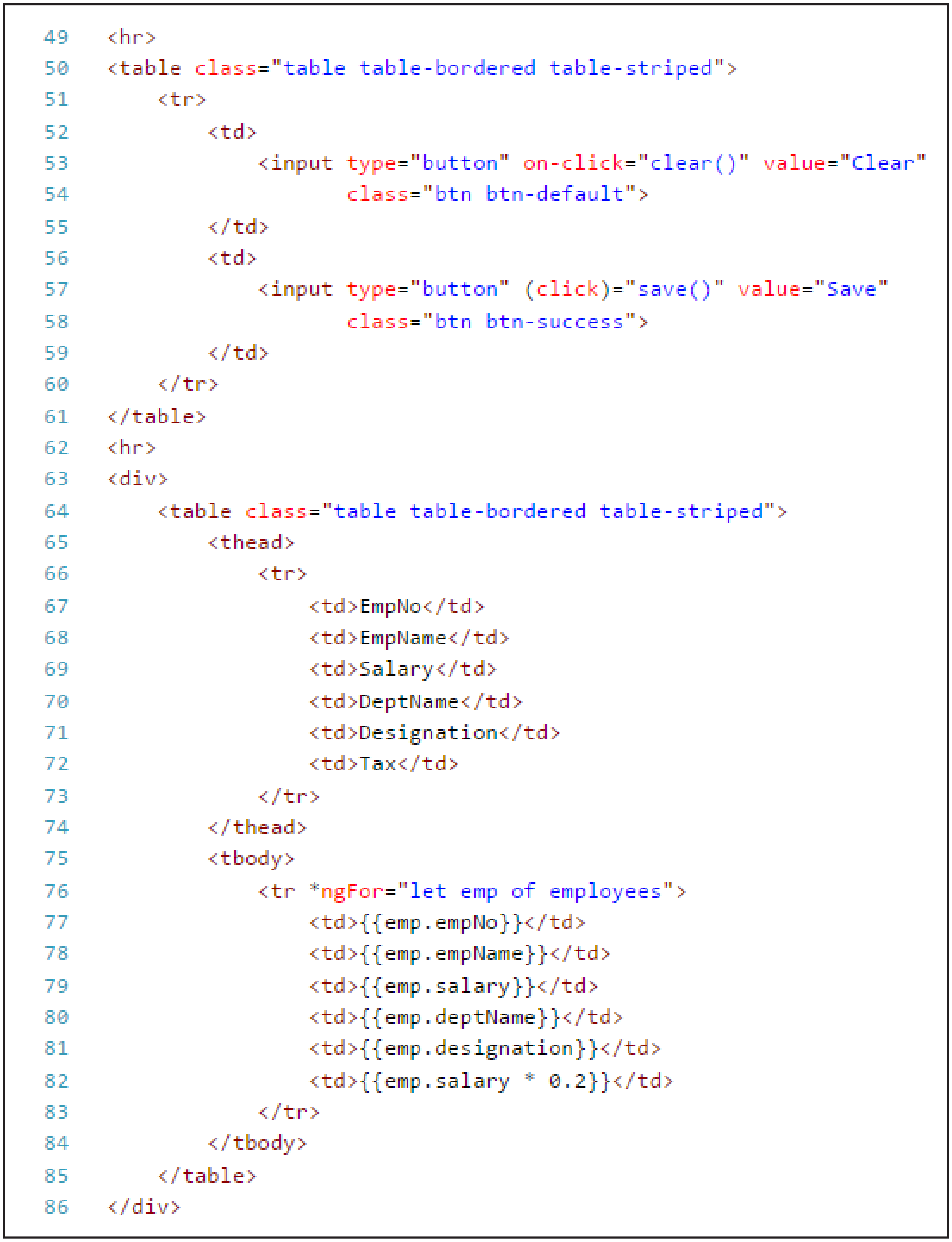


```
import {Component} from '@angular/core';
import {Employee} from './employee.model';

@Component({
    selector:'emp-data',
    templateUrl:'./app/employee.html'
})

export class EmployeeComponent{
    emp:Employee;
    employees:Employee[];
    helpLink:string;
    helpText:string;

    constructor() {
        this.emp = new Employee(0,'',0,'','');
        this.employees =[];
        this.helpText = "Tax Rules";
        this.helpLink  ="../app/taxhelper.html";
    }

    clear (){
        this.emp = new Employee(0,'',0,'','');
    }
    save (){
        if (this.emp.empNo > 0 && this.emp.empName.length > 0
                && this.emp.salary > 0 && this.emp.deptName.length > 0) {
            this.employees.push(this.emp);
            this.emp = new Employee(0, '', 0, '', '');
        }
    }
    setDesignation (){
        if(this.emp.salary>=2000 && this.emp.salary<=12000){
            this.emp.designation="Jr. Programmer";
        }
        if(this.emp.salary>12000 && this.emp.salary<=30000){
            this.emp.designation="Lead";
        }

        if(this.emp.salary>30000 && this.emp.salary<=60000){
            this.emp.designation="Group Lead";
        }

        if(this.emp.salary>60000){
            this.emp.designation="Manager";
        }
    }
}
```

Similar to the click event, other events like ‘keyup’, ‘keupress’etc. can be used for binding and to execute functionality defined in the component using the following syntax.

Modify the Employee.html to use the event binding as shown in Listing 18.10

Listing 18.10: *The employee.html code*



```
<hr>
<table class="table table-bordered table-striped">
    <tr>
        <td>
            <input type="button" on-click="clear()" value="Clear"
                   class="btn btn-default">
        </td>
        <td>
            <input type="button" (click)="save()" value="Save"
                   class="btn btn-success">
        </td>
    </tr>
</table>
<hr>
<div>
    <table class="table table-bordered table-striped">
        <thead>
            <tr>
                <td>EmpNo</td>
                <td>EmpName</td>
                <td>Salary</td>
                <td>DeptName</td>
                <td>Designation</td>
                <td>Tax</td>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor="let emp of employees">
                <td>{{emp.empNo}}</td>
                <td>{{emp.empName}}</td>
                <td>{{emp.salary}}</td>
                <td>{{emp.deptName}}</td>
                <td>{{emp.designation}}</td>
                <td>{{emp.salary * 0.2}}</td>
            </tr>
        </tbody>
    </table>
</div>
```

In listing 18.10, the Salary Textbox is bound with the **setDesignation()** function using the *blur* event of the Textbox. The *Clear* and *Save* buttons are bound with the *clear* and *save* functions using a *click* event. A table is used to display all the employees saved in the Employees array, when the *Save* button is clicked.

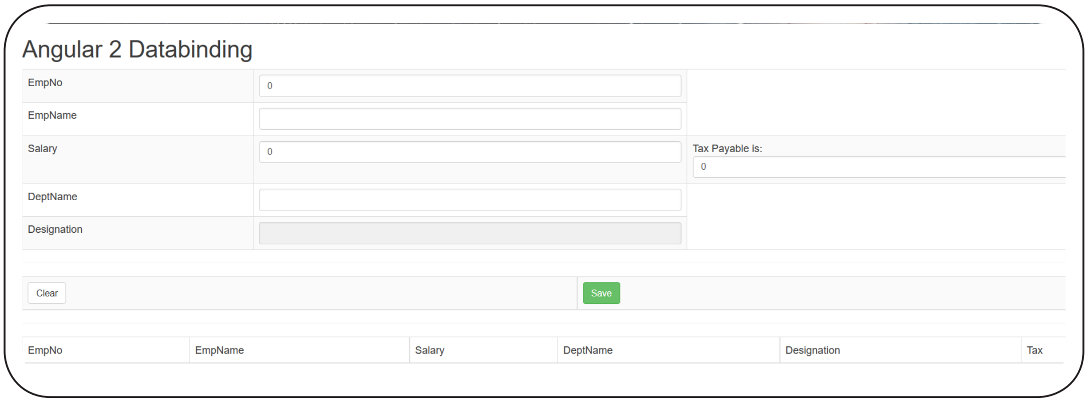
Running the application

In Visual Studio Code, right-click on index.html and select **Open in Command Prompt.** Run the following command from the command prompt

Listing 18.11: *The Command to execute the application*

```
npm run start
```

This will open the browser with index.html in it, as shown in Figure 18.1



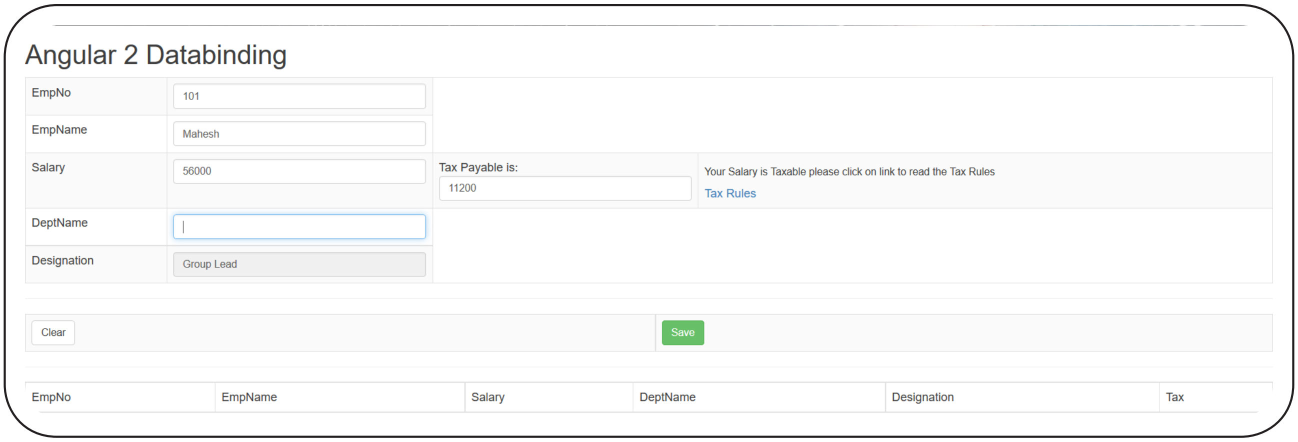
Angular 2 Databinding

EmpNo	0
EmpName	
Salary	0
Tax Payable is:	0
DeptName	
Designation	

EmpNo	EmpName	Salary	DeptName	Designation	Tax

Figure 18.1: *Loading the application*

Enter data for the Employee. On entering the Salary, the Tax will be calculated based on the Salary entered. Focus the cursor out of the Salary TextBox, and the Designation TextBox will show the respective Designation as seen in Figure 18.2:



Angular 2 Databinding

EmpNo	101
EmpName	Mahesh
Salary	56000
Tax Payable is:	11200
DeptName	
Designation	Group Lead

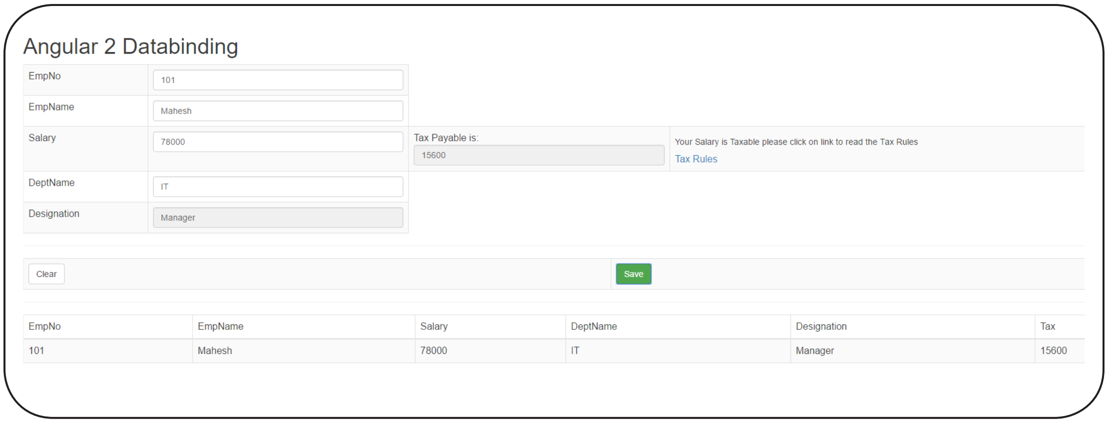
Your Salary is Taxable please click on link to read the Tax Rules

Tax Rules

EmpNo	EmpName	Salary	DeptName	Designation	Tax

Figure 18.2: *Generating Designation based on the Salary*

Enter the value for the DeptName and click on the **Save** button. The table at the bottom of the page will display the corresponding Employee entered. See Figure 18.3.



Angular 2 Databinding

EmpNo	101		
EmpName	Mahesh		
Salary	78000	Tax Payable is: 15600	Your Salary is Taxable please click on link to read the Tax Rules Tax Rules
DeptName	IT		
Designation	Manager		

EmpNo	EmpName	Salary	DeptName	Designation	Tax
101	Mahesh	78000	IT	Manager	15600

***Figure 18.3:** Showing inserted data in table*

Conclusion

Event Binding is used to perform an action in the component when an event like click, focus, or blur occurs in the view. This chapter demonstrated Angular 2's new Event Binding syntax with TypeScript and how to bind to DOM events.

Chapter

19

Angular 2: Two-Way Databinding

CHAPTERS 15 AND 16 provided an overview of Databinding and Interpolation. This Chapter will use the same data model created in Chapter 15 to implement Two-Way databinding in Angular 2.

Two-Way Databinding is one of the most useful feature in a Line-of-Business (LOB) Application. This feature reduces the need for writing additional code for updating the UI every time the Data-Model property value changes, and vice versa**. Angular 2 does not support two-way binding directly**. However, two-way binding in Angular 2 can be derived using property and event bindings.

Two-Way Databinding in Angular 2 is implemented using the **ngModel** directive and using the canonical prefix as **bindon-ngModel**.

THE IMPLEMENTATION

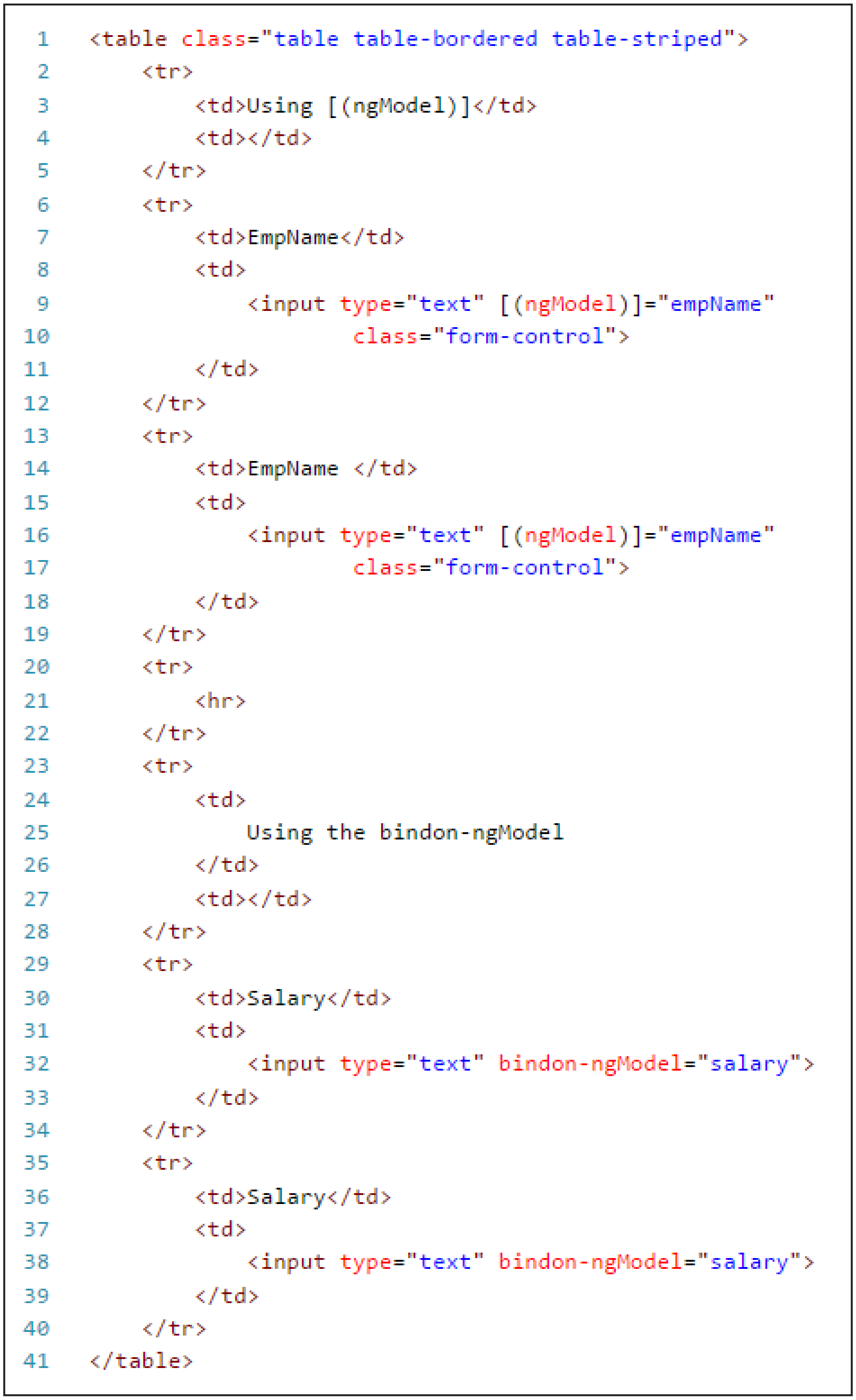
Open the project created in Chapter 16—Interpolation using Visual Studio Code. To implement Two-Way DataBinding in Angular 2, install the @angular/forms package. Open the package.json file and add the @angular/forms package in the dependencies as shown in listing 19.1:

Listing 19.1: *package.json*

```
"@angular/common": "2.0.0",

"@angular/compiler": "2.0.0",

"@angular/core": "2.0.0",

"@angular/http": "2.0.0",

"@angular/forms":"2.0.0",

"@angular/platform-browser": "2.0.0",

"@angular/platform-browser-dynamic": "2.0.0",
```

Change the employee.html file to the one in listing 19.2:

Listing 19.2: *employee.html*

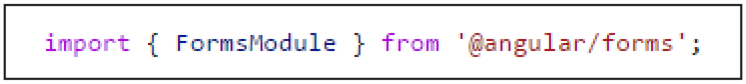


```
<table class="table table-bordered table-striped">
    <tr>
        <td>Using [(ngModel)]</td>
        <td></td>
    </tr>
    <tr>
        <td>EmpName</td>
        <td>
            <input type="text" [(ngModel)]="empName"
                    class="form-control">
        </td>
    </tr>
    <tr>
        <td>EmpName </td>
        <td>
            <input type="text" [(ngModel)]="empName"
                    class="form-control">
        </td>
    </tr>
    <tr>
        <hr>
    </tr>
    <tr>
        <td>
            Using the bindon-ngModel
        </td>
        <td></td>
    </tr>
    <tr>
        <td>Salary</td>
        <td>
            <input type="text" bindon-ngModel="salary">
        </td>
    </tr>
    <tr>
        <td>Salary</td>
        <td>
            <input type="text" bindon-ngModel="salary">
        </td>
    </tr>
</table>
```

Here, Two-Way Databinding is implemented using the *[(ngModel)]* directive set for the <input> element, as well as the attribute in canonical prefix *bindon-ngModel*. The [(ngModel)] directive can be used for elements like TextBox, ListBox, etc. It works by first accepting the value of the property to bind and then emitting an event when value of the model is updated in the view. This event is responsible to update the property of the data model. In listing 19-1, both input elements are bound with the *empName* property of the component EmployeeComponent.

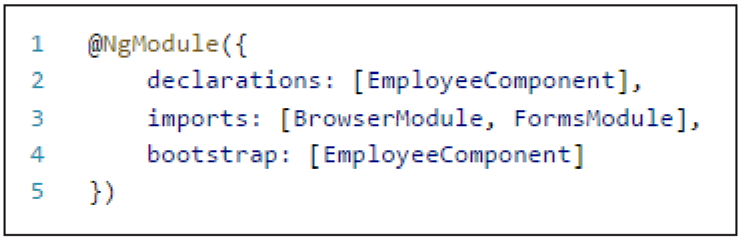
Modify the main.ts file to import FormsModule, which will be used for DataBinding. Modify the main.ts as shown in Listing 19.3

Listing 19.3: *The main.ts modification to import Forms module*



```
import { FormsModule } from '@angular/forms';
```

Listing 19.4: *The main.ts modification to import FormsModule in @NgModule*



```
@NgModule({
    declarations: [EmployeeComponent],
    imports: [BrowserModule, FormsModule],
    bootstrap: [EmployeeComponent]
})
```

The FormsModule provides an access to the form ngModel which is used for Databinding.

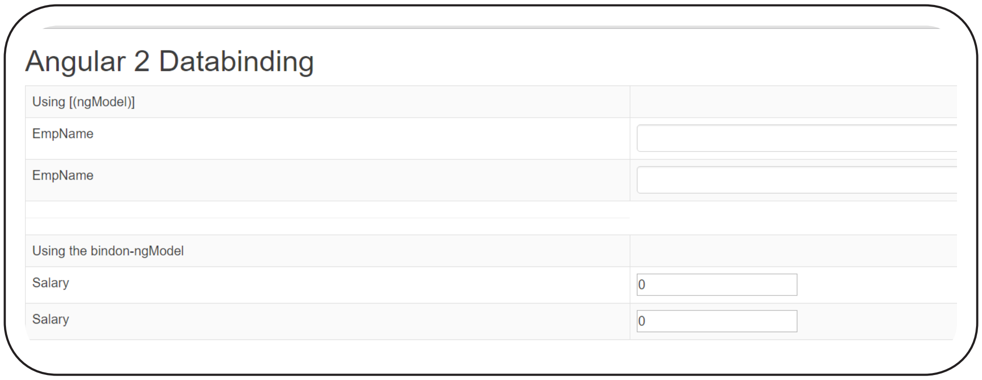
RUNNING THE APPLICATION

Right-click on Index.html > Open in Command Prompt. Run the following command from the Command Prompt.

Listing 19.5

```
npm run start
```

This will display the HTML page shown in Figure 19-1

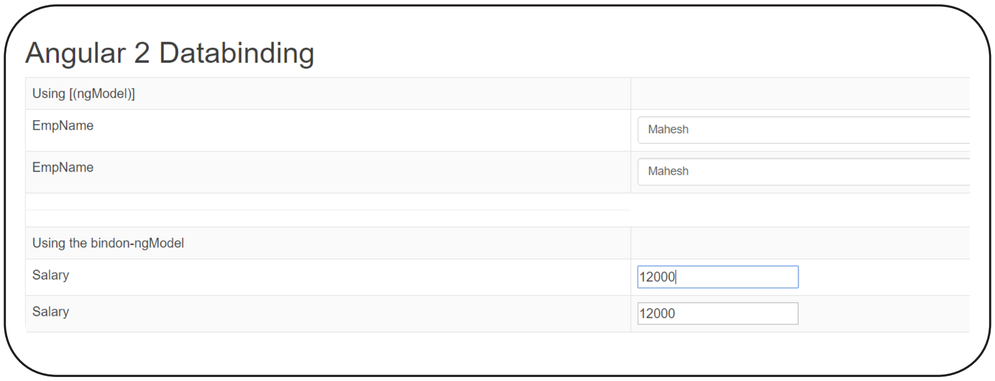


Angular 2 Databinding

Using [(ngModel)]	
EmpName	
EmpName	
Using the bindon-ngModel	
Salary	0
Salary	0

Figure 19.1: *Loading the page in browser*

Enter the EmpName in the first EmpName TextBox, and Salary in the first Salary TextBox. The second Textbox for EmpName and Salary will show the same value as in the first TextBox. See Figure 19-2:

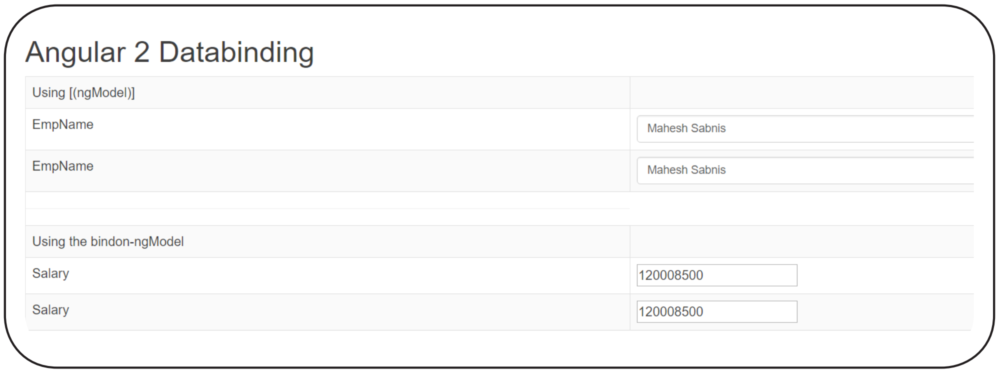


Angular 2 Databinding

Using [(ngModel)]	
EmpName	Mahesh
EmpName	Mahesh
Using the bindon-ngModel	
Salary	12000
Salary	12000

Figure 19.2: *Updating EmpName and Salary (First TextBox EmpName and Salary)*

Now enter some data in the second TextBox for both EmpName and Salary. This will update the respective TextBoxes as shown in Figure 19-3:



Angular 2 Databinding

Using [(ngModel)]	
EmpName	Mahesh Sabnis
EmpName	Mahesh Sabnis
Using the bindon-ngModel	
Salary	120008500
Salary	120008500

Figure 19.3: *Updating EmpName and Salary (Second TextBox EmpName and Salary)*

HOW DOES IT WORK?

Angular 2 provides the **ngModel** directive and the **ngModel** input property sets the element's value property. The **ngModel** detects the change event on the input element and updates the model property bound to the element. The **ngModelChange** output property also listens to any changes to the value property of the UI element.

Conclusion

Two-way data binding combines the input and output binding into a single notation using the ngModel directive.

Chapter 20

Working with Forms

THIS CHAPTER PROVIDES A basics understanding of Angular 2 forms. In Angular 2 applications (and, hence, in Web Applications), the Form plays a crucial role while developing a UI, as forms provide a way for the users to interact with the application. This chapter explains how to create and use Forms. Before implementing the application, let's go over some basics of an Angular 2 Form.

Angular 2 introduces the **ngForm** directive. This directive supplements the **form** element with features like form submit action, which is used to hold the controls in the form having **ngModel** directive applied to them. The **name** attribute is used so that their properties can be monitored for performing validations.

Angular 2 internally creates **FormControls**. This corresponds to the **name** attribute we set for the input element. The **name** attribute represents the relationship between an element in the html document, with its container form tag. The <form> tag manages the control's state, e.g. during validation using its **name** attribute**.**

One more important point to note is that Angular 2 uses **ngModel** with ngModelOptions to link a reference to the model's property. This is used to indicate that a form control is standalone, which allows developers to distinguish that some controls shouldn't be registered with the parent form if necessary. **ngForm** provides **onSubmit()** function which, outputs **ngSubmit** object. This object is used to submit form values.

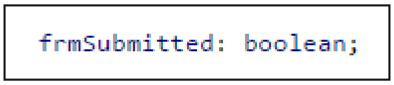
The ngForm directive gets applied to every <form> element present in the template of an Angular 2 component. To implement Angular 2 form, FormsModule is needed. This module class is provided in the @angular/core package. This module needs to be imported in the main.ts file.

The Implementation

To implement the code for Angular 2 forms, this chapter uses the code from Chapter 18 and modifies the **EmployeeComponent** class in **employee.component.ts** file.

The class declares a new Boolean property to check if the form is submitted in the class as shown in Listing 20-1.

Listing 20.1: *The Boolean flag for form submission*

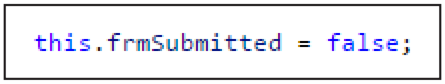


```
frmSubmitted: boolean;
```

This property sets the state of UI elements like <div> and <table> to make them visible or hidden.

This property is initialized in the constructor of the EmployeeComponent constructor as shown in Listing 20.2.

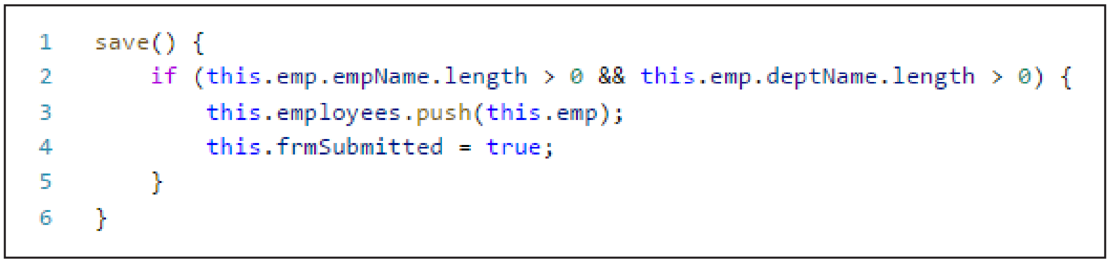
Listing 20.2: *The default value for flag*



```
this.frmSubmitted = false;
```

The **save()** function is modified as shown in Listing 2.03

Listing 20.3: *save() function used in form submission*

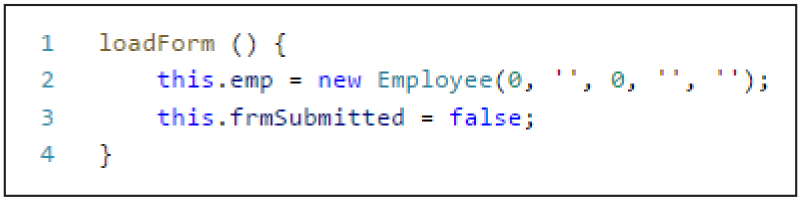


```
save() {
    if (this.emp.empName.length > 0 && this.emp.deptName.length > 0) {
        this.employees.push(this.emp);
        this.frmSubmitted = true;
    }
}
```

The above code sets the value for the *frmSubmitted* property to **true**. The **save()** function is accessed using the submit button.

Add the following function in the class to load the form with empty input elements:

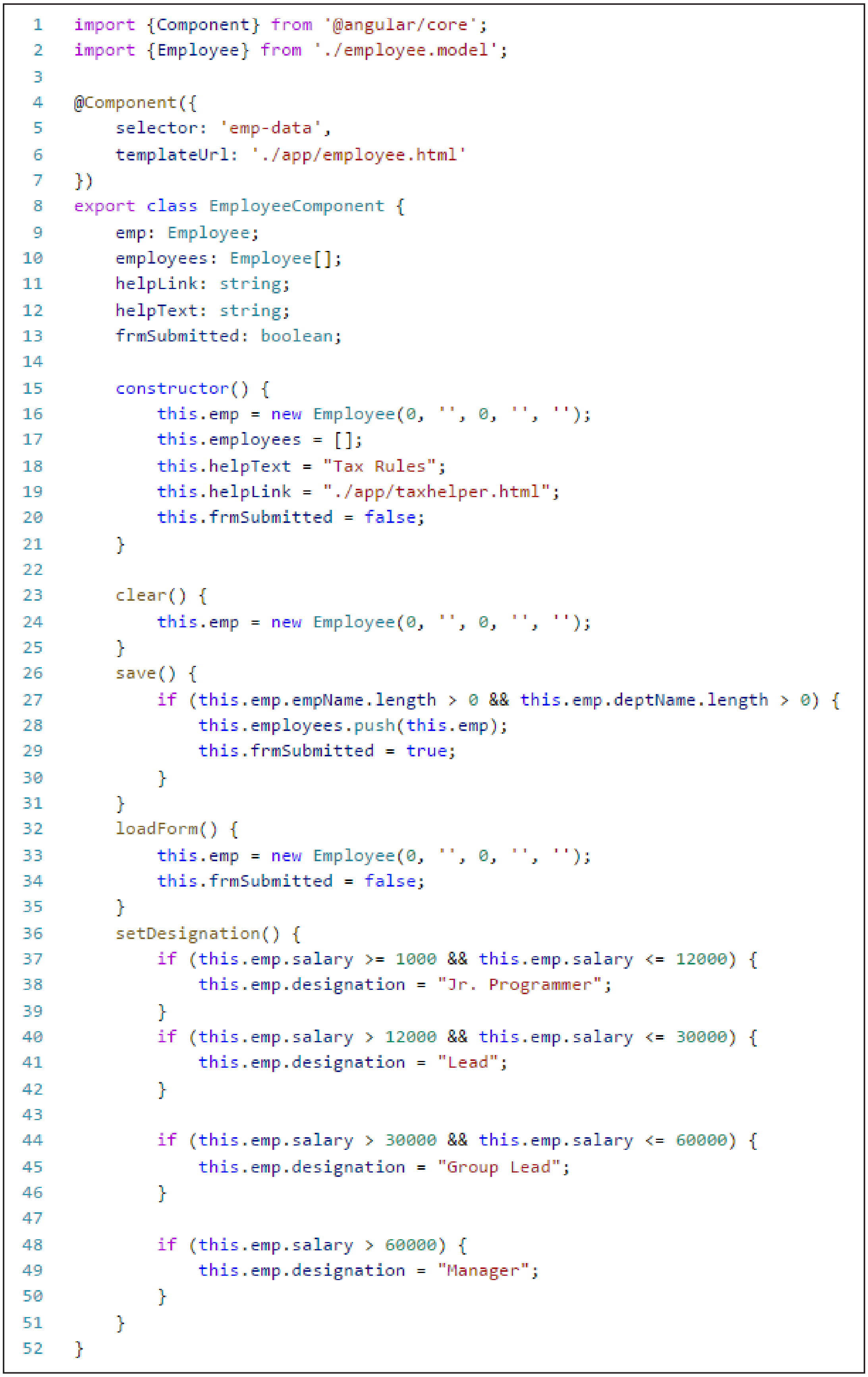
Listing 20.4: *Load form with empty elements*



```
loadForm () {
    this.emp = new Employee(0, '', 0, '', '');
    this.frmSubmitted = false;
}
```

Listing 20.5 shows the complete code for the employee.component.ts file:

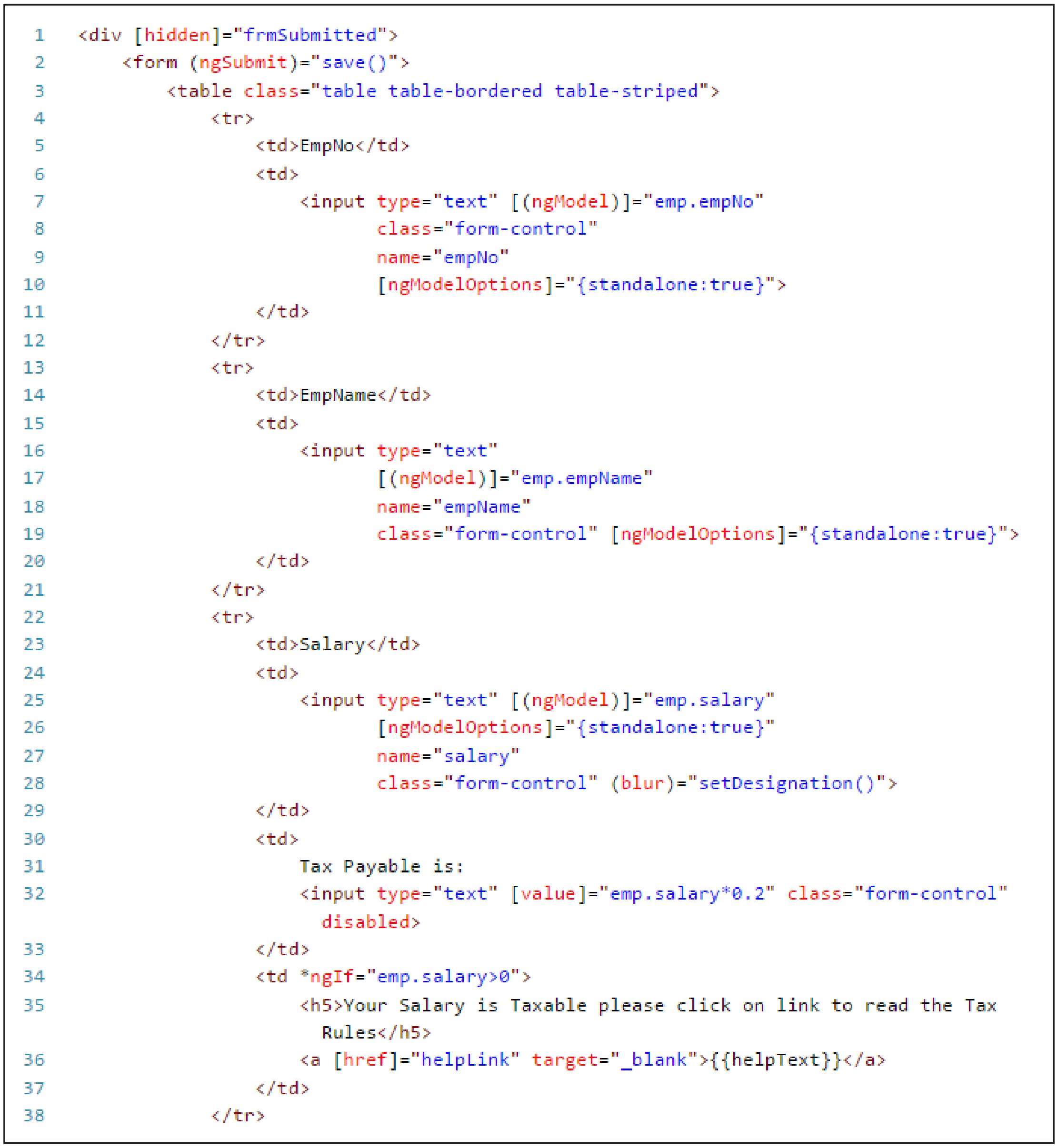
Listing 20.5: *The complete code*



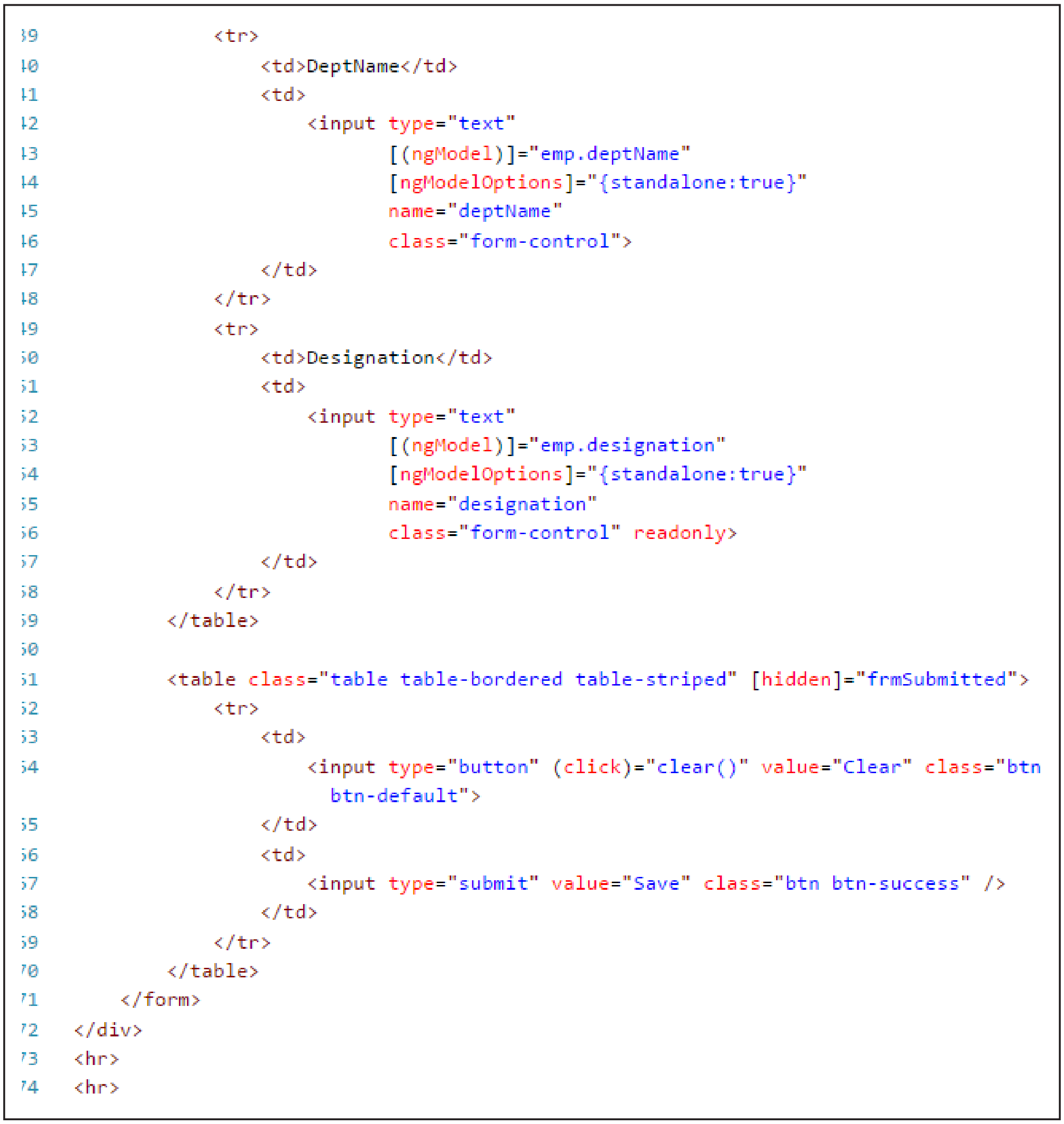
```
import {Component} from '@angular/core';
import {Employee} from './employee.model';

@Component({
    selector: 'emp-data',
    templateUrl: './app/employee.html'
})
export class EmployeeComponent {
    emp: Employee;
    employees: Employee[];
    helpLink: string;
    helpText: string;
    frmSubmitted: boolean;

    constructor() {
        this.emp = new Employee(0, '', 0, '', '');
        this.employees = [];
        this.helpText = "Tax Rules";
        this.helpLink = "./app/taxhelper.html";
        this.frmSubmitted = false;
    }

    clear() {
        this.emp = new Employee(0, '', 0, '', '');
    }
    save() {
        if (this.emp.empName.length > 0 && this.emp.deptName.length > 0) {
            this.employees.push(this.emp);
            this.frmSubmitted = true;
        }
    }
    loadForm() {
        this.emp = new Employee(0, '', 0, '', '');
        this.frmSubmitted = false;
    }
    setDesignation() {
        if (this.emp.salary >= 1000 && this.emp.salary <= 12000) {
            this.emp.designation = "Jr. Programmer";
        }
        if (this.emp.salary > 12000 && this.emp.salary <= 30000) {
            this.emp.designation = "Lead";
        }

        if (this.emp.salary > 30000 && this.emp.salary <= 60000) {
            this.emp.designation = "Group Lead";
        }

        if (this.emp.salary > 60000) {
            this.emp.designation = "Manager";
        }
    }
}
```

The employee.html file is modified by adding the <form> tag enclosing the table containing employee information <input> fields. The code uses the **ngSubmit** event object to bind the **save()** function to the <form>. The <form> tag is enclosed in <div> tag so that it can show/hide the table with <input> elements. The **hidden** property of <div> tag will be bound with the **frmSubmitted** property using **Property Binding.** The code replaces the **Save** button with the **submit** button. The following listing shows the modified markup for the table enclosed using <form>

Listing 20.6: *The form markup with databinding and form submission*



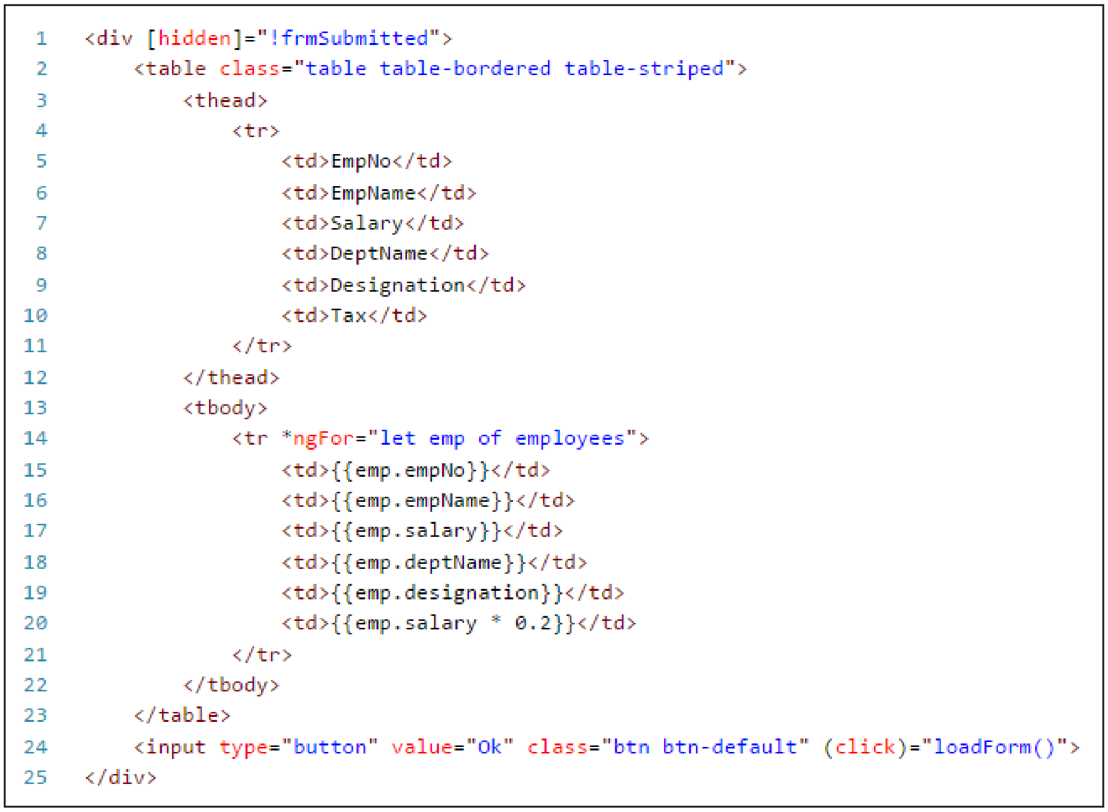
```
<div [hidden]="frmSubmitted">
    <form (ngSubmit)="save()">
        <table class="table table-bordered table-striped">
            <tr>
                <td>EmpNo</td>
                <td>
                    <input type="text" [(ngModel)]="emp.empNo"
                           class="form-control"
                           name="empNo"
                           [ngModelOptions]="{standalone:true}">
                </td>
            </tr>
            <tr>
                <td>EmpName</td>
                <td>
                    <input type="text"
                           [(ngModel)]="emp.empName"
                           name="empName"
                           class="form-control" [ngModelOptions]="{standalone:true}">
                </td>
            </tr>
            <tr>
                <td>Salary</td>
                <td>
                    <input type="text" [(ngModel)]="emp.salary"
                           [ngModelOptions]="{standalone:true}"
                           name="salary"
                           class="form-control" (blur)="setDesignation()">
                </td>
                <td>
                    Tax Payable is:
                    <input type="text" [value]="emp.salary*0.2" class="form-control"
                      disabled>
                </td>
                <td *ngIf="emp.salary>0">
                    <h5>Your Salary is Taxable please click on link to read the Tax
                      Rules</h5>
                    <a [href]="helpLink" target="_blank">{{helpText}}</a>
                </td>
            </tr>
```



```
            <tr>
                <td>DeptName</td>
                <td>
                    <input type="text"
                           [(ngModel)]="emp.deptName"
                           [ngModelOptions]="{standalone:true}"
                           name="deptName"
                           class="form-control">
                </td>
            </tr>
            <tr>
                <td>Designation</td>
                <td>
                    <input type="text"
                           [(ngModel)]="emp.designation"
                           [ngModelOptions]="{standalone:true}"
                           name="designation"
                           class="form-control" readonly>
                </td>
            </tr>
        </table>

        <table class="table table-bordered table-striped" [hidden]="frmSubmitted">
            <tr>
                <td>
                    <input type="button" (click)="clear()" value="Clear" class="btn
                      btn-default">
                </td>
                <td>
                    <input type="submit" value="Save" class="btn btn-success" />
                </td>
            </tr>
        </table>
    </form>
</div>
<hr>
<hr>
```

In employee.html, create a <table> which shows the list of Employee added in the Employees array. This table is enclosed in a <div> tag. The <div> tag is bound using **frmSubmitted** property to the **hidden** property using **Property Binding**, initially hiding this table when the form loads and making it visible when the form is submitted. The code has a new button at the bottom of the table which, is bound with the **loadFrom()** function using click **Event Binding**. When this button is clicked, the Employee List will be hidden and the Employee Form will be displayed. Listing 20-7 shows all the modifications in index.html.

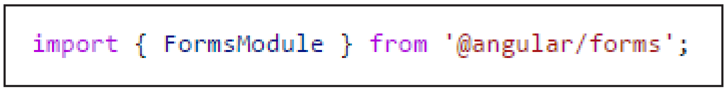
Listing 20.7: *The table which will be populated based on Employees*



```
<div [hidden]="!frmSubmitted">
    <table class="table table-bordered table-striped">
        <thead>
            <tr>
                <td>EmpNo</td>
                <td>EmpName</td>
                <td>Salary</td>
                <td>DeptName</td>
                <td>Designation</td>
                <td>Tax</td>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor="let emp of employees">
                <td>{{emp.empNo}}</td>
                <td>{{emp.empName}}</td>
                <td>{{emp.salary}}</td>
                <td>{{emp.deptName}}</td>
                <td>{{emp.designation}}</td>
                <td>{{emp.salary * 0.2}}</td>
            </tr>
        </tbody>
    </table>
    <input type="button" value="Ok" class="btn btn-default" (click)="loadForm()">
</div>
```

To use the FormsModule for the application, import it into the NgModule defined in the main.ts file as shown in Listing 20.8

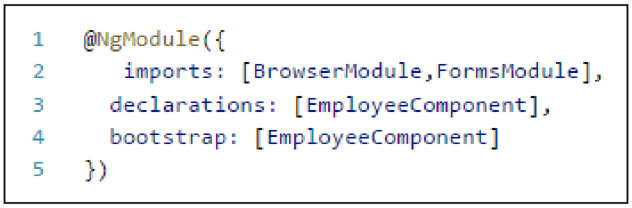
Listing 20.8: *Importing form module*



```
import { FormsModule } from '@angular/forms';
```

The use of FormsModule in the NgModule is shown in the following Listing.

Listing 20.9: *Declaration of @NgModule*



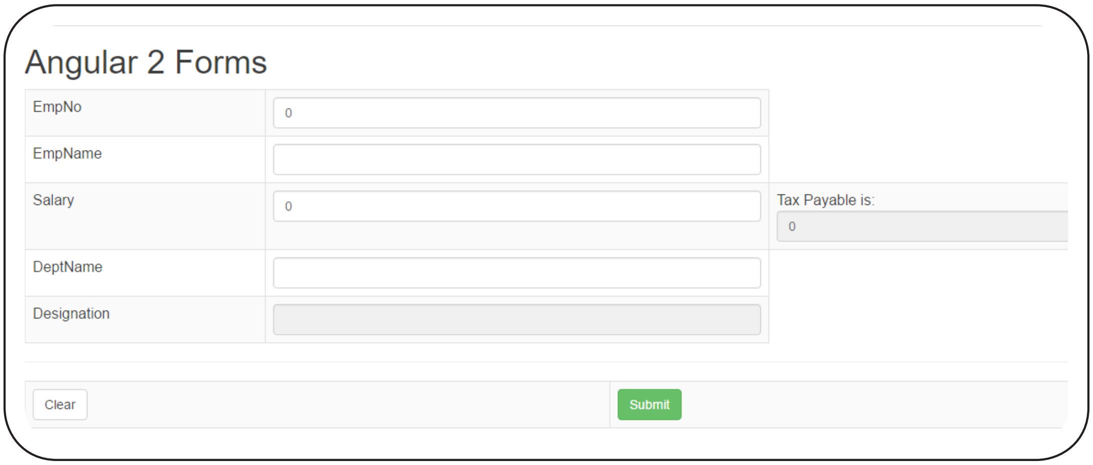
```
@NgModule({
   imports: [BrowserModule,FormsModule],
  declarations: [EmployeeComponent],
  bootstrap: [EmployeeComponent]
})
```

RUNNING THE APPLICATION

Right-click on index.html and select **Open in Command Prompt.** Enter the following command from this command prompt

```
npm run start
```

The Employee Form will be shown as seen in figure 20-1

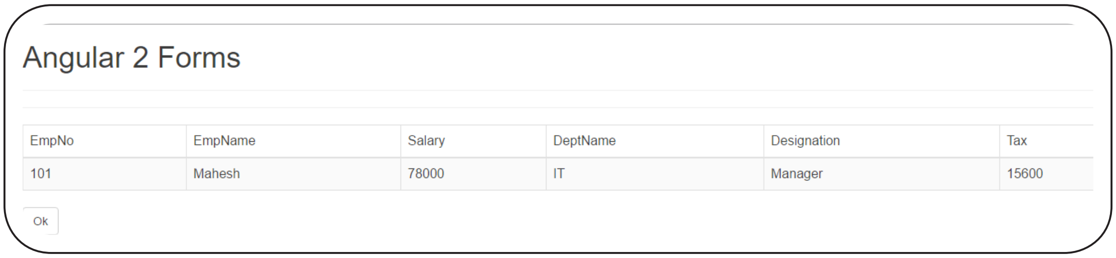


Angular 2 Forms

EmpNo	0	
EmpName		
Salary	0	Tax Payable is: 0
DeptName		
Designation		

***Figure 20.1:** Running form*

Enter Employee details in the TextBoxes and click on the **Submit** button. The Employee List will be displayed as shown in Figure 20-2:



Angular 2 Forms

EmpNo	EmpName	Salary	DeptName	Designation	Tax
101	Mahesh	78000	IT	Manager	15600

***Figure 20.2:** Employees data displayed in table*

Click the **OK** button and the Employee form will be displayed with TextBoxes cleared.

This code does not formally use the **ngForm** object directive because the <form> tag in the Employee.html is present within the scope of the EmployeeComponent. This upgrades the <form> tag to **ngForm**.

Chapter

21

Angular 2 Model-Driven Forms

CHAPTER 20 FOCUSED ON Angular 2 Forms and the basic set of controls and directives to be used in forms. It demonstrated how to post data using Angular form. In Angular 2, Forms are created in two different ways:

1. Model-Driven Forms
   a. The form is designed by importing the ReactiveFormsModule.
   b. The validation logic is separated from the form and handled at the component class level in the code.
   c. Since the validation logic is isolated from the form, it can be unit tested easily.
2. Template-Driven Form
   a. The form is designed by importing the FormsModule.
   b. The validations are handed in the template .
   c. The unit testing of the validation is tricky.

This chapter will discuss and implement **Model-Driven forms**.

Model-Driven Forms

Traditionally, a Form is composed of the following parts:

- The DOM, for form rendering.
- The field definition within the DOM, which is also known as a Template.
- The client-side UI logic, e.g. validation, field default values, etc. known as the Form Model.
- The domain model, which defines the fields that are exposed to UI elements.

In an earlier Angular release, the form model used for validation was provided using the validation directives in the UI itself. In Angular 2, a form model is created using the

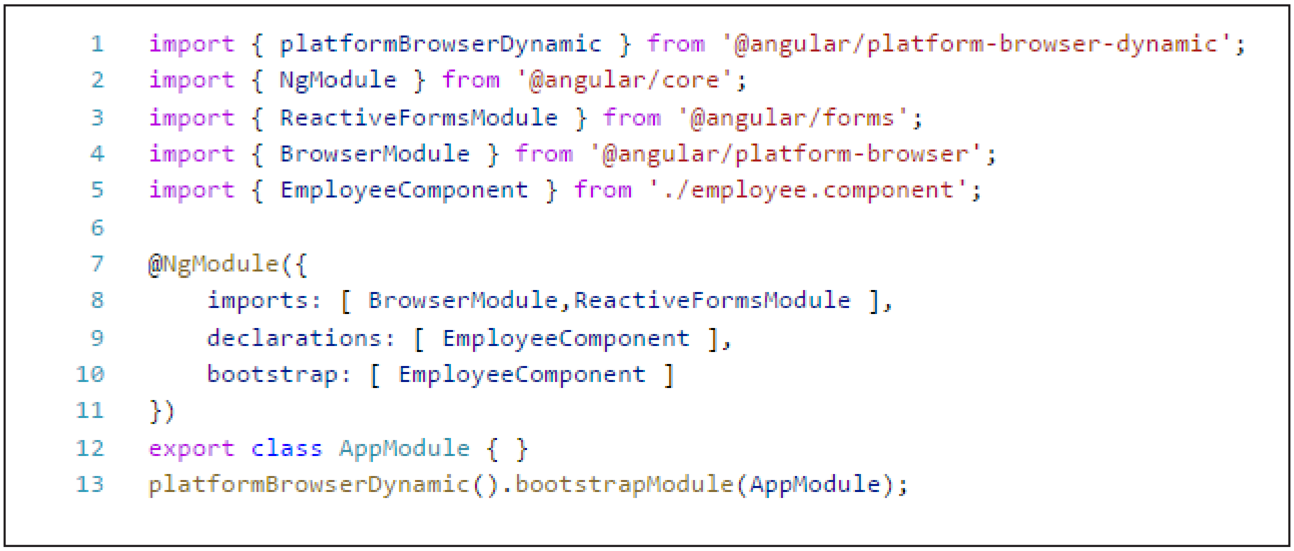
FormGroup object. This object is used to aggregate values of each child **FormControl** into one single object using its name as key. This object can be used to create a form model to track validity of the form, and its controls.

This chapter uses the FormGroup object to create the Form model for data binding with UI. Since the Model-Driven approach isolates the UI logic from the DOM, it is more testable and easy for maintaining code.

This chapter is implemented using the code of Chapter 20. If you want to follow along, make a copy of the code from Chapter 20 and make sure it is running before proceeding further.

Modify main.ts to use the ReactiveFormsModule module from the @angular/forms package as shown in the following listing:

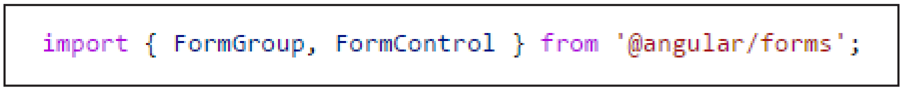
Listing 21.1 main.ts file importing ReactiveForms module



```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';
import { EmployeeComponent } from './employee.component';

@NgModule({
    imports: [ BrowserModule,ReactiveFormsModule ],
    declarations: [ EmployeeComponent ],
    bootstrap: [ EmployeeComponent ]
})
export class AppModule { }
platformBrowserDynamic().bootstrapModule(AppModule);
```

Modify employee.component.ts to implement the Model-Driven form by importing @angular/forms module of Angular 2 as shown in the following listing:

Listing 21.2: *Importing form group and control module*

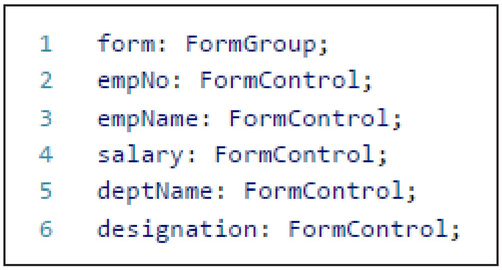


```
import { FormGroup, FormControl } from '@angular/forms';
```

In the code, create the FormGroup so that the model can be defined to expose the user interface using Data-Binding. The **FormControl** represents an input field which provides input value and validity of the value. This is used to sync all form control elements into a FormGroup. The **FormGroup** can also represent a collection of all form elements whose values will be used to represent the state of the form.

In the EmployeeComponent class, declare the FormGroup and FormControl objects as shown in Listing 21.3.

Listing 21.3: *Declaration of FormGroup and FormControl Objects*

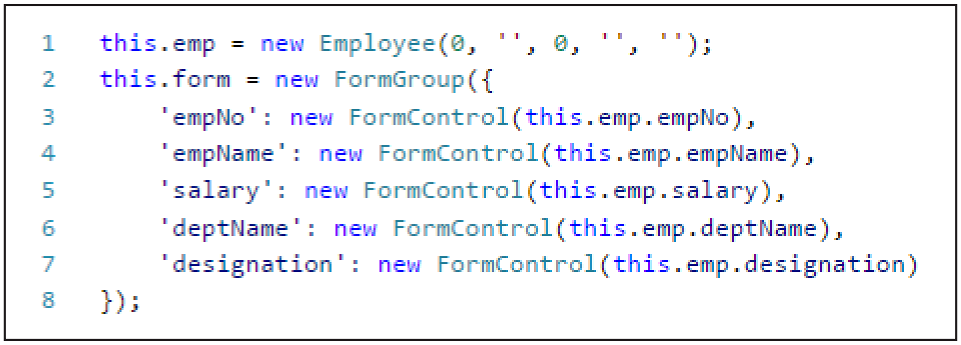


```
form: FormGroup;
empNo: FormControl;
empName: FormControl;
salary: FormControl;
deptName: FormControl;
designation: FormControl;
```

The **form** object is used to define a collection of fields that are defined using Form-Control object e.g. EmpNo, EmpName, etc.

The Constructor of the EmployeeComponent class uses the FormGroup object to group all the Controls for defining Data-Model by using properties of the Employee model class as shown Listing 21.4.

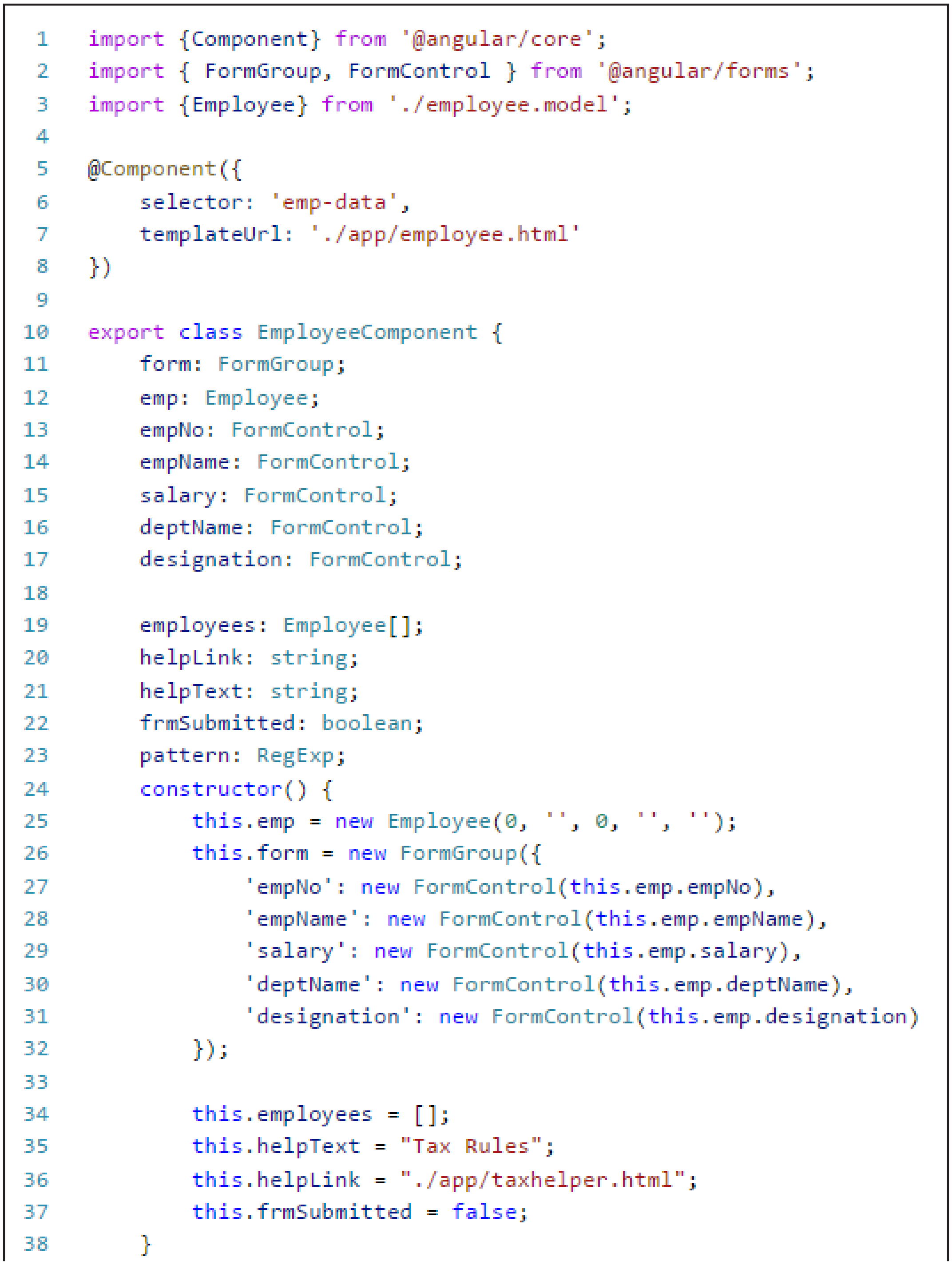
Listing 21.4: *Creating FormGroup using FormControl*



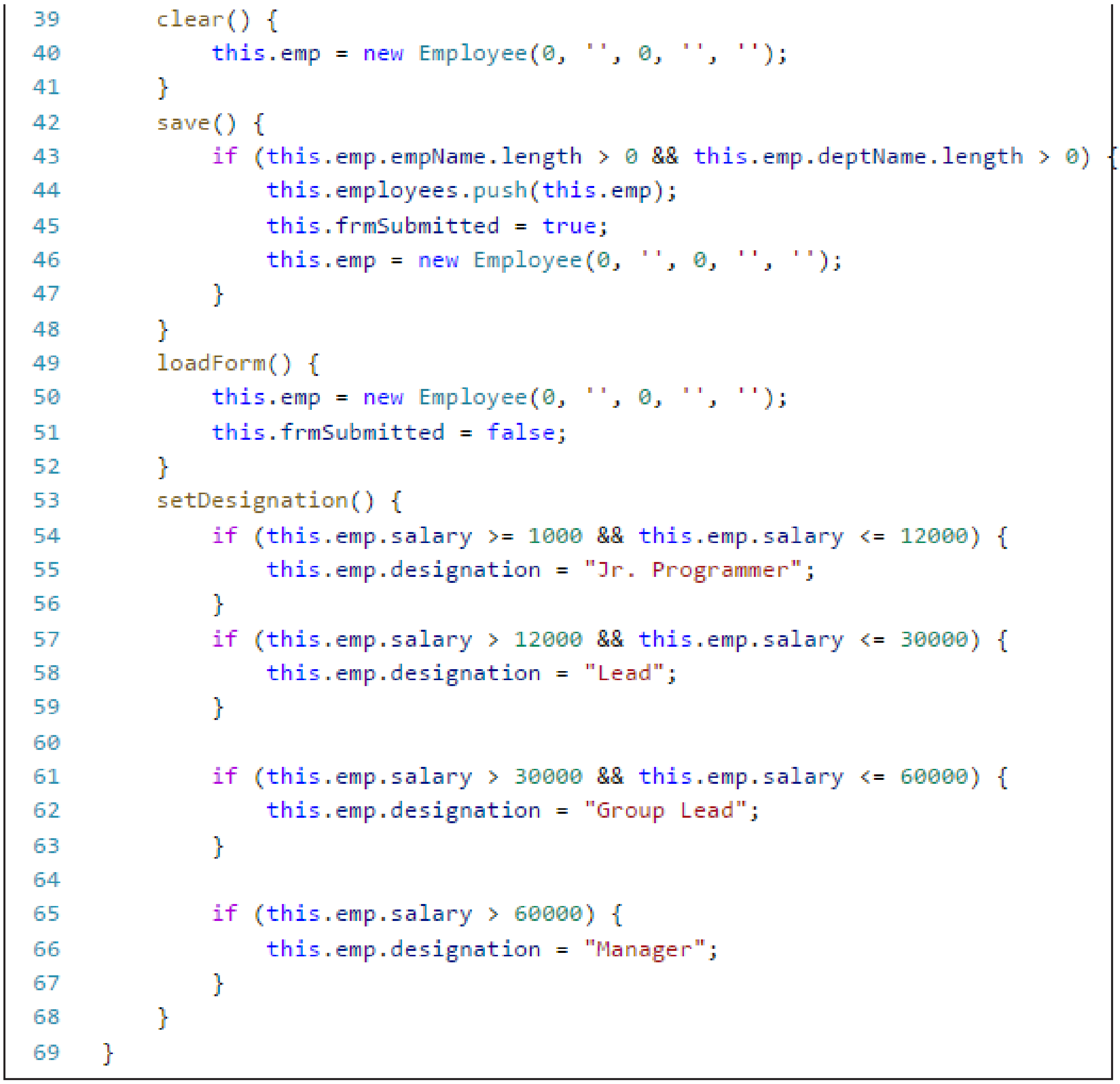
```
this.emp = new Employee(0, '', 0, '', '');
this.form = new FormGroup({
    'empNo': new FormControl(this.emp.empNo),
    'empName': new FormControl(this.emp.empName),
    'salary': new FormControl(this.emp.salary),
    'deptName': new FormControl(this.emp.deptName),
    'designation': new FormControl(this.emp.designation)
});
```

The FormControl object is instantiated using the Employee properties passed to it. The **FormControl** constructor can optionally also pass the validation rules as input parameter for validation of properties. The implementation of the validations can be seen in forthcoming Chapters 22 and 23. The entire EmployeeComponent code will be as shown in listing 21-5:

Listing 21.5: *The Complete code*



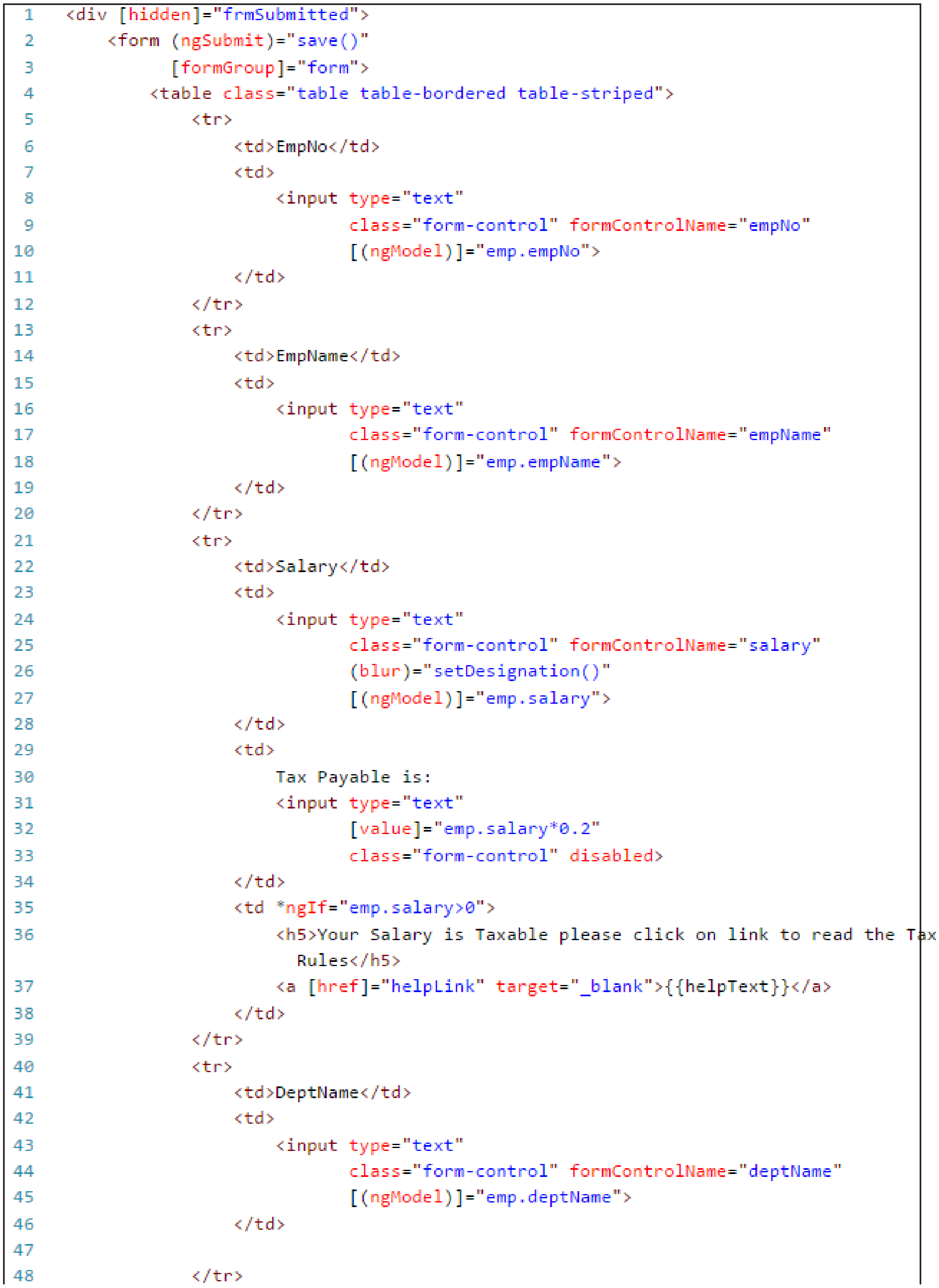
```
import {Component} from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';
import {Employee} from './employee.model';

@Component({
    selector: 'emp-data',
    templateUrl: './app/employee.html'
})

export class EmployeeComponent {
    form: FormGroup;
    emp: Employee;
    empNo: FormControl;
    empName: FormControl;
    salary: FormControl;
    deptName: FormControl;
    designation: FormControl;

    employees: Employee[];
    helpLink: string;
    helpText: string;
    frmSubmitted: boolean;
    pattern: RegExp;
    constructor() {
        this.emp = new Employee(0, '', 0, '', '');
        this.form = new FormGroup({
            'empNo': new FormControl(this.emp.empNo),
            'empName': new FormControl(this.emp.empName),
            'salary': new FormControl(this.emp.salary),
            'deptName': new FormControl(this.emp.deptName),
            'designation': new FormControl(this.emp.designation)
        });

        this.employees = [];
        this.helpText = "Tax Rules";
        this.helpLink = "./app/taxhelper.html";
        this.frmSubmitted = false;
    }
```



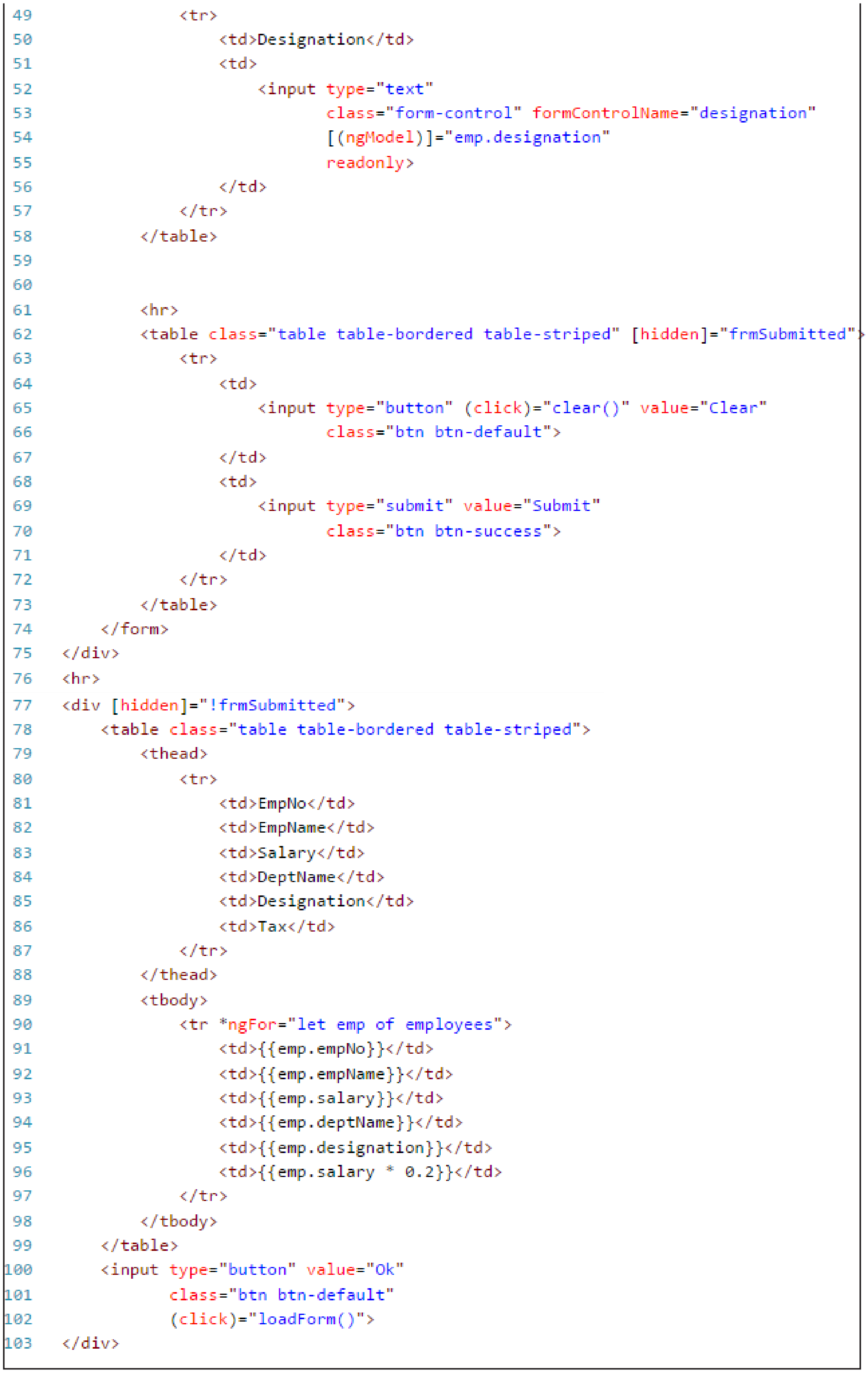
```
    clear() {
        this.emp = new Employee(0, '', 0, '', '');
    }
    save() {
        if (this.emp.empName.length > 0 && this.emp.deptName.length > 0) {
            this.employees.push(this.emp);
            this.frmSubmitted = true;
            this.emp = new Employee(0, '', 0, '', '');
        }
    }
    loadForm() {
        this.emp = new Employee(0, '', 0, '', '');
        this.frmSubmitted = false;
    }
    setDesignation() {
        if (this.emp.salary >= 1000 && this.emp.salary <= 12000) {
            this.emp.designation = "Jr. Programmer";
        }
        if (this.emp.salary > 12000 && this.emp.salary <= 30000) {
            this.emp.designation = "Lead";
        }

        if (this.emp.salary > 30000 && this.emp.salary <= 60000) {
            this.emp.designation = "Group Lead";
        }

        if (this.emp.salary > 60000) {
            this.emp.designation = "Manager";
        }
    }
}
```

To define the model binding with DOM elements, modify the employee.html <form> tag using the **formGroup** directive. This represents the FormGroup defined in the EmployeeComponent. The **formControlName** represents the name of the form element into the formGroup. The employee.html will be modified as shown in listing 21-6:

Listing 21.6: *The markup with Databinding and formControlName Declaration*



```
<div [hidden]="frmSubmitted">
    <form (ngSubmit)="save()"
          [formGroup]="form">
        <table class="table table-bordered table-striped">
            <tr>
                <td>EmpNo</td>
                <td>
                    <input type="text"
                           class="form-control" formControlName="empNo"
                           [(ngModel)]="emp.empNo">
                </td>
            </tr>
            <tr>
                <td>EmpName</td>
                <td>
                    <input type="text"
                           class="form-control" formControlName="empName"
                           [(ngModel)]="emp.empName">
                </td>
            </tr>
            <tr>
                <td>Salary</td>
                <td>
                    <input type="text"
                           class="form-control" formControlName="salary"
                           (blur)="setDesignation()"
                           [(ngModel)]="emp.salary">
                </td>
                <td>
                    Tax Payable is:
                    <input type="text"
                           [value]="emp.salary*0.2"
                           class="form-control" disabled>
                </td>
                <td *ngIf="emp.salary>0">
                    <h5>Your Salary is Taxable please click on link to read the Tax
                      Rules</h5>
                    <a [href]="helpLink" target="_blank">{{helpText}}</a>
                </td>
            </tr>
            <tr>
                <td>DeptName</td>
                <td>
                    <input type="text"
                           class="form-control" formControlName="deptName"
                           [(ngModel)]="emp.deptName">
                </td>

            </tr>
```



```
            <tr>
                <td>Designation</td>
                <td>
                    <input type="text"
                           class="form-control" formControlName="designation"
                           [(ngModel)]="emp.designation"
                           readonly>
                </td>
            </tr>
        </table>


        <hr>
        <table class="table table-bordered table-striped" [hidden]="frmSubmitted">
            <tr>
                <td>
                    <input type="button" (click)="clear()" value="Clear"
                           class="btn btn-default">
                </td>
                <td>
                    <input type="submit" value="Submit"
                           class="btn btn-success">
                </td>
            </tr>
        </table>
    </form>
</div>
<hr>
<div [hidden]="!frmSubmitted">
    <table class="table table-bordered table-striped">
        <thead>
            <tr>
                <td>EmpNo</td>
                <td>EmpName</td>
                <td>Salary</td>
                <td>DeptName</td>
                <td>Designation</td>
                <td>Tax</td>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor="let emp of employees">
                <td>{{emp.empNo}}</td>
                <td>{{emp.empName}}</td>
                <td>{{emp.salary}}</td>
                <td>{{emp.deptName}}</td>
                <td>{{emp.designation}}</td>
                <td>{{emp.salary * 0.2}}</td>
            </tr>
        </tbody>
    </table>
    <input type="button" value="Ok"
           class="btn btn-default"
           (click)="loadForm()">
</div>
```

Save the project. To run the project, right-click on Index.html in the File explorer of VS Code and select option **Open in Command Prompt.** Run the following command from this command prompt:

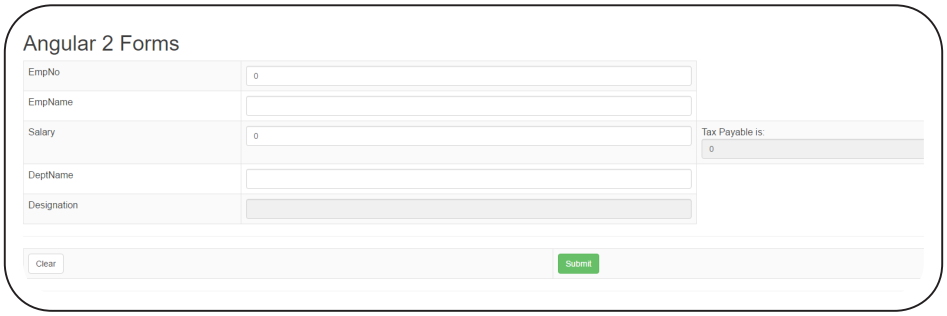
```
npm run start
```

This will start the server.

Open the browser and enter the following address:

http://localhost:3000

The browser will display the following output:

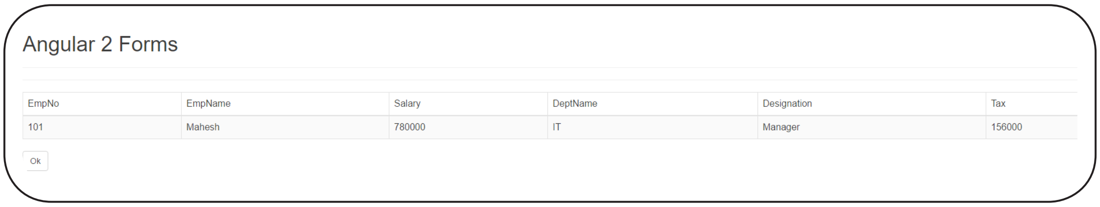


Angular 2 Forms

EmpNo	0	
EmpName		
Salary	0	Tax Payable is: 0
DeptName		
Designation		

Figure 21.1: *Loading form in browser*

Enter values for the fields and click on the submit button. The data will be saved as shown in Figure 21-2:



Angular 2 Forms

EmpNo	EmpName	Salary	DeptName	Designation	Tax
101	Mahesh	780000	IT	Manager	156000

Figure 21.2: *Result with list of Employees.*

Click on the **Ok** button to clear the form.

Conclusion

Model-Driven form is a new feature introduced in Angular 2. This feature allows you to isolate fields from the DOM using FormBuilder, ControlGroup and Control objects. Since fields are isolated from the DOM, they can be set for validation rules and unit-tested easily. This leads to maintainable code.

Chapter 22

Angular 2 Form Validation

IN ANY WEB APPLICATION, form validation plays a very important role. Validation provides a mechanism to restrict the end-user from entering wrong inputs. Angular 2 uses in-built validation features of HTML 5 like required, minLength, maxLength, etc. One important point to be noted here is that although HTML 5 validations can be suppressed using the **novalidate** attribute of the <form> tag, Angular 2 can validate the form fields using validation attributes like required, minLength, maxLength, etc.

This chapter will implement validations on the **EmployeeComponent** created in Chapter 21.

Validation of controls in the HTML form can be performed using the following attributes:

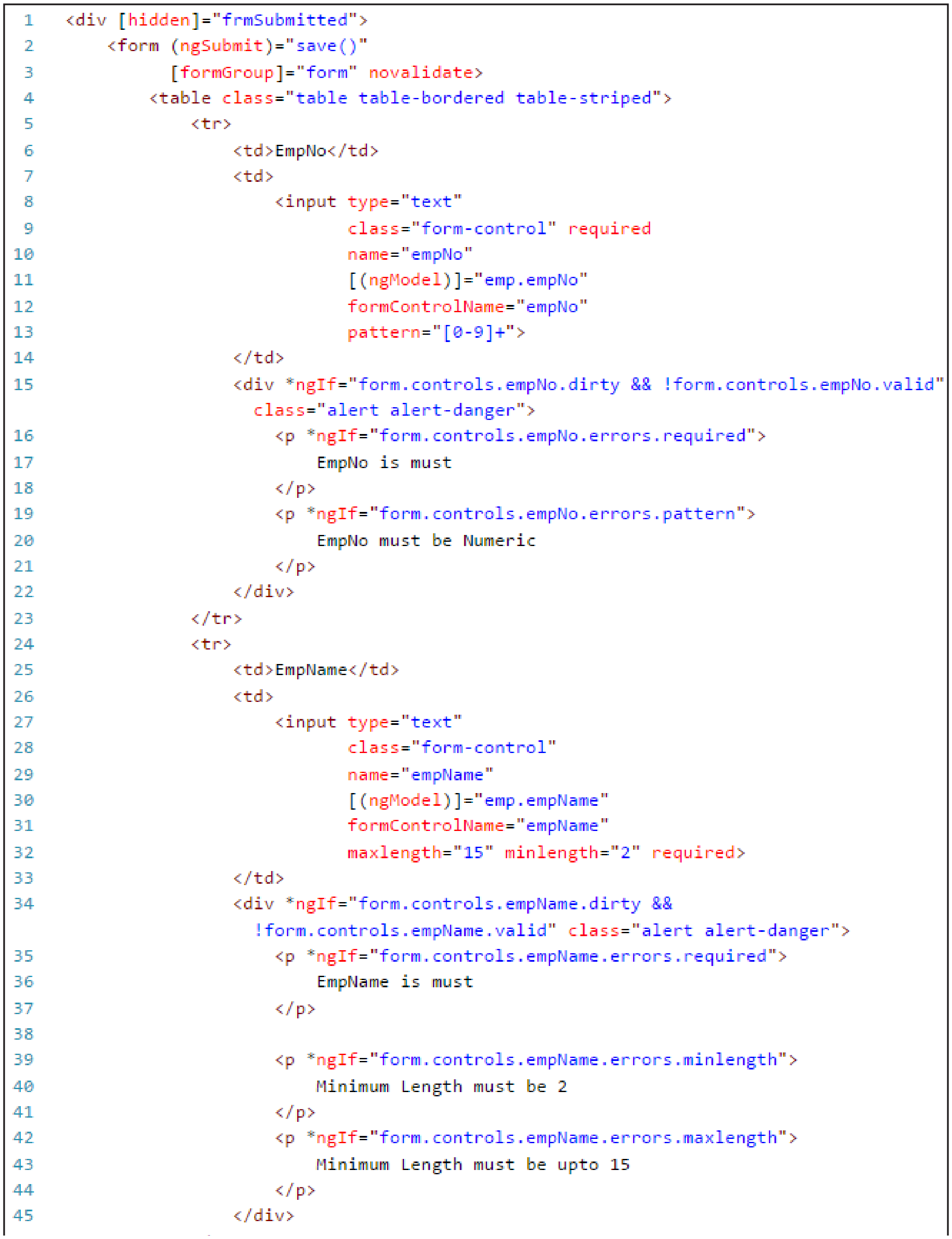
- **required:** Used for specifying the value of input element as mandatory
- **pattern:** sets the regular expression for the data to be entered in an input element
- **minlength:** The minimum length of the text entered in the input element
- **maxlength:** The maximum length of the text entered in the input element

To implement Databinding, the ngModel directive is used; but to validate the data updates using ngModel, the following set of classes must be used to validate the updated state. Angular 2 adds these classes to the form controls when the validation runs on them.

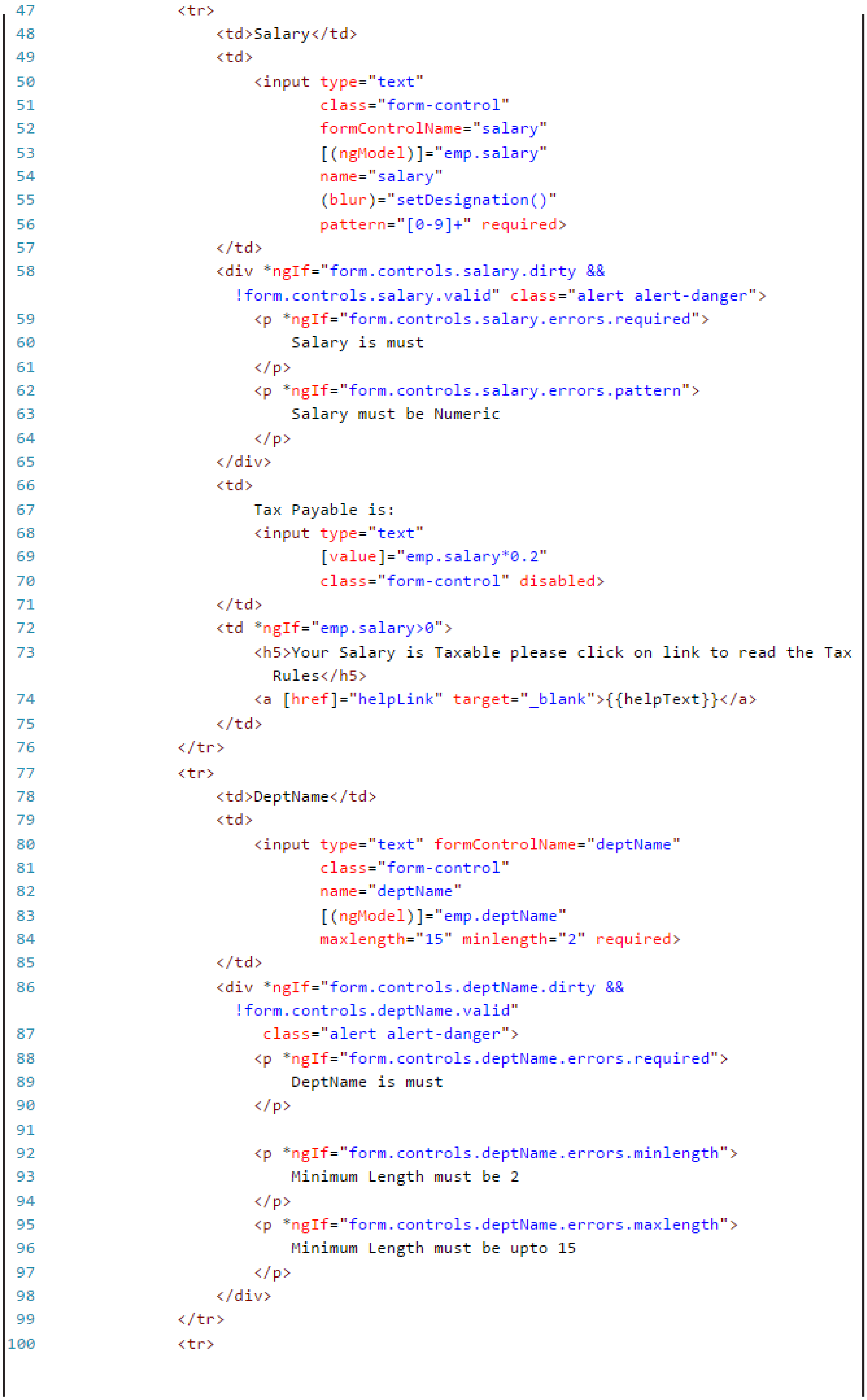
- **ng-valid:** true when the control's value is valid.
- **ng-invalid:** executed as true when the control's value is invalid.
- **ng-dirty:** true when the control's value is changed.
- **ng-pristine:** executed as true when control's value is unchanged.

To implement this chapter, modify Employee.html as mentioned in the following listing:

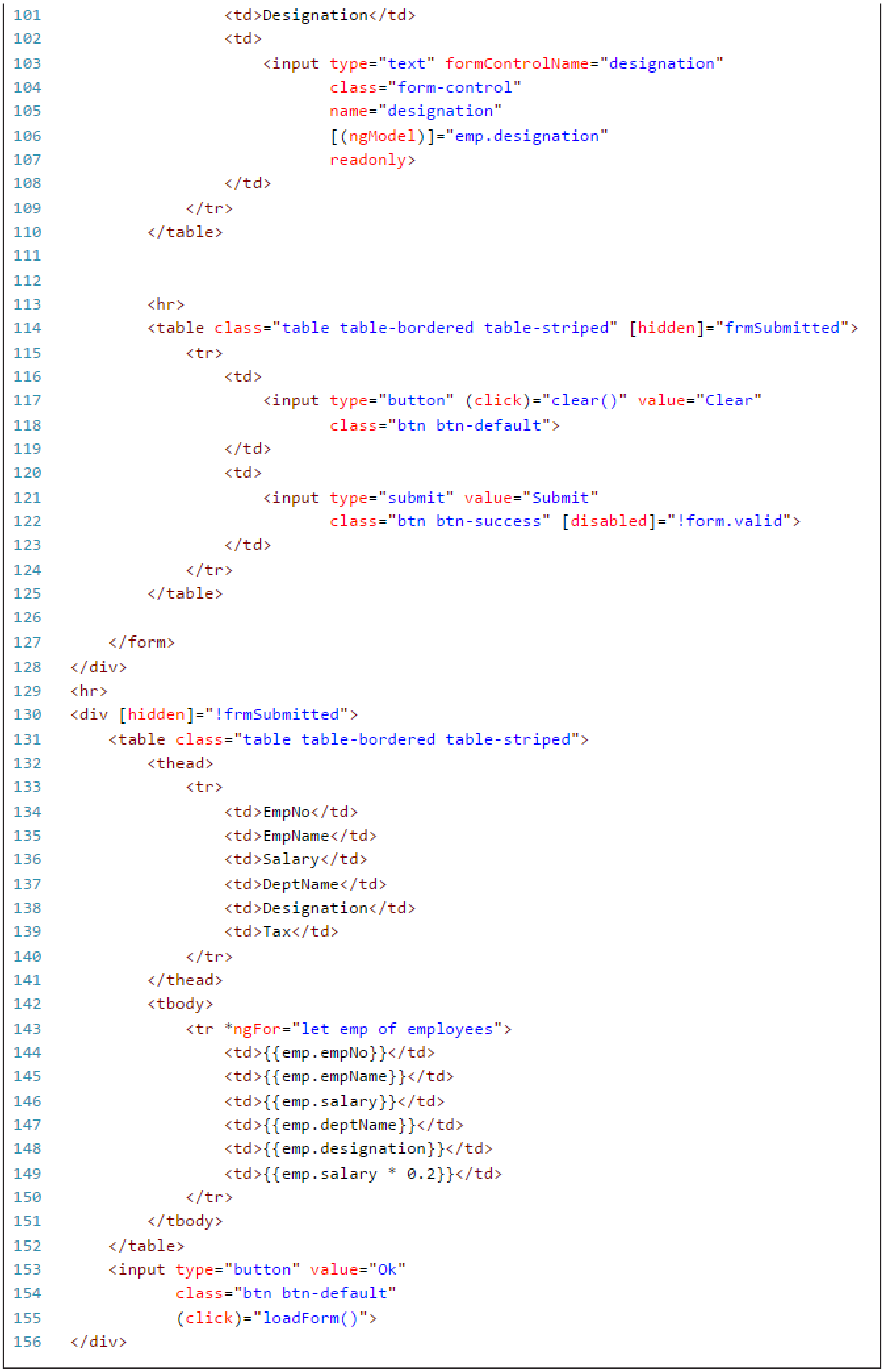
Listing 22.1: *Markup with validations*



```
<div [hidden]="frmSubmitted">
    <form (ngSubmit)="save()"
          [formGroup]="form" novalidate>
        <table class="table table-bordered table-striped">
            <tr>
                <td>EmpNo</td>
                <td>
                    <input type="text"
                           class="form-control" required
                           name="empNo"
                           [(ngModel)]="emp.empNo"
                           formControlName="empNo"
                           pattern="[0-9]+">
                </td>
                <div *ngIf="form.controls.empNo.dirty && !form.controls.empNo.valid"
                  class="alert alert-danger">
                    <p *ngIf="form.controls.empNo.errors.required">
                        EmpNo is must
                    </p>
                    <p *ngIf="form.controls.empNo.errors.pattern">
                        EmpNo must be Numeric
                    </p>
                </div>
            </tr>
            <tr>
                <td>EmpName</td>
                <td>
                    <input type="text"
                           class="form-control"
                           name="empName"
                           [(ngModel)]="emp.empName"
                           formControlName="empName"
                           maxlength="15" minlength="2" required>
                </td>
                <div *ngIf="form.controls.empName.dirty &&
                  !form.controls.empName.valid" class="alert alert-danger">
                    <p *ngIf="form.controls.empName.errors.required">
                        EmpName is must
                    </p>

                    <p *ngIf="form.controls.empName.errors.minlength">
                        Minimum Length must be 2
                    </p>
                    <p *ngIf="form.controls.empName.errors.maxlength">
                        Minimum Length must be upto 15
                    </p>
                </div>
```

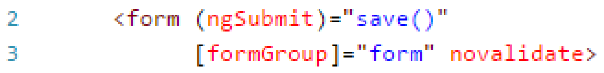


```
            <tr>
                <td>Salary</td>
                <td>
                    <input type="text"
                           class="form-control"
                           formControlName="salary"
                           [(ngModel)]="emp.salary"
                           name="salary"
                           (blur)="setDesignation()"
                           pattern="[0-9]+" required>
                </td>
                <div *ngIf="form.controls.salary.dirty &&
                  !form.controls.salary.valid" class="alert alert-danger">
                    <p *ngIf="form.controls.salary.errors.required">
                        Salary is must
                    </p>
                    <p *ngIf="form.controls.salary.errors.pattern">
                        Salary must be Numeric
                    </p>
                </div>
                <td>
                    Tax Payable is:
                    <input type="text"
                           [value]="emp.salary*0.2"
                           class="form-control" disabled>
                </td>
                <td *ngIf="emp.salary>0">
                    <h5>Your Salary is Taxable please click on link to read the Tax
                      Rules</h5>
                    <a [href]="helpLink" target="_blank">{{helpText}}</a>
                </td>
            </tr>
            <tr>
                <td>DeptName</td>
                <td>
                    <input type="text" formControlName="deptName"
                           class="form-control"
                           name="deptName"
                           [(ngModel)]="emp.deptName"
                           maxlength="15" minlength="2" required>
                </td>
                <div *ngIf="form.controls.deptName.dirty &&
                  !form.controls.deptName.valid"
                     class="alert alert-danger">
                    <p *ngIf="form.controls.deptName.errors.required">
                        DeptName is must
                    </p>

                    <p *ngIf="form.controls.deptName.errors.minlength">
                        Minimum Length must be 2
                    </p>
                    <p *ngIf="form.controls.deptName.errors.maxlength">
                        Minimum Length must be upto 15
                    </p>
                </div>
            </tr>
            <tr>
```



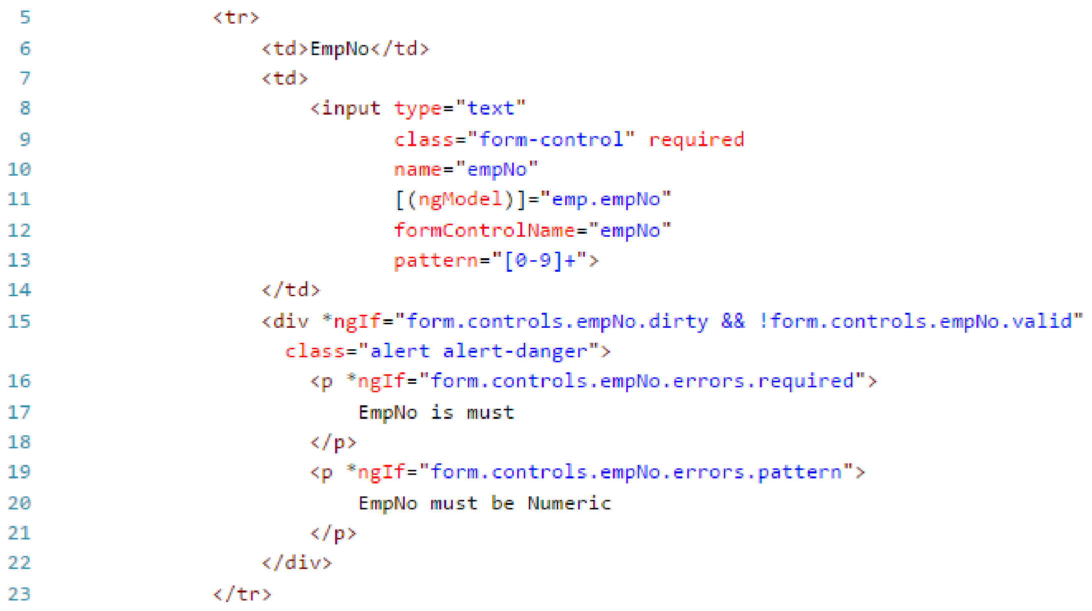
```
                <td>Designation</td>
                <td>
                    <input type="text" formControlName="designation"
                           class="form-control"
                           name="designation"
                           [(ngModel)]="emp.designation"
                           readonly>
                </td>
            </tr>
        </table>


        <hr>
        <table class="table table-bordered table-striped" [hidden]="frmSubmitted">
            <tr>
                <td>
                    <input type="button" (click)="clear()" value="Clear"
                           class="btn btn-default">
                </td>
                <td>
                    <input type="submit" value="Submit"
                           class="btn btn-success" [disabled]="!form.valid">
                </td>
            </tr>
        </table>

    </form>
</div>
<hr>
<div [hidden]="!frmSubmitted">
    <table class="table table-bordered table-striped">
        <thead>
            <tr>
                <td>EmpNo</td>
                <td>EmpName</td>
                <td>Salary</td>
                <td>DeptName</td>
                <td>Designation</td>
                <td>Tax</td>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor="let emp of employees">
                <td>{{emp.empNo}}</td>
                <td>{{emp.empName}}</td>
                <td>{{emp.salary}}</td>
                <td>{{emp.deptName}}</td>
                <td>{{emp.designation}}</td>
                <td>{{emp.salary * 0.2}}</td>
            </tr>
        </tbody>
    </table>
    <input type="button" value="Ok"
           class="btn btn-default"
           (click)="loadForm()">
</div>
```

Observe the following declarations in the form tag as shown in Listing 22.1:



```
<form (ngSubmit)="save()"
      [formGroup]="form" novalidate>
```

The **[formGroup]** is used for checking the validity of the form. The validation at the field level is defined using the following attribute values set for the <input> elements (check Listing 22-1)

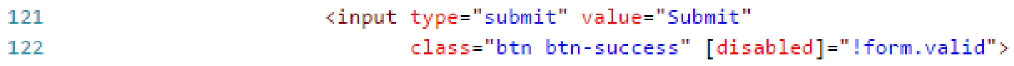


```
<tr>
    <td>EmpNo</td>
    <td>
        <input type="text"
               class="form-control" required
               name="empNo"
               [(ngModel)]="emp.empNo"
               formControlName="empNo"
               pattern="[0-9]+">
    </td>
    <div *ngIf="form.controls.empNo.dirty && !form.controls.empNo.valid"
      class="alert alert-danger">
        <p *ngIf="form.controls.empNo.errors.required">
            EmpNo is must
        </p>
        <p *ngIf="form.controls.empNo.errors.pattern">
            EmpNo must be Numeric
        </p>
    </div>
</tr>
```

This code taken from Listing 22-1 has the following validations implemented:

- The **`pattern`** attribute is used to define regular expression validation on the input element.
- The **`formControlName="empno"`** defines the <input> element as form field and the **`form.controls.empNo.dirty`** is used to verify the status of the element. If this is changed, then the value entered in the input element will be checked for validity using *ngIf directive. The **`form.controls.empNo.pattern`** value is used to check if the value entered is valid against the pattern attribute value for input element.

Note the following markup carefully:



```
<input type="submit" value="Submit"
       class="btn btn-success" [disabled]="!form.valid">
```

The expression `[disabled]="!empForm.valid"` will disable the submit button if the form has invalid values.

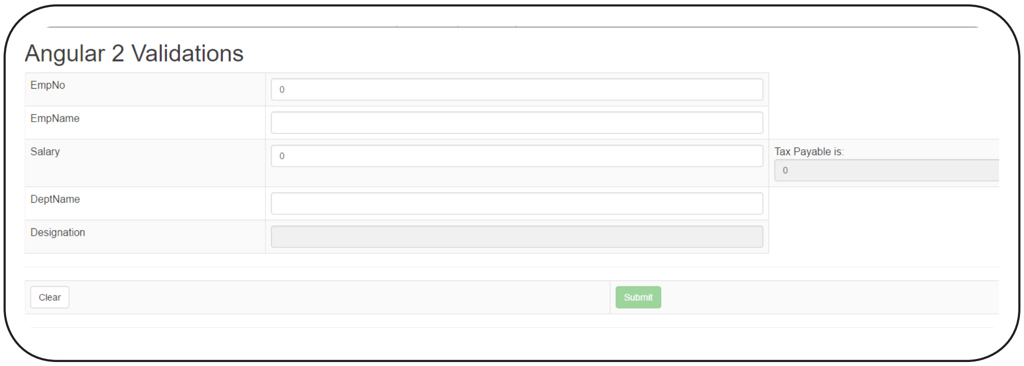
To run the application, right click on the index.html and select **Open in Command Prompt**. Run the following command on the command prompt:

```
npm run start
```

This command will start the server. Open the browser and enter the following URL:

```
http://localhost:3000
```

The form will be displayed in the browser with submit button as disabled:



Angular 2 Validations

EmpNo: 0

EmpName:

Salary: 0

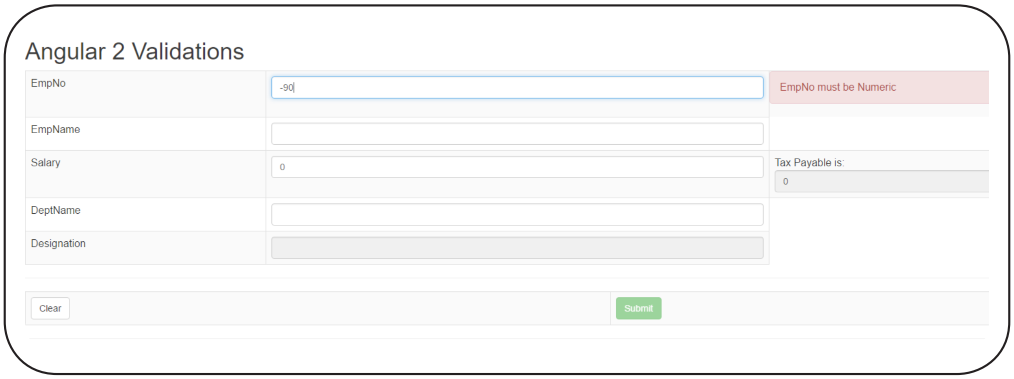
Tax Payable is: 0

DeptName:

Designation:

Figure 22.1: *Loading form*

Enter a negative value for the EmpNo and the field will be immediately validated with a validation error as shown in Figure 22-2:



Angular 2 Validations

EmpNo: -90

EmpNo must be Numeric

EmpName:

Salary: 0

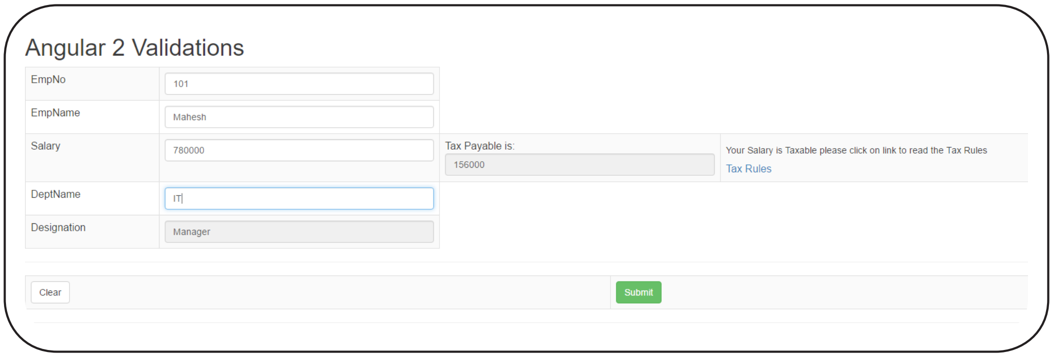
Tax Payable is: 0

DeptName:

Designation:

Figure 22.2: *Result with Validations*

Enter valid values in all textboxes to enable the Submit button as shown in Figure 22-3:



Angular 2 Validations

EmpNo: 101
EmpName: Mahesh
Salary: 780000
Tax Payable is: 156000
Your Salary is Taxable please click on link to read the Tax Rules
Tax Rules
DeptName: IT
Designation: Manager

Figure 22.3: *Valid Form*

Conclusion

We saw how Angular 2 provides three out of the box validators which can be applied using HTML properties. Chapter 23 shows yet another way of validating forms using the Control class.

Chapter

23

Angular 2 Form Validation with Control Object and Custom Validation

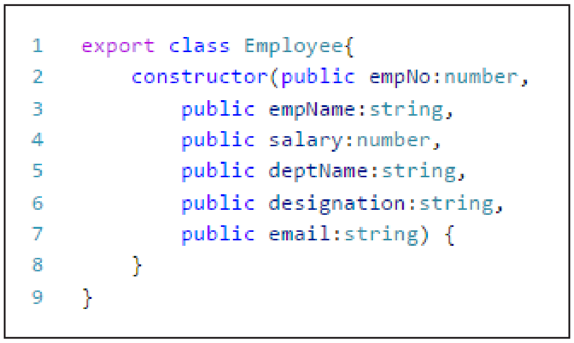
CHAPTER 22 EXPLAINED THE use of inbuilt validators to perform standard validation on DOM elements in Angular 2. It used the **ngForm** object to evaluate the validations on every DOM element, as well as ***ngIf** directive for evaluating validations and displaying error messages.

Custom validation is needed to implement a domain specific validation as per business needs. This helps to implement more accuracy in the data posted by the end-user, to the application. E.g. performing a validation on the digits of a Credit-Card with respect to its format.

This chapter will explain how to implement Form validation using the **formControlName** object (discussed in Chapter 21) and the implementation of custom validation. The **formControlName** object is used to define fields to bind with the DOM elements. The constructor of the Control object accepts the model property and the validation rule as input parameters.

Open the code for Chapter 22 and modify the employee.model.ts file to add a new property named email as string, as shown in the following listing:

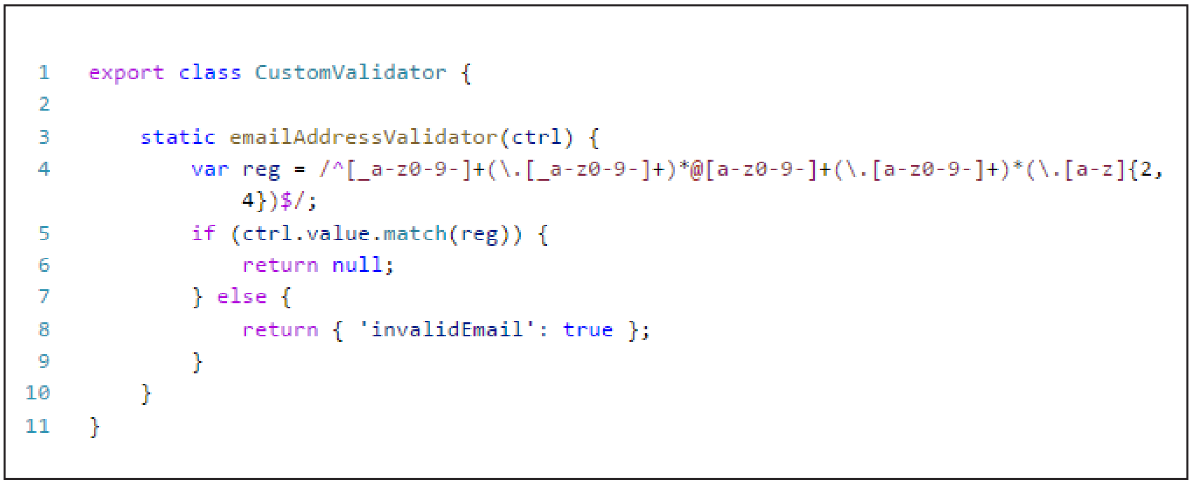
Listing 23.1: *The Employee class with Email property*



```
export class Employee{
    constructor(public empNo:number,
        public empName:string,
        public salary:number,
        public deptName:string,
        public designation:string,
        public email:string) {
    }
}
```

Add a new file named validator.ts to the project. This file will contain the **CustomValidator** class for validating the e-mail address for lowercase characters only. Add the code to it as shown in Listing 23-2.

Listing 23.2: *The Custom Validation Implementation*

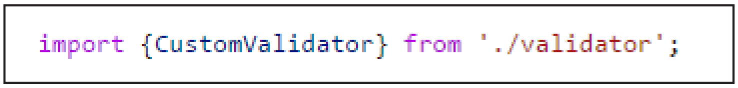


```
export class CustomValidator {

    static emailAddressValidator(ctrl) {
        var reg = /^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,
            4})$/;
        if (ctrl.value.match(reg)) {
            return null;
        } else {
            return { 'invalidEmail': true };
        }
    }
}
```

The **emailAddressValidator** function contains the logic for validating e-mail addresses using a regular expression.

To use this custom validator in the EmployeeComponent class, the code imports the CustomValidator class as shown in listing 23-3.

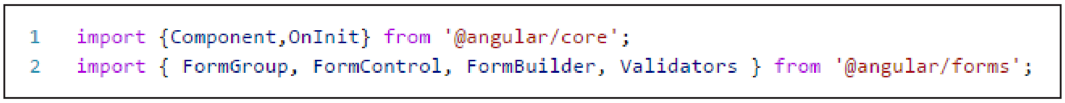
Listing 23.3: *Importing Custom validator*



```
import {CustomValidator} from './validator';
```

Import the following modules for component initialization and validators class as shown in Listing 23-4:

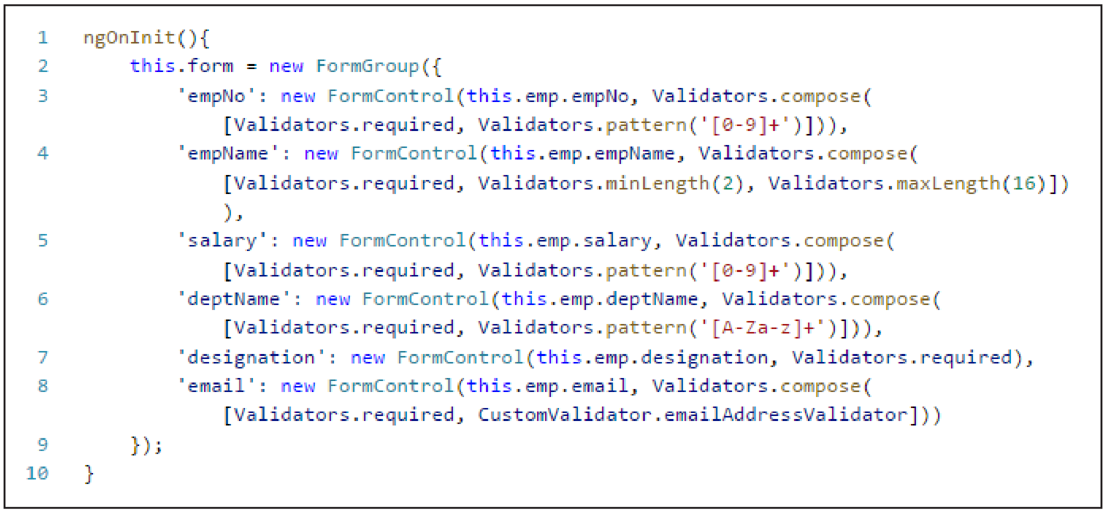
Listing 23.4: *Importing required dependencies*



```
import {Component,OnInit} from '@angular/core';
import { FormGroup, FormControl, FormBuilder, Validators } from '@angular/forms';
```

Also add validation rules in the ngOnInit of the EmployeeComponent class as shown in Listing 23-5:

Listing 23.5: *Implementing ngOnInit() with Custom validator*

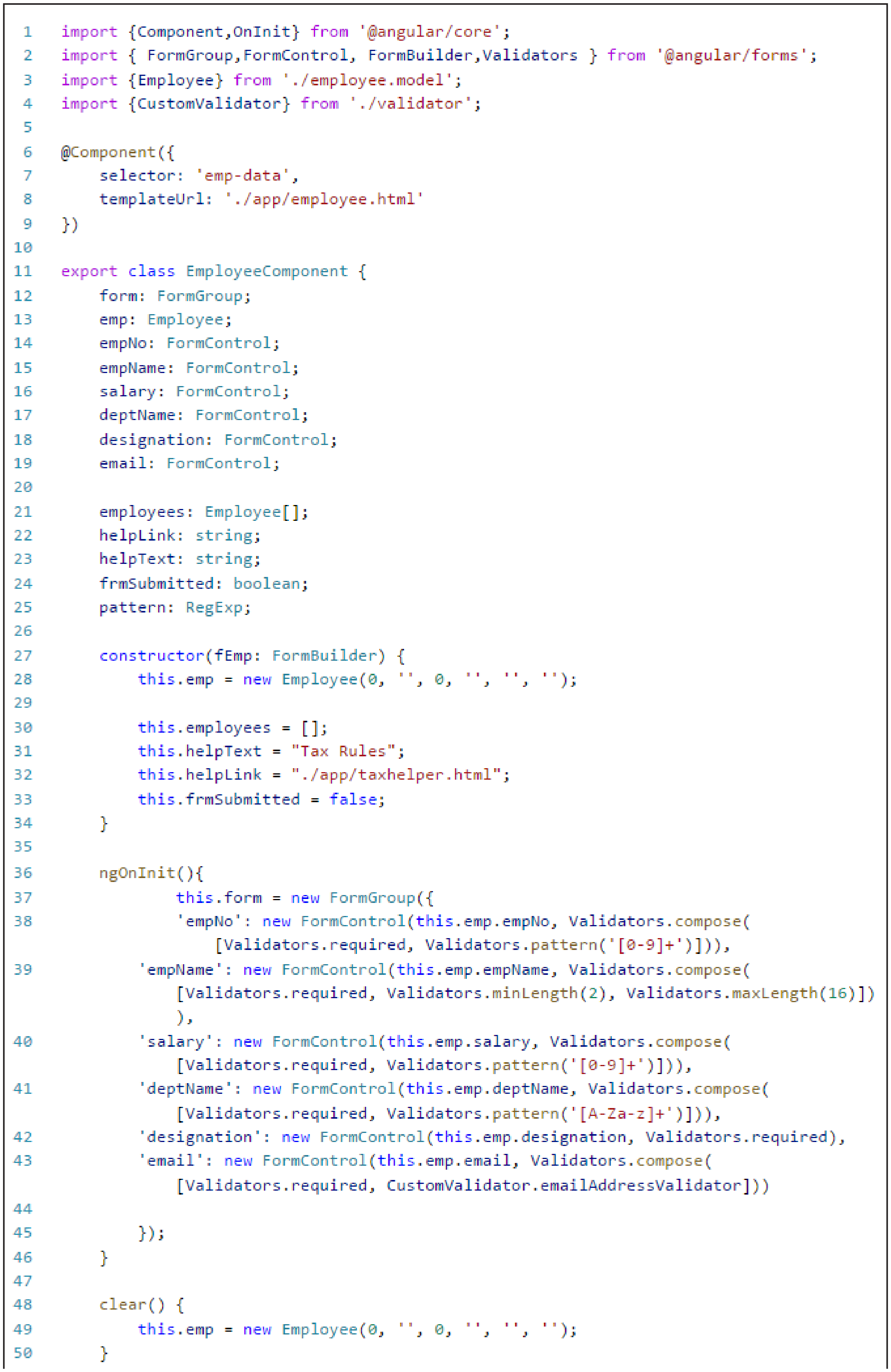


```
ngOnInit(){
    this.form = new FormGroup({
        'empNo': new FormControl(this.emp.empNo, Validators.compose(
            [Validators.required, Validators.pattern('[0-9]+')])),
        'empName': new FormControl(this.emp.empName, Validators.compose(
            [Validators.required, Validators.minLength(2), Validators.maxLength(16)])
            ),
        'salary': new FormControl(this.emp.salary, Validators.compose(
            [Validators.required, Validators.pattern('[0-9]+')])),
        'deptName': new FormControl(this.emp.deptName, Validators.compose(
            [Validators.required, Validators.pattern('[A-Za-z]+')])),
        'designation': new FormControl(this.emp.designation, Validators.required),
        'email': new FormControl(this.emp.email, Validators.compose(
            [Validators.required, CustomValidator.emailAddressValidator]))
    });
}
```

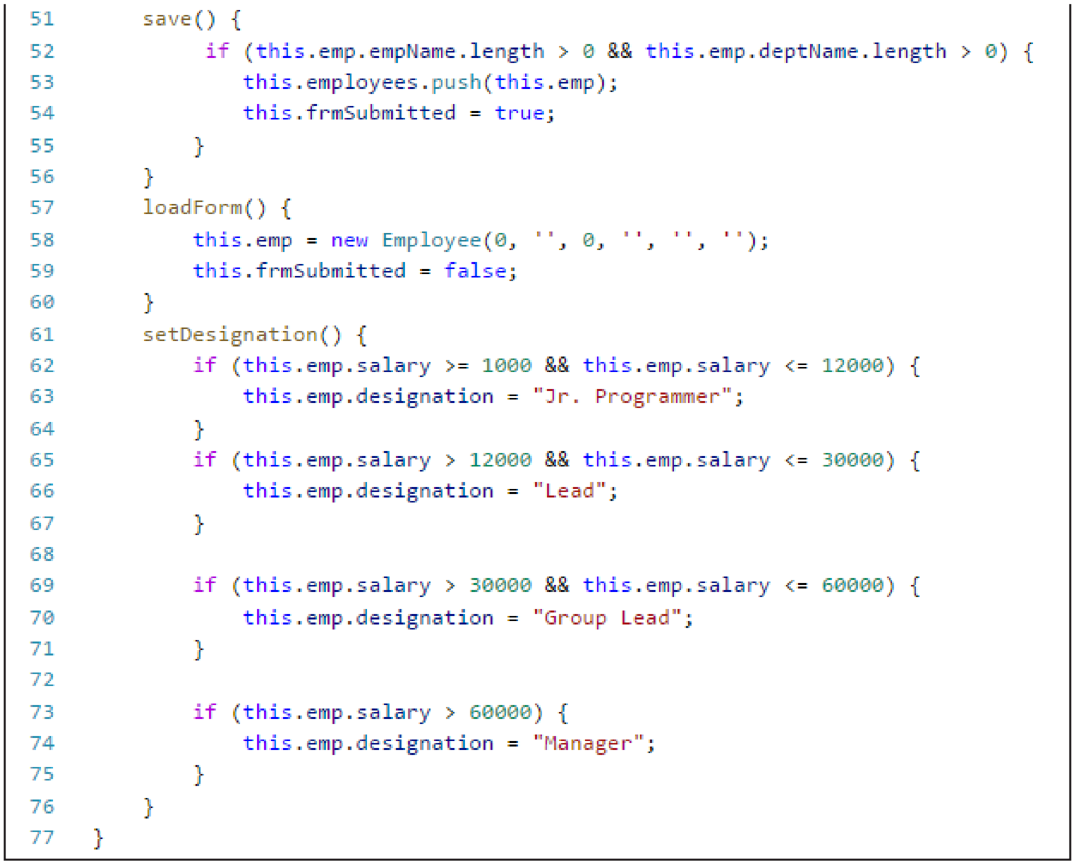
Please note that the code has an array of validation rules to the FormGroup constructor using **Validators.compose()** function. In the code, the required validator and pattern validator rules are used for the fields. The **email** field is validated using the static method **emailAddressValidator** defined in the class CustomValidator.

The entire code of the EmployeeComponent.ts file is as shown in listing 23-6.

Listing 23.6: *The complete code*



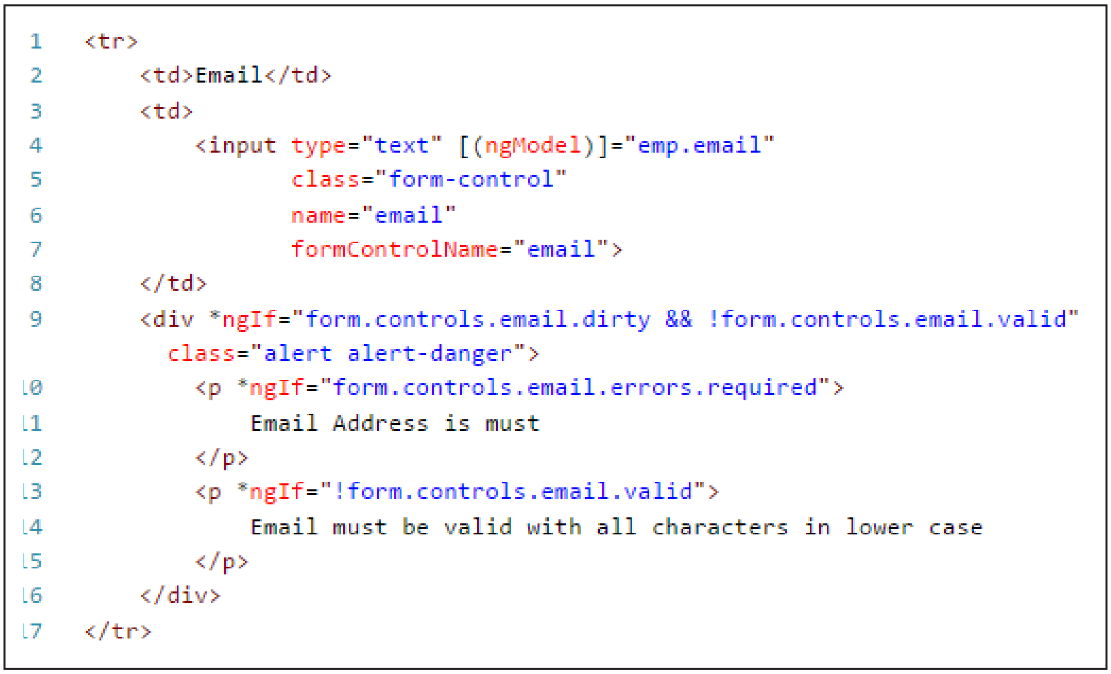
```
import {Component,OnInit} from '@angular/core';
import { FormGroup,FormControl, FormBuilder,Validators } from '@angular/forms';
import {Employee} from './employee.model';
import {CustomValidator} from './validator';

@Component({
    selector: 'emp-data',
    templateUrl: './app/employee.html'
})

export class EmployeeComponent {
    form: FormGroup;
    emp: Employee;
    empNo: FormControl;
    empName: FormControl;
    salary: FormControl;
    deptName: FormControl;
    designation: FormControl;
    email: FormControl;

    employees: Employee[];
    helpLink: string;
    helpText: string;
    frmSubmitted: boolean;
    pattern: RegExp;

    constructor(fEmp: FormBuilder) {
        this.emp = new Employee(0, '', 0, '', '', '');

        this.employees = [];
        this.helpText = "Tax Rules";
        this.helpLink = "./app/taxhelper.html";
        this.frmSubmitted = false;
    }

    ngOnInit(){
            this.form = new FormGroup({
            'empNo': new FormControl(this.emp.empNo, Validators.compose(
                [Validators.required, Validators.pattern('[0-9]+')])),
        'empName': new FormControl(this.emp.empName, Validators.compose(
            [Validators.required, Validators.minLength(2), Validators.maxLength(16)])
            ),
        'salary': new FormControl(this.emp.salary, Validators.compose(
            [Validators.required, Validators.pattern('[0-9]+')])),
        'deptName': new FormControl(this.emp.deptName, Validators.compose(
            [Validators.required, Validators.pattern('[A-Za-z]+')])),
        'designation': new FormControl(this.emp.designation, Validators.required),
        'email': new FormControl(this.emp.email, Validators.compose(
            [Validators.required, CustomValidator.emailAddressValidator]))

        });
    }

    clear() {
        this.emp = new Employee(0, '', 0, '', '', '');
    }
```



```
    save() {
         if (this.emp.empName.length > 0 && this.emp.deptName.length > 0) {
            this.employees.push(this.emp);
            this.frmSubmitted = true;
        }
    }
    loadForm() {
        this.emp = new Employee(0, '', 0, '', '', '');
        this.frmSubmitted = false;
    }
    setDesignation() {
        if (this.emp.salary >= 1000 && this.emp.salary <= 12000) {
            this.emp.designation = "Jr. Programmer";
        }
        if (this.emp.salary > 12000 && this.emp.salary <= 30000) {
            this.emp.designation = "Lead";
        }

        if (this.emp.salary > 30000 && this.emp.salary <= 60000) {
            this.emp.designation = "Group Lead";
        }

        if (this.emp.salary > 60000) {
            this.emp.designation = "Manager";
        }
    }
}
```

To experience validation rules on the UI, modify the Email field and its validation rule of Employee.html as shown in listing 23-7.

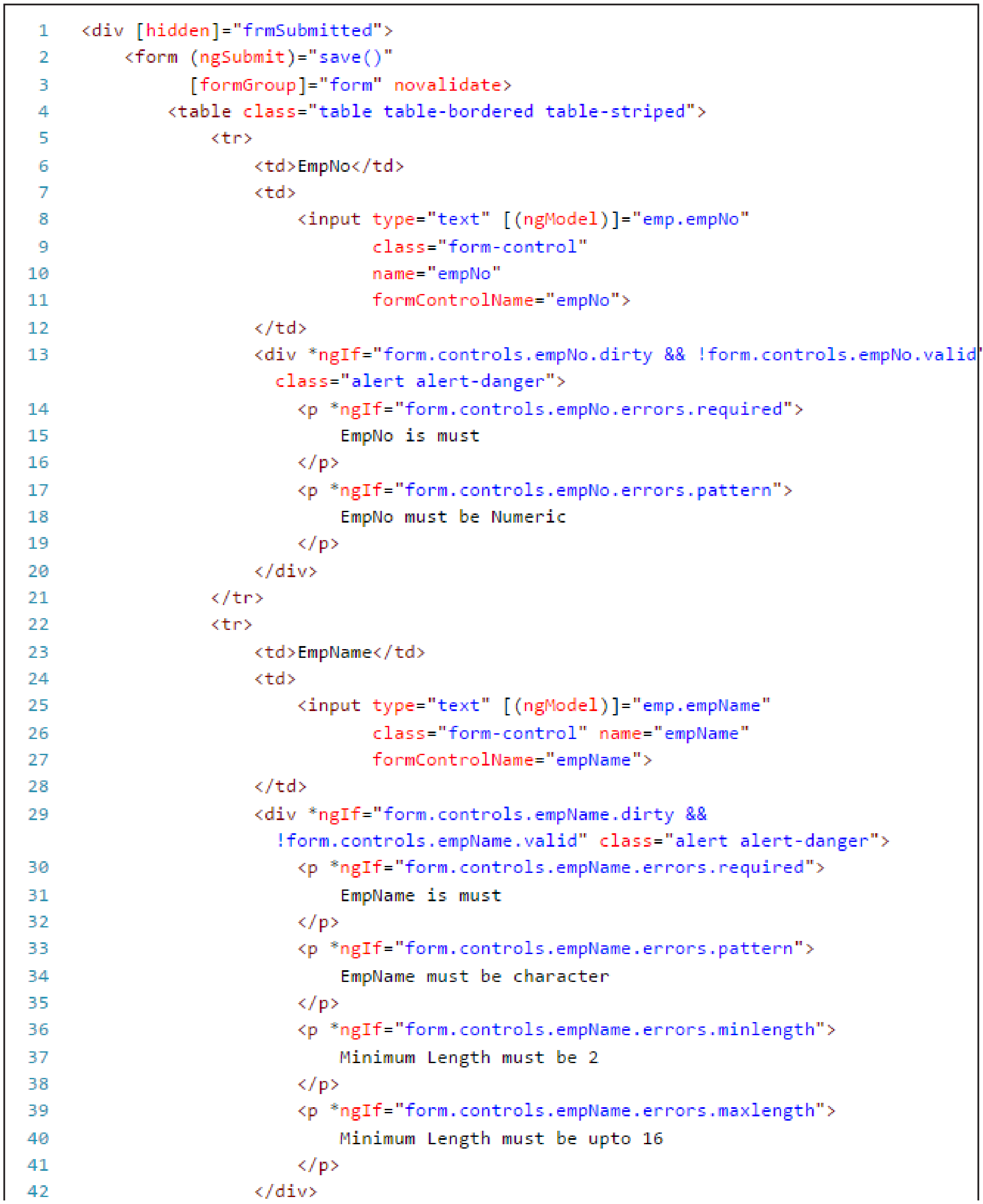
Listing 23.7: *Using Custom validator in Html*



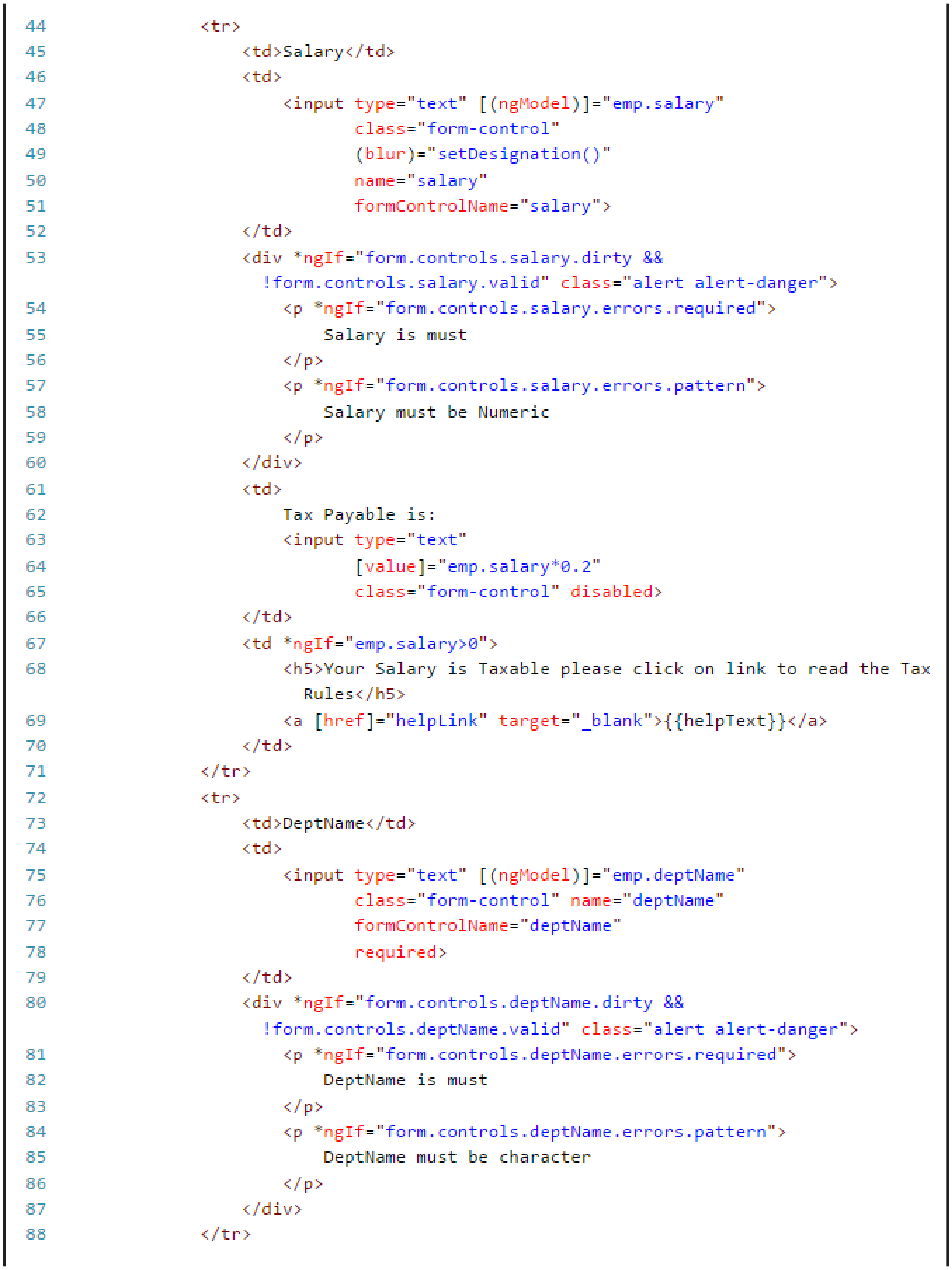
```
<tr>
    <td>Email</td>
    <td>
        <input type="text" [(ngModel)]="emp.email"
               class="form-control"
               name="email"
               formControlName="email">
    </td>
    <div *ngIf="form.controls.email.dirty && !form.controls.email.valid"
      class="alert alert-danger">
        <p *ngIf="form.controls.email.errors.required">
            Email Address is must
        </p>
        <p *ngIf="!form.controls.email.valid">
            Email must be valid with all characters in lower case
        </p>
    </div>
</tr>
```

The entire code of Employee.html is shown in listing 23-8.

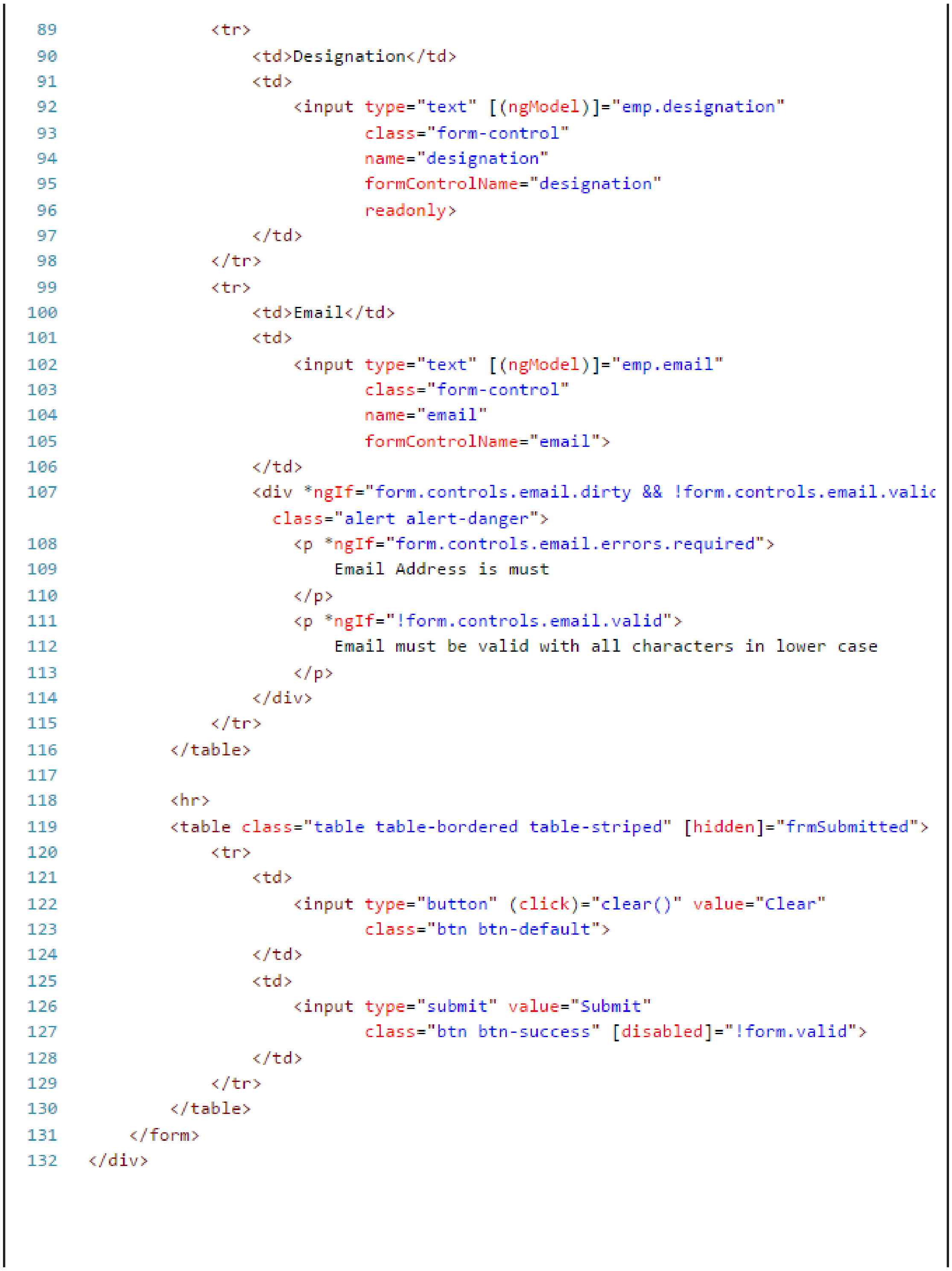
Listing 23.8: *The complete html code*



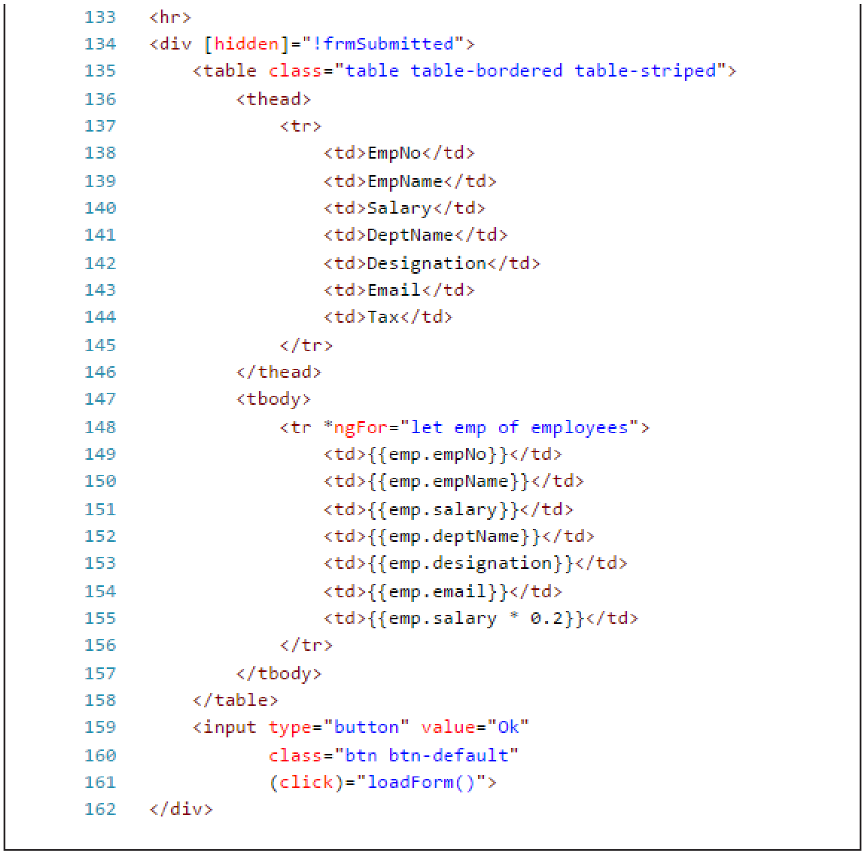
```
<div [hidden]="frmSubmitted">
    <form (ngSubmit)="save()"
          [formGroup]="form" novalidate>
        <table class="table table-bordered table-striped">
            <tr>
                <td>EmpNo</td>
                <td>
                    <input type="text" [(ngModel)]="emp.empNo"
                           class="form-control"
                           name="empNo"
                           formControlName="empNo">
                </td>
                <div *ngIf="form.controls.empNo.dirty && !form.controls.empNo.valid"
                  class="alert alert-danger">
                    <p *ngIf="form.controls.empNo.errors.required">
                        EmpNo is must
                    </p>
                    <p *ngIf="form.controls.empNo.errors.pattern">
                        EmpNo must be Numeric
                    </p>
                </div>
            </tr>
            <tr>
                <td>EmpName</td>
                <td>
                    <input type="text" [(ngModel)]="emp.empName"
                           class="form-control" name="empName"
                           formControlName="empName">
                </td>
                <div *ngIf="form.controls.empName.dirty &&
                  !form.controls.empName.valid" class="alert alert-danger">
                    <p *ngIf="form.controls.empName.errors.required">
                        EmpName is must
                    </p>
                    <p *ngIf="form.controls.empName.errors.pattern">
                        EmpName must be character
                    </p>
                    <p *ngIf="form.controls.empName.errors.minlength">
                        Minimum Length must be 2
                    </p>
                    <p *ngIf="form.controls.empName.errors.maxlength">
                        Minimum Length must be upto 16
                    </p>
                </div>
```



```
                <tr>
                    <td>Salary</td>
                    <td>
                        <input type="text" [(ngModel)]="emp.salary"
                               class="form-control"
                               (blur)="setDesignation()"
                               name="salary"
                               formControlName="salary">
                    </td>
                    <div *ngIf="form.controls.salary.dirty && !form.controls.salary.valid" class="alert alert-danger">
                        <p *ngIf="form.controls.salary.errors.required">
                            Salary is must
                        </p>
                        <p *ngIf="form.controls.salary.errors.pattern">
                            Salary must be Numeric
                        </p>
                    </div>
                    <td>
                        Tax Payable is:
                        <input type="text"
                               [value]="emp.salary*0.2"
                               class="form-control" disabled>
                    </td>
                    <td *ngIf="emp.salary>0">
                        <h5>Your Salary is Taxable please click on link to read the Tax Rules</h5>
                        <a [href]="helpLink" target="_blank">{{helpText}}</a>
                    </td>
                </tr>
                <tr>
                    <td>DeptName</td>
                    <td>
                        <input type="text" [(ngModel)]="emp.deptName"
                               class="form-control" name="deptName"
                               formControlName="deptName"
                               required>
                    </td>
                    <div *ngIf="form.controls.deptName.dirty && !form.controls.deptName.valid" class="alert alert-danger">
                        <p *ngIf="form.controls.deptName.errors.required">
                            DeptName is must
                        </p>
                        <p *ngIf="form.controls.deptName.errors.pattern">
                            DeptName must be character
                        </p>
                    </div>
                </tr>
```



```
            <tr>
                <td>Designation</td>
                <td>
                    <input type="text" [(ngModel)]="emp.designation"
                           class="form-control"
                           name="designation"
                           formControlName="designation"
                           readonly>
                </td>
            </tr>
            <tr>
                <td>Email</td>
                <td>
                    <input type="text" [(ngModel)]="emp.email"
                           class="form-control"
                           name="email"
                           formControlName="email">
                </td>
                <div *ngIf="form.controls.email.dirty && !form.controls.email.valic
                  class="alert alert-danger">
                    <p *ngIf="form.controls.email.errors.required">
                        Email Address is must
                    </p>
                    <p *ngIf="!form.controls.email.valid">
                        Email must be valid with all characters in lower case
                    </p>
                </div>
            </tr>
        </table>

        <hr>
        <table class="table table-bordered table-striped" [hidden]="frmSubmitted">
            <tr>
                <td>
                    <input type="button" (click)="clear()" value="Clear"
                           class="btn btn-default">
                </td>
                <td>
                    <input type="submit" value="Submit"
                           class="btn btn-success" [disabled]="!form.valid">
                </td>
            </tr>
        </table>
    </form>
</div>
```



```
<hr>
<div [hidden]="!frmSubmitted">
    <table class="table table-bordered table-striped">
        <thead>
            <tr>
                <td>EmpNo</td>
                <td>EmpName</td>
                <td>Salary</td>
                <td>DeptName</td>
                <td>Designation</td>
                <td>Email</td>
                <td>Tax</td>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor="let emp of employees">
                <td>{{emp.empNo}}</td>
                <td>{{emp.empName}}</td>
                <td>{{emp.salary}}</td>
                <td>{{emp.deptName}}</td>
                <td>{{emp.designation}}</td>
                <td>{{emp.email}}</td>
                <td>{{emp.salary * 0.2}}</td>
            </tr>
        </tbody>
    </table>
    <input type="button" value="Ok"
           class="btn btn-default"
           (click)="loadForm()">
</div>
```

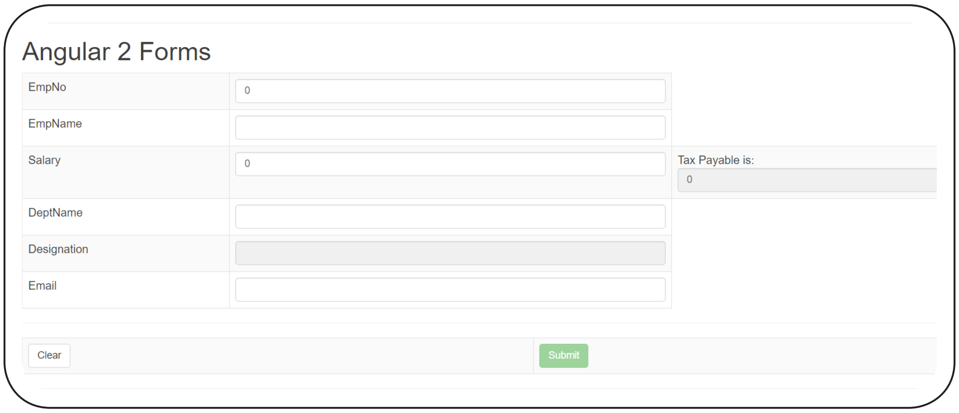
Notice that since the form validation for model properties (like EmpNo, EmpName) is implemented using the FormControl class, and for Validators object using the validation rules in the EmployeeComponent class (as explained in Listing 23-6 in ngOnInit() function)**,** all the validation attributes on DOM elements (like **required** and **pattern)** are removed from the template as seen in listing 23.8.

To run the application, right click on index.html in VS Code and **select Open in Command prompt**. This will open the Command prompt. Run the following command from the command prompt:

```
npm run start
```

This will start the server.

Open the browser and enter the URL http://localhost:3000 to load the form in the browser as shown Figure 23-1.

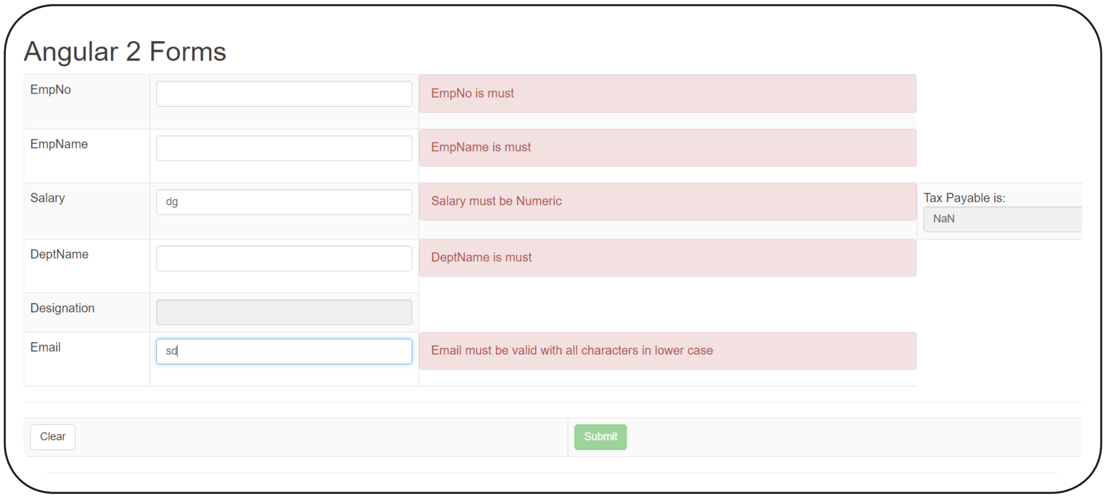


Angular 2 Forms

EmpNo	0	
EmpName		
Salary	0	Tax Payable is: 0
DeptName		
Designation		
Email		

Figure 23.1: *Loading form*

Note that the Submit button is disabled. Enter invalid values in input elements (e.g. delete 0 from EmpNo), enter text in Salary element, etc. and the form will show validation errors as seen in Figure 23-2.



Angular 2 Forms

EmpNo		EmpNo is must	
EmpName		EmpName is must	
Salary	dg	Salary must be Numeric	Tax Payable is: NaN
DeptName		DeptName is must	
Designation			
Email	sd	Email must be valid with all characters in lower case	

Figure 23.2: *Result with validation*

When all valid values are entered, the Submit button is enabled and you can post the values to the server.

The validation rules defined in the **ngOnInit()** function isolates the form model code from the DOM.

Conclusion

Angular 2 provides different ways to validate data in a form. The Form control object has a uniform yet flexible API to validate the values using both built-in and custom validations. This chapter discussed how to write a custom validator using the Form control.

Chapter 24

Template-Driven Forms

THE PREVIOUS CHAPTERS (FROM 20 to 23) explored Angular 2 form basics, Model-Driven forms, standard validations, and Custom validations. The advantage of the Model-Driven form approach is that the form-model is isolated from the DOM, simplifying maintenance and testing.

Why Template Driven Forms?

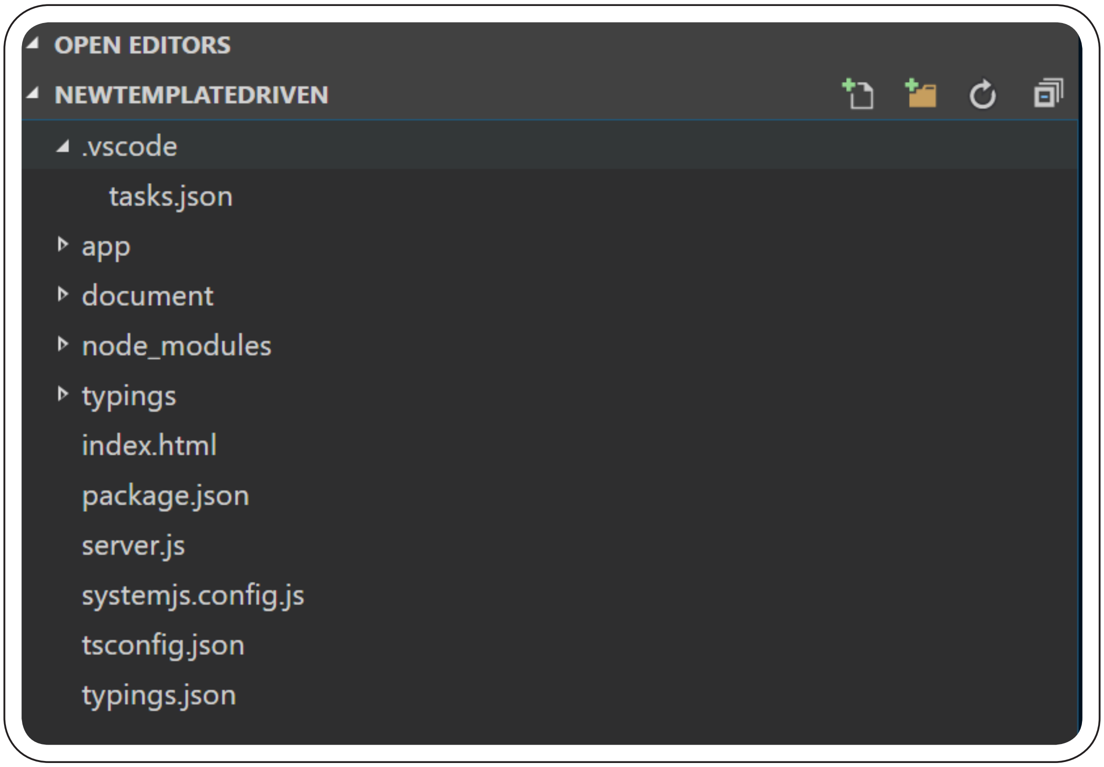
Forms in an Angular 2 application can be developed using another approach called Template-Driven forms. Typically, this approach can be used to build utility forms like login-forms, simple data-entry forms (create-contact forms); those who need a simple model, as well as data-binding and validation rules, and can be accessed repeatedly in the application. This type of form can be developed with a very little code requiredusing an important concept known as Template Reference variables. This is explained in a later section of this chapter.

In Angular 2, Template Driven forms automatically applies form-level directive to the <form> which creates **FromGroup** and links it with the form. Each <input> element in the form will be registered with the control group. Validation attributes e.g. required, pattern, etc. can be applied on these <input> elements for validations.

Furthermore, by using [(ngModel)], the databinding between model and form element can also be implemented, hence the application code (the component code) need not contain complex logic. Since, the validation, binding, etc. is implemented in a declarative way, they are known as **Template Driven Forms.**

To implement a Template-Driven form, create a new project using Visual Studio Code. Follow all steps given in Chapter 5—Angular 2 Development Environment.

The Project will have a project structure as shown in Figure 24-1:

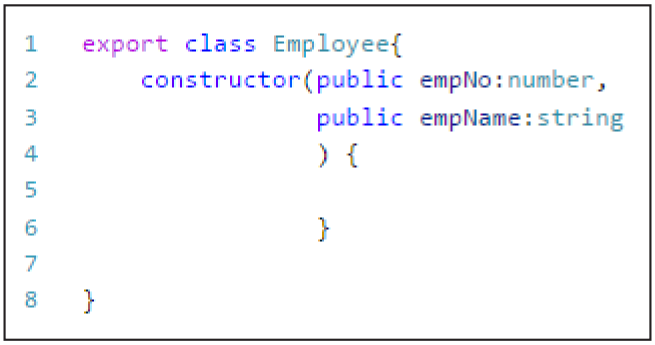


OPEN EDITORS
NEWTEMPLATEDRIVEN
.vscode
tasks.json
app
document
node_modules
typings
index.html
package.json
server.js
systemjs.config.js
tsconfig.json
typings.json

Figure 24.1: *The Project structure*

In the app folder, add a new file of the name employee.model.ts with the code as shown in listing 24-1:

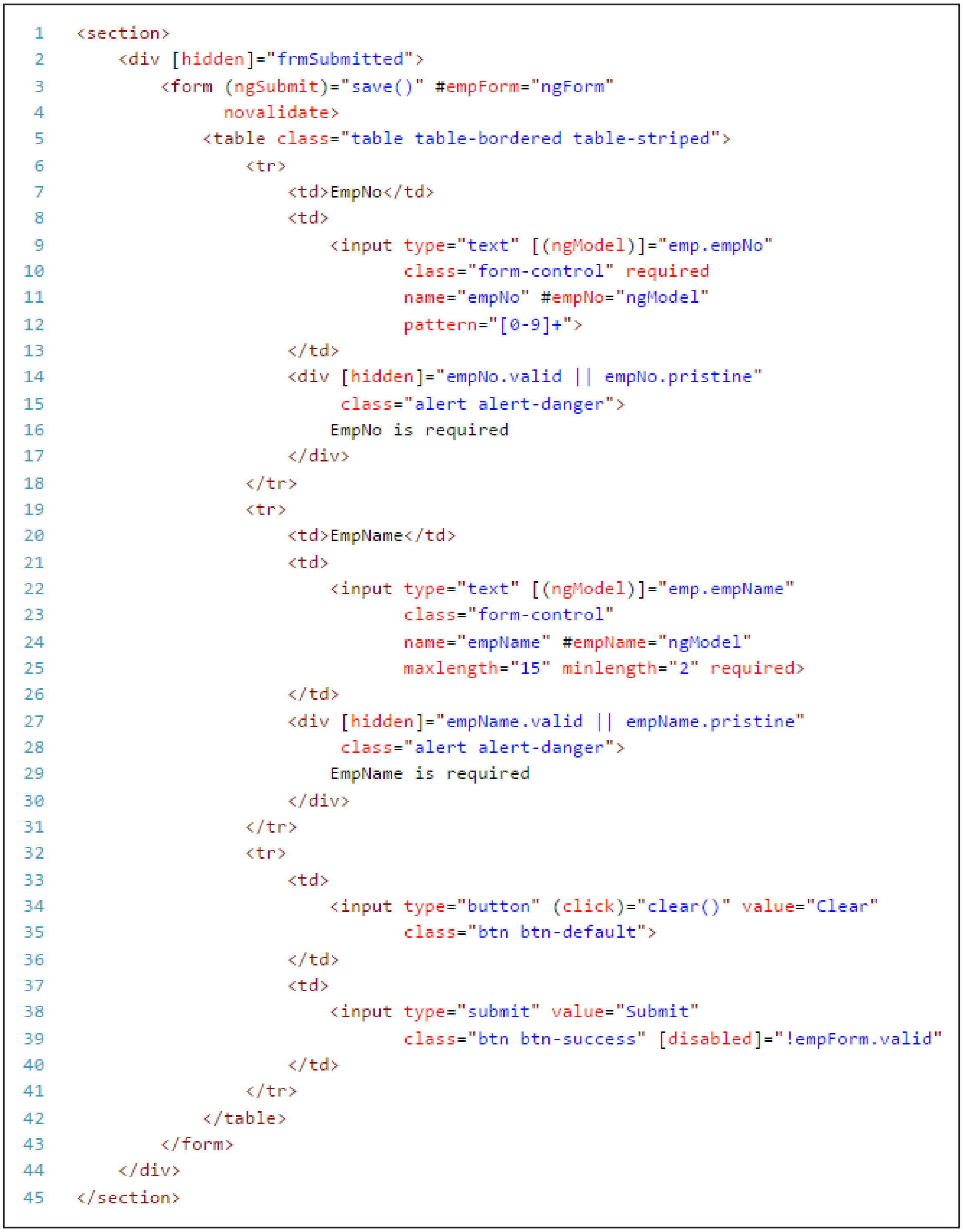
Listing 24.1: *The Model class*



```
export class Employee{
    constructor(public empNo:number,
                public empName:string
                ) {

                }

}
```

Now in the same app folder, add a new file of name employee.html with HTML markup as shown in Listing 24.2.

Listing 24.2: *Html markup with form*



```
<section>
    <div [hidden]="frmSubmitted">
        <form (ngSubmit)="save()" #empForm="ngForm"
              novalidate>
            <table class="table table-bordered table-striped">
                <tr>
                    <td>EmpNo</td>
                    <td>
                        <input type="text" [(ngModel)]="emp.empNo"
                               class="form-control" required
                               name="empNo" #empNo="ngModel"
                               pattern="[0-9]+">
                    </td>
                    <div [hidden]="empNo.valid || empNo.pristine"
                         class="alert alert-danger">
                        EmpNo is required
                    </div>
                </tr>
                <tr>
                    <td>EmpName</td>
                    <td>
                        <input type="text" [(ngModel)]="emp.empName"
                               class="form-control"
                               name="empName" #empName="ngModel"
                               maxlength="15" minlength="2" required>
                    </td>
                    <div [hidden]="empName.valid || empName.pristine"
                         class="alert alert-danger">
                        EmpName is required
                    </div>
                </tr>
                <tr>
                    <td>
                        <input type="button" (click)="clear()" value="Clear"
                               class="btn btn-default">
                    </td>
                    <td>
                        <input type="submit" value="Submit"
                               class="btn btn-success" [disabled]="!empForm.valid"
                    </td>
                </tr>
            </table>
        </form>
    </div>
</section>
```

The above code uses a new syntax for defining references to an instance for ngForm i.e. #empForm, and for ngModel it is #empNo. This is a concept of **Template Reference Variable.**

Template Reference Variable

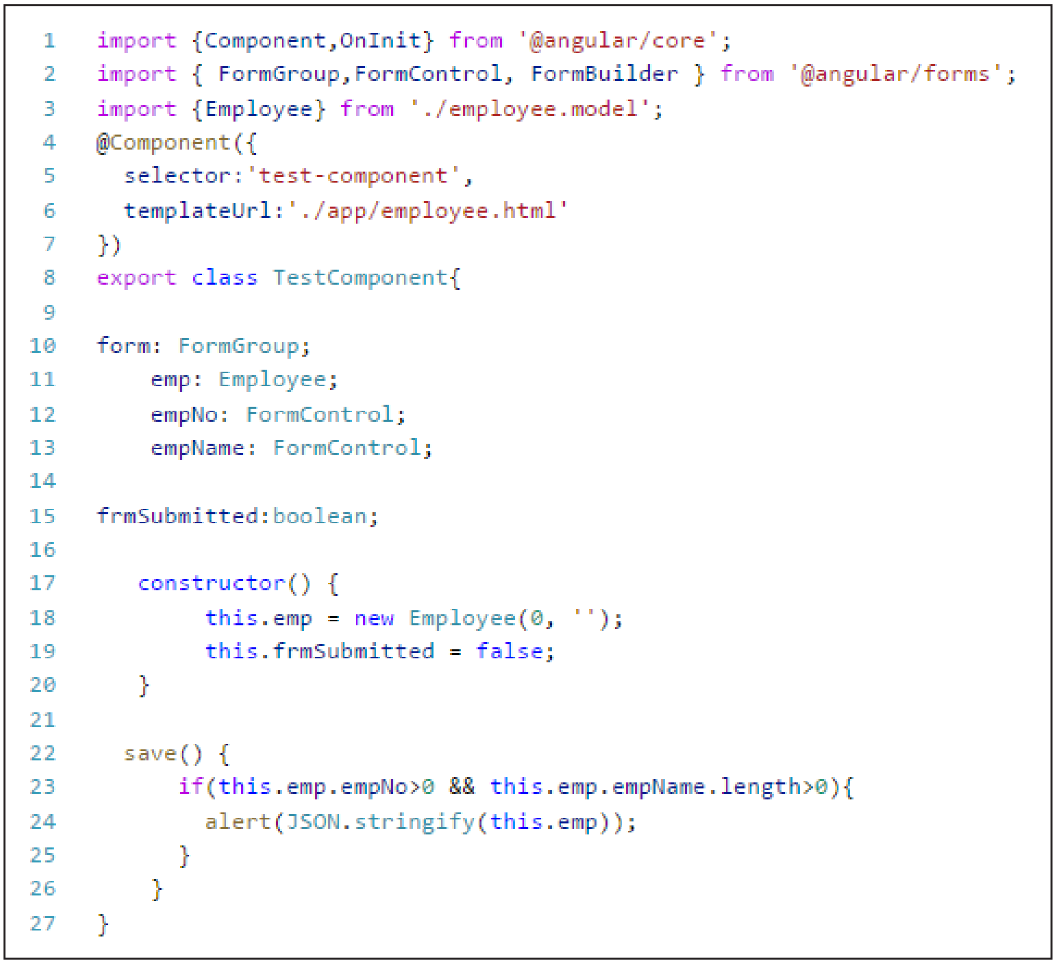
Angular supports creating variables for the elements on the page; these variables are called template reference variables. This object contains the DOM object of the element on which it is applied. The advantage of template reference variable is that it can be referred from anywhere in the template. The variable's value is set to the element on which it is defined. The other syntax for defining the Template Reference variable is using **ref-** prefix, e.g. the #empNo can also be defined as ref-empNo as shown in the following snippet.

```
<input ref-empNo placeholder="Emp No"/>
```

Using the Template Reference Variable #empForm, the ngForm directive is exposed and accessed with its reference. This can further be used for tracking validity of the input.

In the project, add another file in the same app folder with the name **template.component.ts.** Add the following code in it:

Listing 24.3: *The component code*

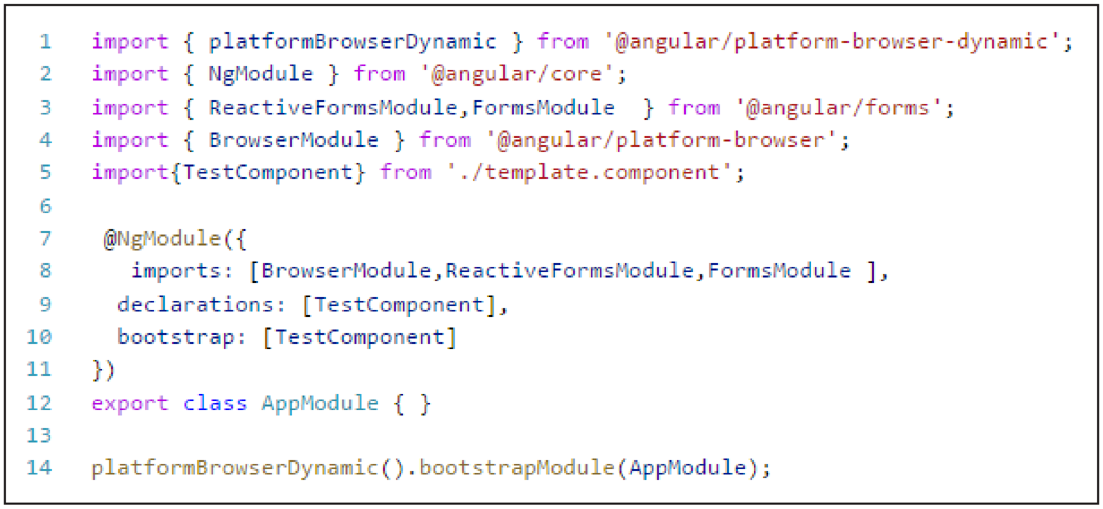


```
import {Component,OnInit} from '@angular/core';
import { FormGroup,FormControl, FormBuilder } from '@angular/forms';
import {Employee} from './employee.model';
@Component({
  selector:'test-component',
  templateUrl:'./app/employee.html'
})
export class TestComponent{

form: FormGroup;
    emp: Employee;
    empNo: FormControl;
    empName: FormControl;

frmSubmitted:boolean;

   constructor() {
        this.emp = new Employee(0, '');
        this.frmSubmitted = false;
   }

  save() {
      if(this.emp.empNo>0 && this.emp.empName.length>0){
        alert(JSON.stringify(this.emp));
      }
    }
}
```

The Template-Driven form simply uses the form variable object (**#empForm)** for performing form-level validations.

In the app folder, add main.ts and write the following code as shown in listing 24-3.

Listing 24.3: *The NgModule*

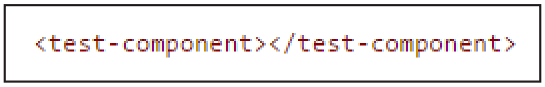


```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { ReactiveFormsModule,FormsModule  } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';
import{TestComponent} from './template.component';

 @NgModule({
    imports: [BrowserModule,ReactiveFormsModule,FormsModule ],
   declarations: [TestComponent],
   bootstrap: [TestComponent]
})
export class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

This will import the TestComponent class.

Make the changes in body tag of index.html as shown in listing 24-4

Listing 24.4: *Using template url as html tag in index.html*



```
<test-component></test-component>
```

This step renders the html template.

To run the application, right click on index.html in VS Code and select **Open in Command Prompt**. Enter the following command on the Command prompt

```
npm run start
```

In the browser enter the URL:

http://localhost:3000

The following result will be displayed:

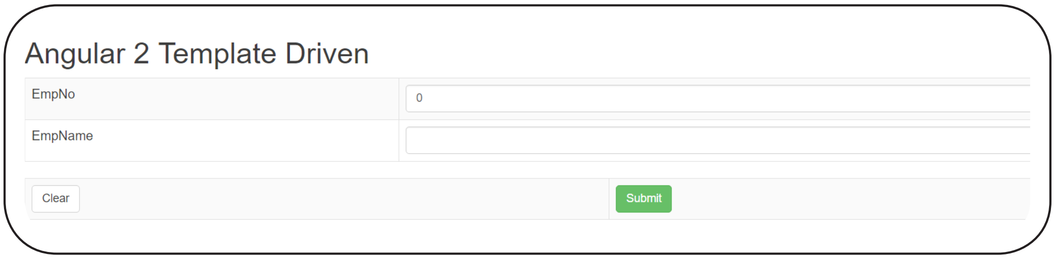
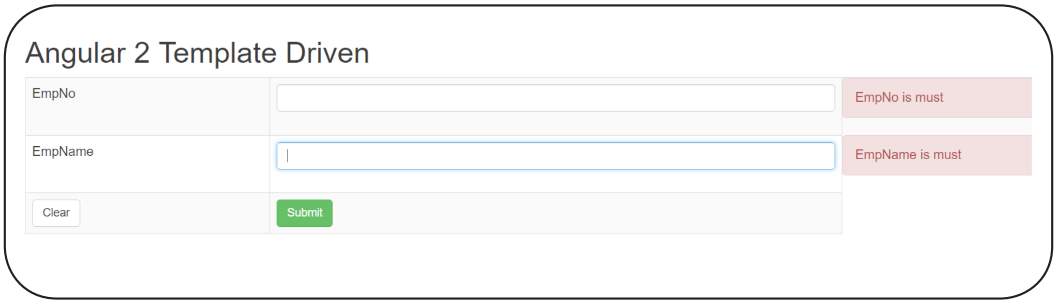


Figure 24.2: *Loading the template-driven form*

To test the validations, remove default value of EmpNo in the TextBox and enter some text in EmpName TextBox. Now remove the text and the following result will be displayed instantaneously:



EmpNo is must

EmpName is must

Figure 24.3: *Execution of the template-driven from with validation*

Conclusion

The advantage of template-driven forms is that the approach is very simple and very familiar to Angular 1 developers. This article demonstrated how to build template-driven forms with validation using the latest forms module.

Chapter 25 Services

SO FAR, WE HAVE covered Angular 2 Components, Directives, and Forms. As an application starts using these features and keeps growing in size, the development team starts looking for options to separate the business logic out of these UI pieces. The can be used anywhere across the application once it is out of any component or directive.

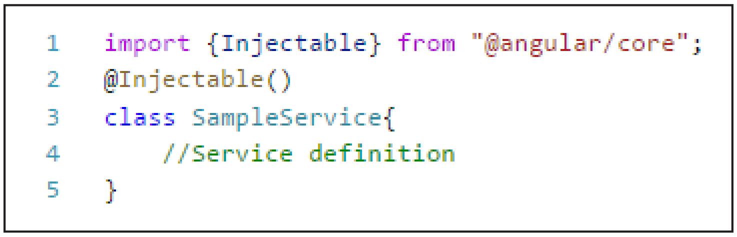
Angular 2 has *Services* for this purpose.

Service is an overloaded term in software development. The term "service" is basically a piece of code that holds some common logic and is available to any other piece. In Angular 2, a service is a TypeScript class. It can be injected into any component, directive, or service and can be used whenever needed.

Services in Angular 2

Any TypeScript class can be used as a service in Angular 2. Angular 2 has a decorator *Injectable,* to be used with services when metadata about the service has to be emitted. Even if not needed, it is recommended to apply this decorator to make it future-proof. The following snippet shows the syntax of using *Injectable* on a service:

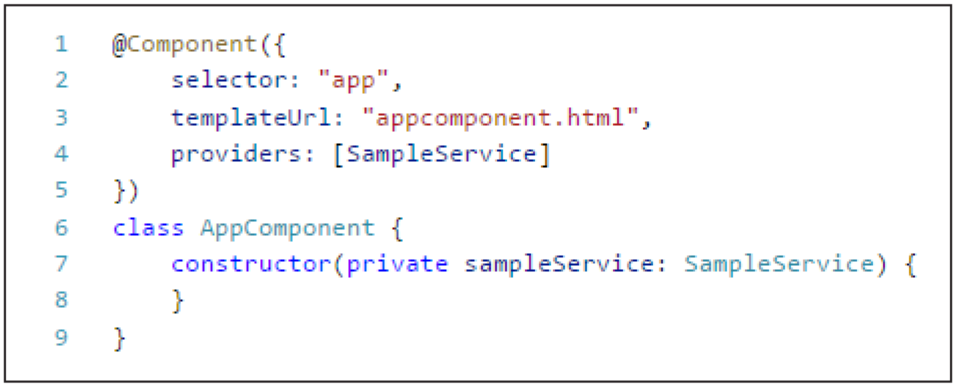
Listing 25.1: *Syntax of creating a service*



```
import {Injectable} from "@angular/core";
@Injectable()
class SampleService{
    //Service definition
}
```

To use this service in a component, it must be declared in the *providers* property of the *Component* annotation. Snippet 25.2 shows how to declare a service in a component and inject it:

Listing 25.2: *Syntax of creating a component*

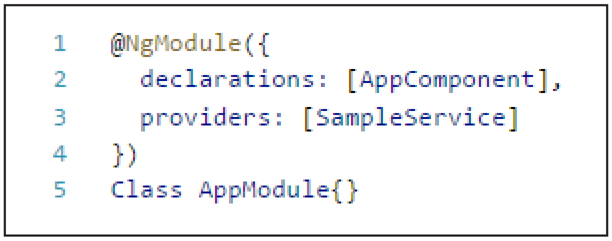


```
@Component({
    selector: "app",
    templateUrl: "appcomponent.html",
    providers: [SampleService]
})
class AppComponent {
    constructor(private sampleService: SampleService) {
    }
}
```

The *providers* array can have any number of services declared in it. Now the service *SampleService* can be used by any code block that loads under the subtree of the *AppComponent*.

Alternatively, it can be specified in the *providers* block of the module to make it available to the entire module. Listing 25.3 shows this:

Listing 25.3: *Declaring a service in a module*



```
@NgModule({
  declarations: [AppComponent],
  providers: [SampleService]
})
Class AppModule{}
```

TypeScript is being used throughout this book, so there is no need to specify anything other than type of the service for dependency injection. This is because TypeScript type is used as the token for dependency injection. When the TypeScript code is converted to JavaScript, it creates the DI annotations for all dependencies using the type assigned to the injected objects.

Creating and Using a Service

Let's create a books list service. The application consists of a component to which a user can add a book, mark the book as *read*, and delete the book. The component will load a static list of book; the service will hold the static list of books, manage the tasks of marking books as read, and delete books from the list. The component will call these methods of the service based on the user's action on the page.

To setup the project, use the starter project created in Chapter 5. To this, install the @angular/forms package using the following command:

```
npm install @angular/forms --save
```

This package has to be registered with SystemJS to be able to refer it in the application. Add the following entry to the *map* configuration in the file *systemjs.config.js:*

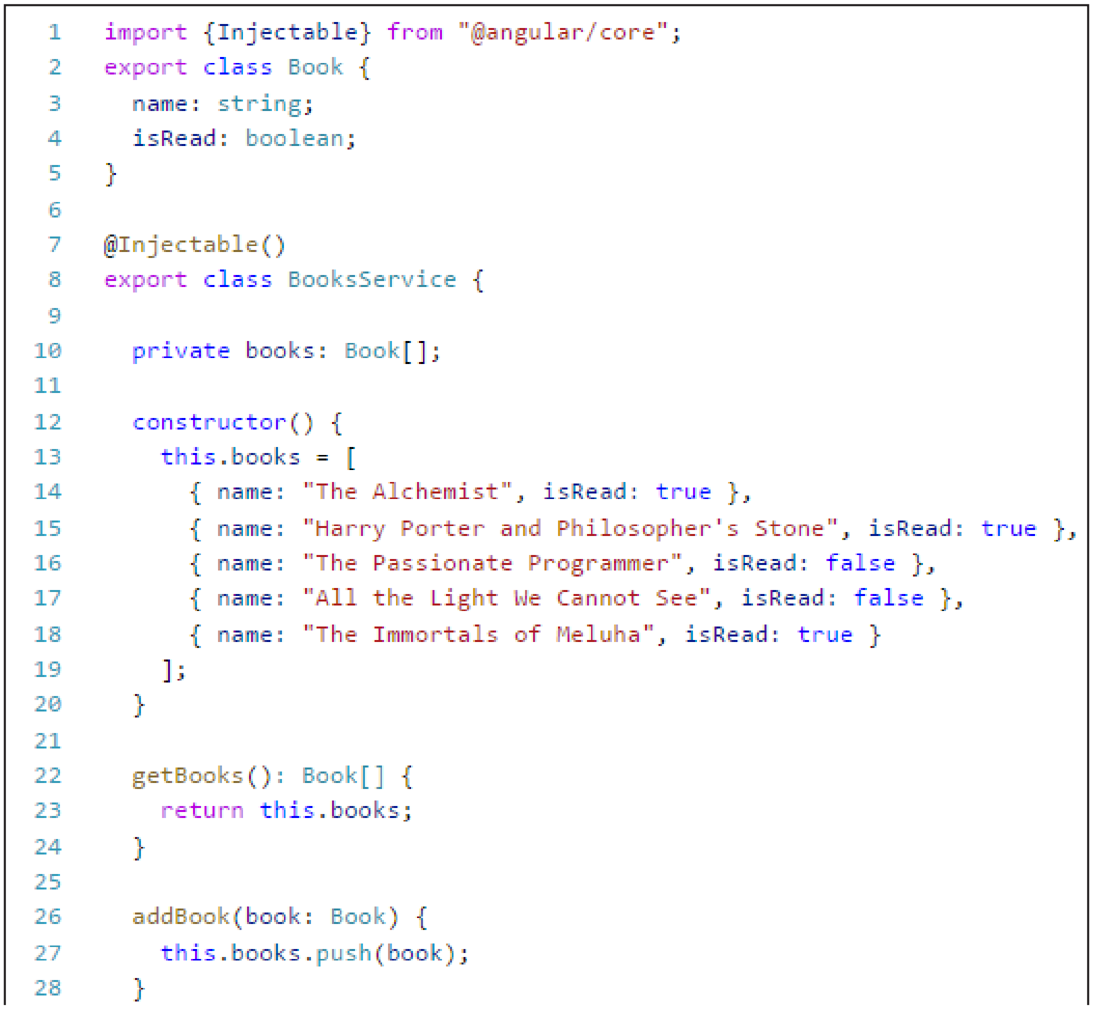
```
"@angular/forms": "node_modules/@angular/forms"
```

And the following entry to the *packages* section in the file *systemjs.config.js*:

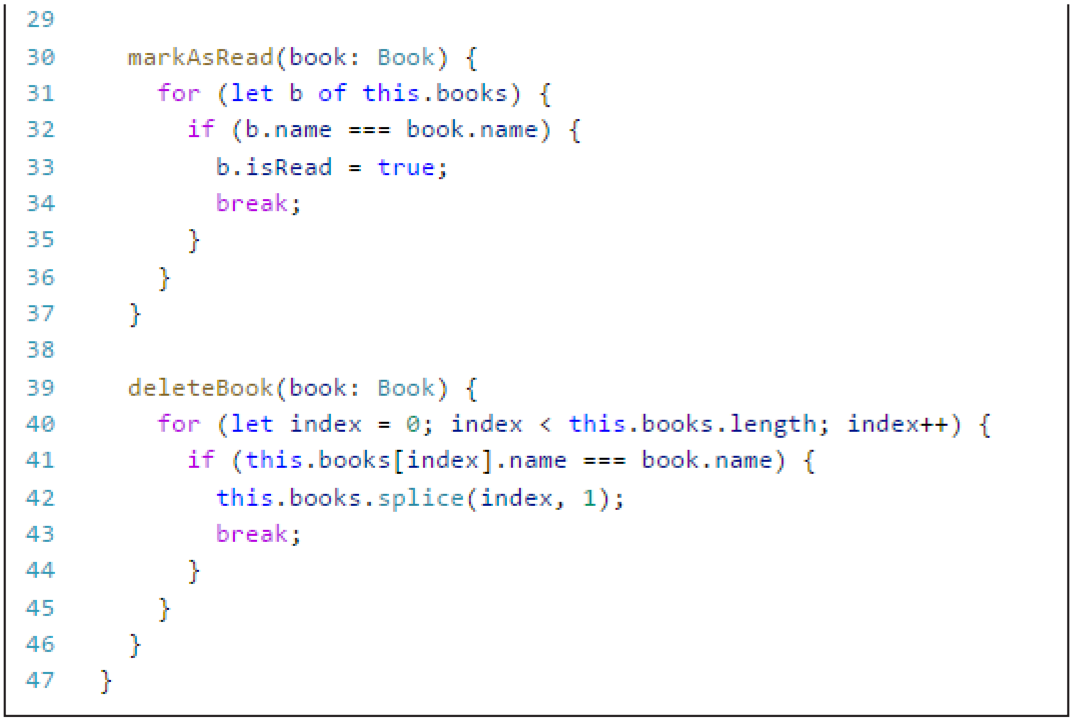
```
"@angular/forms": { "main": "bundles/forms.umd.js", "defaultExtension": "js" }
```

With this, the setup is ready and we can start building the application. First, let’s define the service with the methods described above. Snippet 25.4 shows the service:

Listing 25.4: *Code of model class for Book and the BooksService*



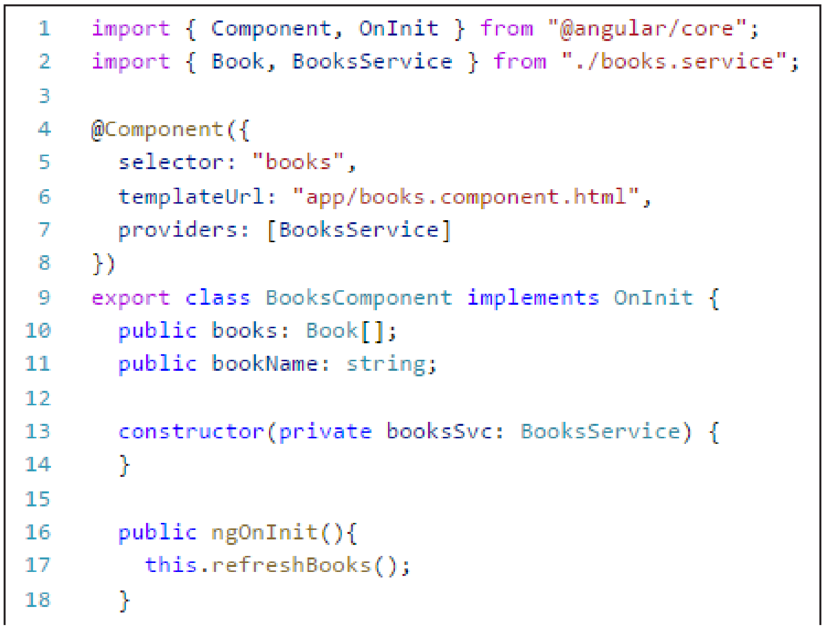
```
import {Injectable} from "@angular/core";
export class Book {
  name: string;
  isRead: boolean;
}

@Injectable()
export class BooksService {

  private books: Book[];

  constructor() {
    this.books = [
      { name: "The Alchemist", isRead: true },
      { name: "Harry Porter and Philosopher's Stone", isRead: true },
      { name: "The Passionate Programmer", isRead: false },
      { name: "All the Light We Cannot See", isRead: false },
      { name: "The Immortals of Meluha", isRead: true }
    ];
  }

  getBooks(): Book[] {
    return this.books;
  }

  addBook(book: Book) {
    this.books.push(book);
  }
```



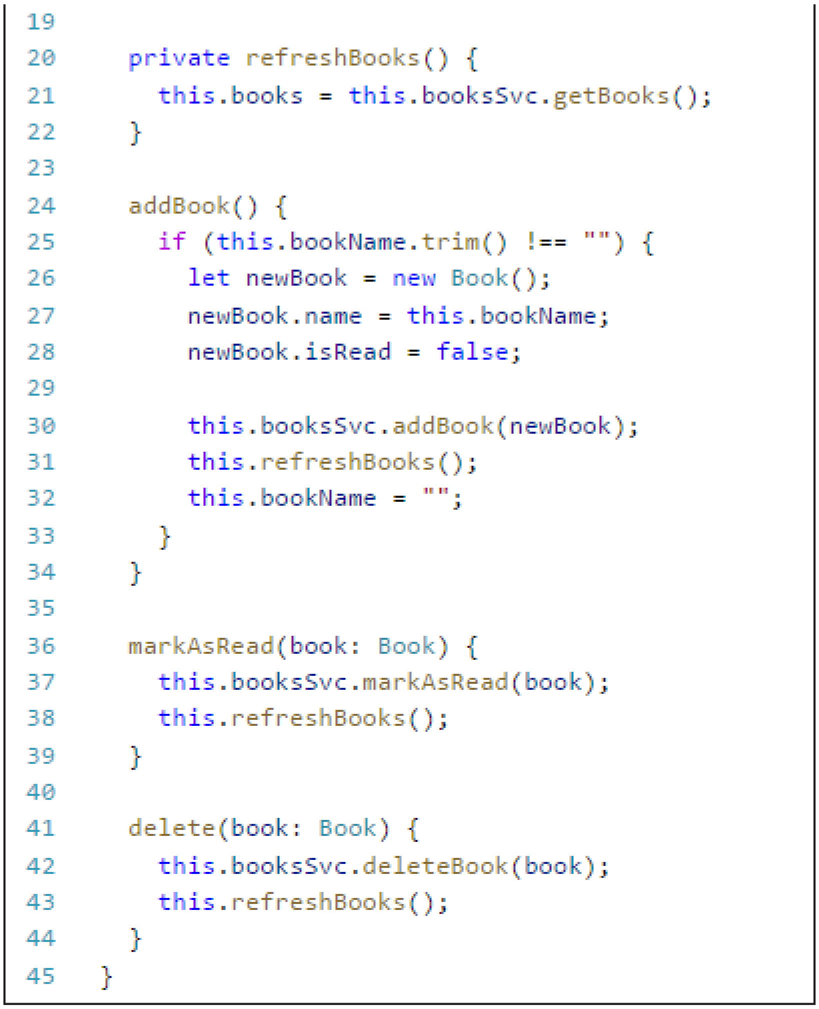
```
  markAsRead(book: Book) {
    for (let b of this.books) {
      if (b.name === book.name) {
        b.isRead = true;
        break;
      }
    }
  }

  deleteBook(book: Book) {
    for (let index = 0; index < this.books.length; index++) {
      if (this.books[index].name === book.name) {
        this.books.splice(index, 1);
        break;
      }
    }
  }
}
```

The component is straight forward. It consumes the service seen in Listing 25.4 and exposes the functionalities of the service to the view through its wrappers around the service methods. The component loads the books in the *ngOnInit* lifecycle hook and, in every operation performed from the UI, the books list is refreshed from the service. Snippet 25.5 shows the code of the *BooksComponent*:

Listing 25.5: *Books list component to be added to the file books.component.ts*



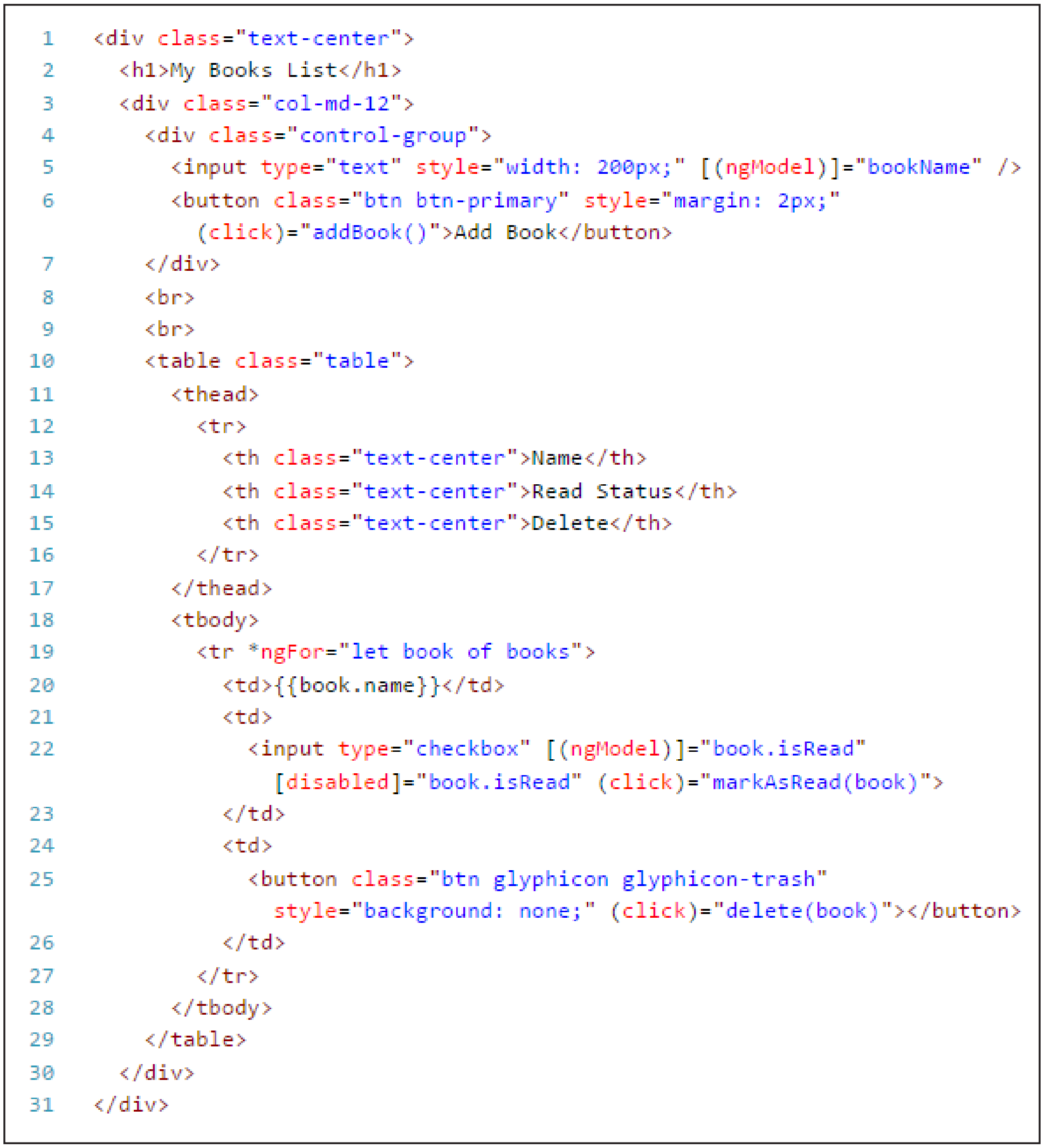
```
import { Component, OnInit } from "@angular/core";
import { Book, BooksService } from "./books.service";

@Component({
  selector: "books",
  templateUrl: "app/books.component.html",
  providers: [BooksService]
})
export class BooksComponent implements OnInit {
  public books: Book[];
  public bookName: string;

  constructor(private booksSvc: BooksService) {
  }

  public ngOnInit(){
    this.refreshBooks();
  }
```



```

  private refreshBooks() {
    this.books = this.booksSvc.getBooks();
  }

  addBook() {
    if (this.bookName.trim() !== "") {
      let newBook = new Book();
      newBook.name = this.bookName;
      newBook.isRead = false;

      this.booksSvc.addBook(newBook);
      this.refreshBooks();
      this.bookName = "";
    }
  }

  markAsRead(book: Book) {
    this.booksSvc.markAsRead(book);
    this.refreshBooks();
  }

  delete(book: Book) {
    this.booksSvc.deleteBook(book);
    this.refreshBooks();
  }
}
```

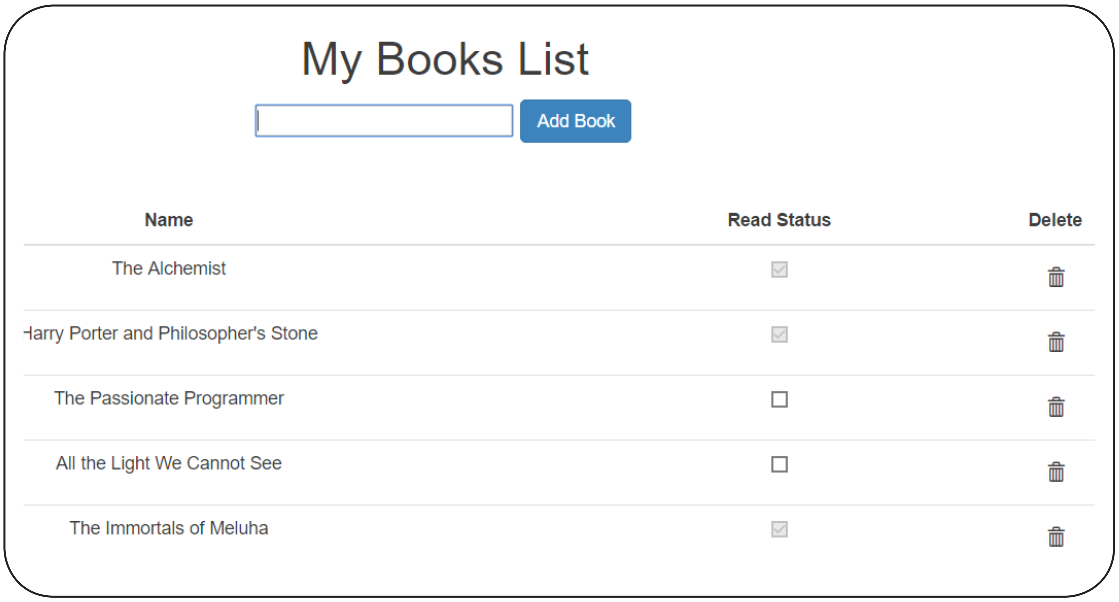
The view of the component consists of a textbox, a button to add a new book, and a table that lists the books in the service. Every row in the table has a checkbox to mark a book as read. The checkbox is disabled with a button to delete the book after a book is marked as read (see Figure 25.1). The view uses Angular 2's bindings to bind the members in the component to different actions and controls. Snippet 25.6 shows the markup of the view:

Listing 25.6: *Template of BooksComponent to be added to the file books.component.html*



```
<div class="text-center">
  <h1>My Books List</h1>
  <div class="col-md-12">
    <div class="control-group">
      <input type="text" style="width: 200px;" [(ngModel)]="bookName" />
      <button class="btn btn-primary" style="margin: 2px;"
        (click)="addBook()">Add Book</button>
    </div>
    <br>
    <br>
    <table class="table">
      <thead>
        <tr>
          <th class="text-center">Name</th>
          <th class="text-center">Read Status</th>
          <th class="text-center">Delete</th>
        </tr>
      </thead>
      <tbody>
        <tr *ngFor="let book of books">
          <td>{{book.name}}</td>
          <td>
            <input type="checkbox" [(ngModel)]="book.isRead"
              [disabled]="book.isRead" (click)="markAsRead(book)">
          </td>
          <td>
            <button class="btn glyphicon glyphicon-trash"
              style="background: none;" (click)="delete(book)"></button>
          </td>
        </tr>
      </tbody>
    </table>
  </div>
</div>
```

When you run the application, you will see a list of books on the screen. You can add new books, mark the existing books as read, and delete books. Figure 25.1 shows how the page looks on the browser:



My Books List

Add Book

Name	Read Status	Delete
The Alchemist	☑	🗑
Harry Porter and Philosopher's Stone	☑	🗑
The Passionate Programmer	□	🗑
All the Light We Cannot See	□	🗑
The Immortals of Meluha	☑	🗑

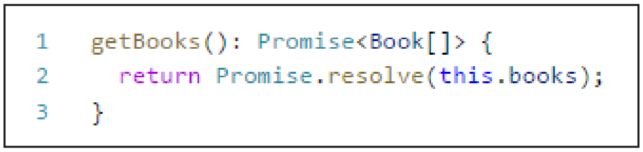
Figure 25.1: *Screen shot of books list*

Using Asynchronous Services

A common pattern across most of the enterprise applications is that the data of the application comes from a server and the call to a deployed back-end service is always handled asynchronously. Even when the services are not ready, a good practice is to follow asynchronous approach with data, so that a lot of code doesn't need to change when the backend services are ready. Let's refactor the methods of the *BooksService* to make them asynchronous.

As the sample uses in-memory data, it can resolve the value immediately using an ES6 promise to mimic the asynchronous behavior. Snippet 25.7 shows the asynchronous version of the *getBooks* method:

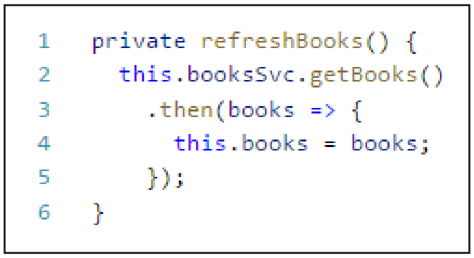
Listing 25.7: *Asynchronous version of getBooks method*



```
getBooks(): Promise<Book[]> {
  return Promise.resolve(this.books);
}
```

The value returned from this method cannot be assigned to the *books* array in the component. The value is made available after the *promise* is resolved, and can be assigned in the success callback of the promise. Snippet 25.8 shows the modified *refreshBooks* method of the component:

Listing 25.8: *refreshBook method consuming async getBooks method*



```
private refreshBooks() {
  this.booksSvc.getBooks()
    .then(books => {
      this.books = books;
    });
}
```

After making these changes, the application runs as it previously ran. Though the sample uses ES6 promises here, because of zones in Angular 2, the change detector need not be called explicitly. As soon as the asynchronous operation is completed, the zone kicks in the change detection process and the UI is updated according to the new data.

You may refer to the sample code of this chapter to see how the other methods are modified to follow asynchronous syntax.

Conclusion

Services define some important parts of an Angular 2 application and can be used to hold the business logic, data access logic, and anything else which is independent of the UI of the application. This chapter demonstrated simple services. Future chapters will cover more advanced features of Services.

Chapter 26

Dependency Injection Overview

ANY COMPLEX APPLICATION MUST follow a set of design patterns and obey the principles of object-oriented design (OOD) to make the code easy to read, maintain, and extend. Though JavaScript is not an object- oriented programming language, it uses objects quite extensively. So, most of the principles followed in the OOPs languages can still be applied to JavaScript. The code of an application written in JavaScript would be extensible and easy to understand if it follows these principles.

One of the widely used patterns to keep the code cleaner is **Dependency Injection (DI)**. According to Dependency Injection, a software component should not create its dependencies; rather, the dependencies should be injected by an external source. The external source should know how objects are created, which reduces the chance of duplicating code for creating objects. The external source is called **Inversion of Control** container or, in short, the IoC container. Angular 1 uses DI quite extensively and this pattern is continued in Angular 2, in a more flexible way.

Dependency Injection in Angular 2

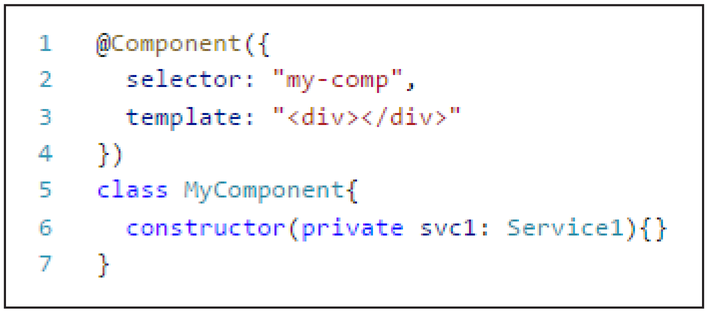
Angular's DI system is based on **injectors**. Angular creates an injector when the application is bootstrapped. All the application dependencies are registered in the injector, and the injector serves them when they are requested. The injector can be configured by registering providers at the module level or at a component level. For now, the discussion will be focused on a module and components will be discussed in a later section in this chapter.

A provider registered in the *NgModule* decorator is registered in the root injector. All of the built-in services of Angular are registered in the root injector as well. These dependencies are available across the application; they are not limited to the module that registers them.

Injecting Dependencies into Components

A registered dependency can be injected into a component's constructor. The TypeScript type of the injectable is used by Angular 2 for metadata information while injecting the dependency. Listing 26.1 shows how to inject a sample service in a component:

Listing 26.1: *DI with components*



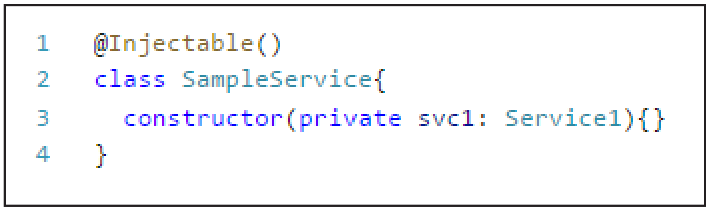
```
@Component({
  selector: "my-comp",
  template: "<div></div>"
})
class MyComponent{
  constructor(private svc1: Service1){}
}
```

In general, the dependencies injected in a component are marked as *private*. This is to keep them private to the component class and allow access to them within the component using the *this* keyword.

Injecting Dependencies into Services

Like components, Services in Angular 2 are TypeScript classes. A service with dependencies needs to be marked with the decorator *@Injectable*. The following snippet shows this:

Listing 26.2: *DI with Services*



```
@Injectable()
class SampleService{
  constructor(private svc1: Service1){}
}
```

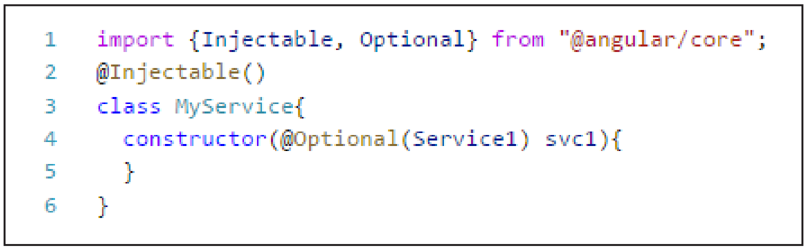
As in the case of components, the dependencies in the services are also marked as private to make them available inside the service class and to not expose them through the objects.

Though the decorator *Injectable* is necessary when the service has dependencies, best practice is to apply this decorator even when the service doesn't depend on any objects in order to make the service future-proof. The decorator also differentiates the classes used for model types and the services.

Optional Dependencies

If a component or a service depends on an object that comes from a third party library or an object that holds a value conditionally, it can be marked as an optional dependency. For example, you may define a service named *Logger* to print logs in the browser console while debugging the application, but this service is not a mandatory object for any of the code blocks in the application. They can be functional even when they don't log the messages. So, the service *Logger* can be marked as an optional dependency. The decorator @*Optional* makes a dependency optional. Before using an object obtained as an optional dependency, it is best to determine whether it holds a value. Listing 26.3 shows usage of an optional dependency:

Listing 26.3: *Optional dependency*

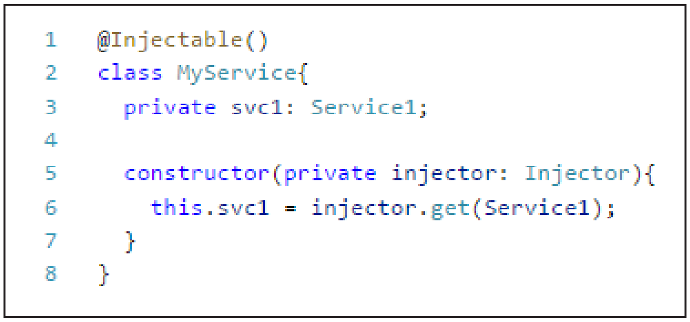


```
import {Injectable, Optional} from "@angular/core";
@Injectable()
class MyService{
  constructor(@Optional(Service1) svc1){
  }
}
```

Using Injector to get Dependencies

In rare cases, a service or a component may not need all of the dependencies up front and may load such dependencies on an as-needed basis. Such dependencies can be loaded dynamically using an *Injector*. *Injector* by itself is an injectable, and can be injected into a service or a component like any other service. Listing 26.4 gets a service using *Injector*:

Listing 26.4: *Using an Injector*



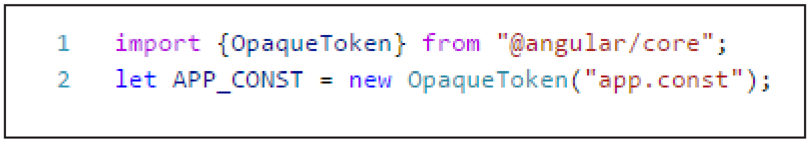
```
@Injectable()
class MyService{
  private svc1: Service1;

  constructor(private injector: Injector){
    this.svc1 = injector.get(Service1);
  }
}
```

Injecting non-services

Aside from services, an application would need several other types of dependencies. These include configuration objects, application constants, global objects, or any other types of objects. Angular 2 provides a way to inject these values using DI.

Registering the objects as dependencies involves a couple of steps. A dependency token must be created for the object, which can be done using the class *OpaqueToken* defined in the core package of Angular 2. The following snippet creates a token:

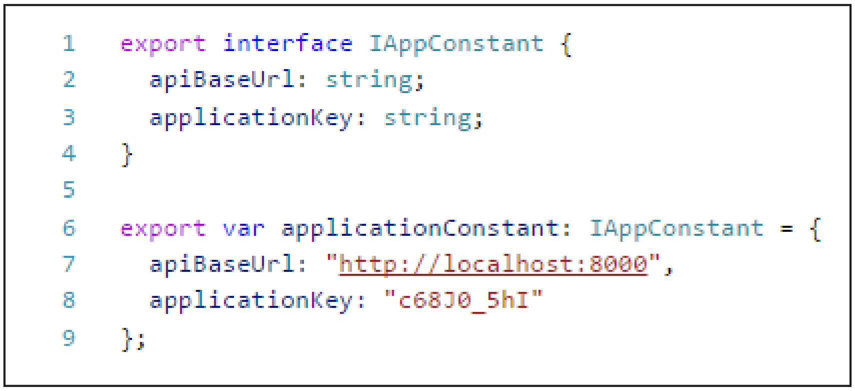
Listing 26.5: *Creating an OpaqueToken*



```
import {OpaqueToken} from "@angular/core";
let APP_CONST = new OpaqueToken("app.const");
```

This token can be used to provide any value; creating a type for the value is optional. Let's create an interface to represent the structure of the value and store a couple of values in the object.

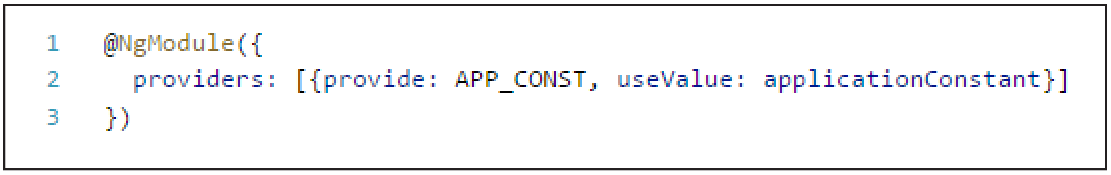
Listing 26.6: *Creating an object to be registered as a dependency*



```
export interface IAppConstant {
  apiBaseUrl: string;
  applicationKey: string;
}

export var applicationConstant: IAppConstant = {
  apiBaseUrl: "http://localhost:8000",
  applicationKey: "c68J0_5hI"
};
```

Now this object needs to be registered with the token created in Listing 26.5. This value needs to be registered with the *providers* option in the *NgModule* decorator. The following snippet shows the syntax for both of them:

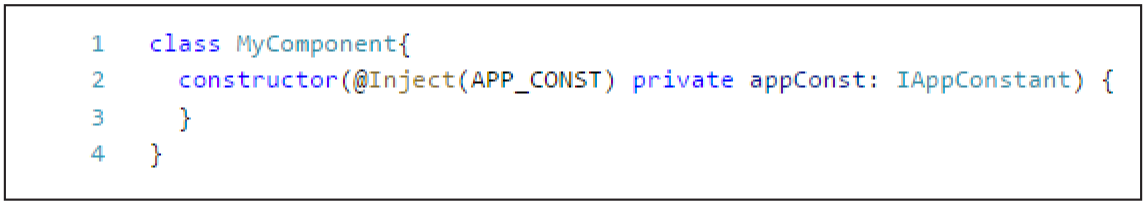
Listing 26.7: *Registering a token as a dependency*



```
@NgModule({
  providers: [{provide: APP_CONST, useValue: applicationConstant}]
})
```

To inject this value into any code block, the *Inject* decorator must be used and the token *APP_CONST* must be passed as an argument to the decorator. As the *Inject* decorator emits metadata for the member injected in, it is not needed to specify the data type of the object. It is still advised to assign the data type for type safety and tooling support while using the object. The following snippet injects *APP_CONST* into the *MyComponent* class:

Listing 26.7: *Injecting a token into a component*

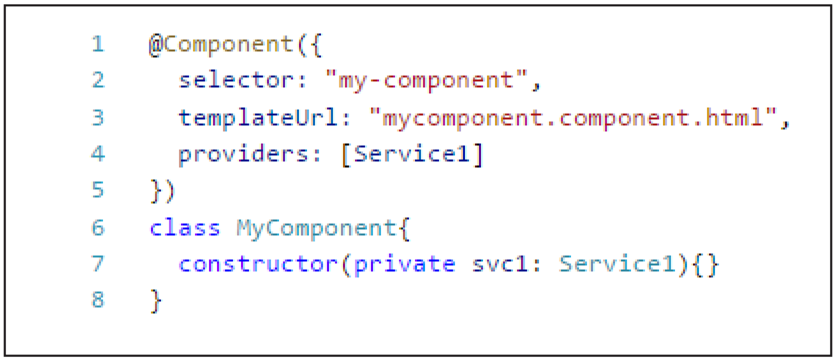


```
class MyComponent{
  constructor(@Inject(APP_CONST) private appConst: IAppConstant) {
  }
}
```

Injecting Dependencies into Components

Angular 2 supports injectors at the component level. Injectors can be configured at the component level using the *providers* option in the *Component* decorator. The injector (created at the level of a component) acts in the component and its children components. The following snippet shows how a dependency can be registered in a component:

Listing 26.8: *Registering a service in a component*



```
@Component({
  selector: "my-component",
  templateUrl: "mycomponent.component.html",
  providers: [Service1]
})
class MyComponent{
  constructor(private svc1: Service1){}
}
```

An instance of the service *Service1*, registered in listing 26.9, is made available to the component *MyComponent* and to all of its children components and directives. If a child component registers the same service again, it creates a new instance of the service. Additionally, since the component tree gets a new instance of the service, the root-level instance of the service won't be considered once the same service is registered at the component level.

The *providers* option in the component is similar to the one at the module level. It can be used to register the *OpaqueToken* discussed above as well.

Conclusion

Angular 2 is equped with a powerful Dependency Injection mechanism. This chapter explored different ways to inject services and other objects in Angular 2, which greatly aide in keeping the code base cleaner and making the code testable.

Chapter 27

Using Services

CHAPTER 25 DEMONSTRATED HOW Services can be used to provide some functionality to the components, and how Services can be injected into other services. Once a service is declared either in the module or a component and is used in other services and components that are in the same context, the service becomes *singleton*. In other words, the same object of the service will be shared across the application.

While this pattern works in several scenarios, sometimes the application may need different instances of the service, at different levels. For some services, every consuming component may need a new instance. The application may tend to take advantage of a class that provides a better implementation of the functionality than a service does.

This chapter explores different ways of using the services with examples. To follow along, you can start with the development environment created in chapter 5. To this, install the @angular/forms package using the following command:

```
npm install @angular/forms --save
```

This package has to be registered with SystemJS to be able to refer it in the application. Add the following entry to the *map* configuration in the file *systemjs.config.js:*

```
"@angular/forms": "node_modules/@angular/forms"
```

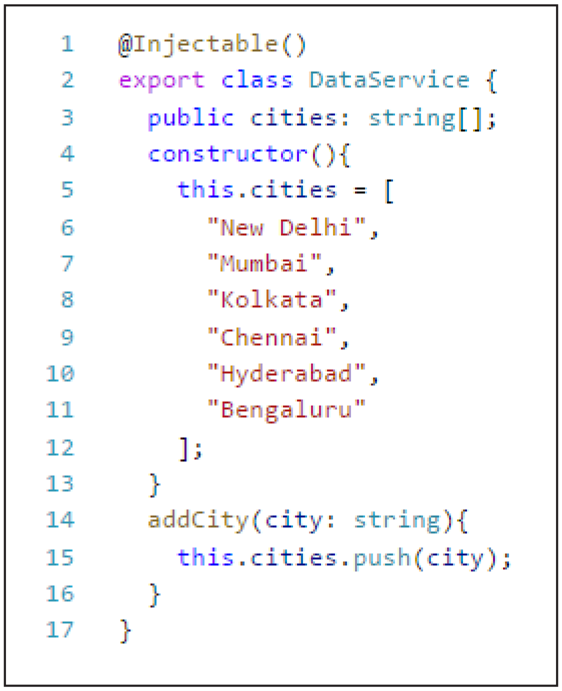
And the following entry to the *packages* section in the file *systemjs.config.js*:

```
"@angular/forms": { "main": "bundles/forms.umd.js", "defaultExtension": "js" }
```

Using Service in a sub-tree of Components

Let's create a service to hold a list of cities and use it in two components. Both of these components will have the same UI; the first one would be loading the second one in its own template. Listing 27.1 shows the definition of the service:

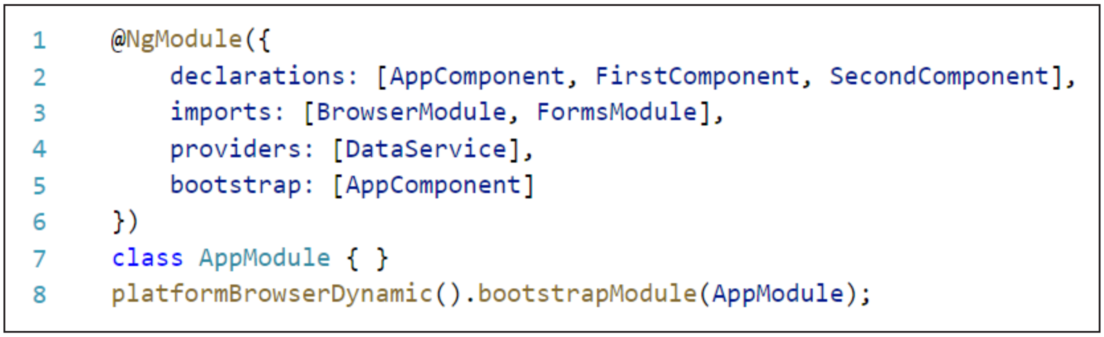
Listing 27.1: *Code of DataService*



```
@Injectable()
export class DataService {
  public cities: string[];
  constructor(){
    this.cities = [
      "New Delhi",
      "Mumbai",
      "Kolkata",
      "Chennai",
      "Hyderabad",
      "Bengaluru"
    ];
  }
  addCity(city: string){
    this.cities.push(city);
  }
}
```

The service must be declared in the *providers* option of the module. Listing 27.2 defines the module for the application and declares the components and services needed. The components will be defined in subsequent sections of the chapter.

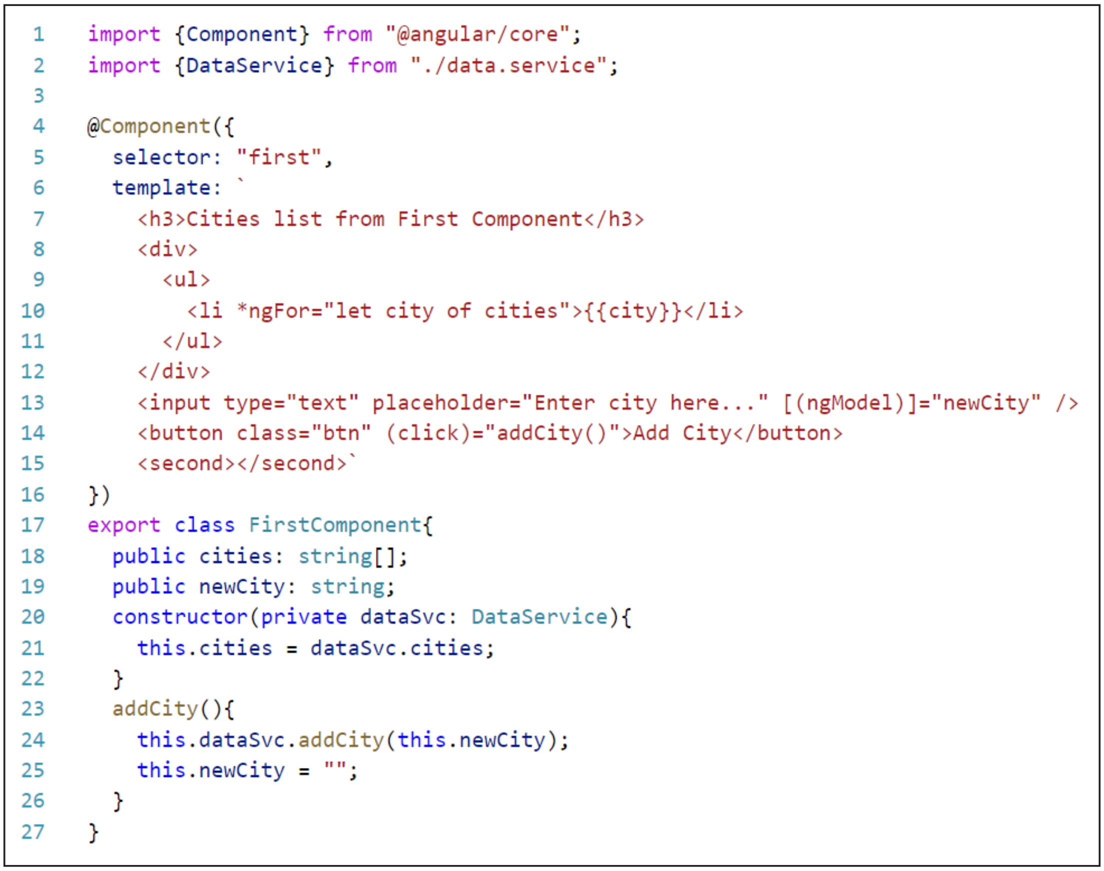
Listing 27.2: *Code of the module for the demo application*



```
@NgModule({
    declarations: [AppComponent, FirstComponent, SecondComponent],
    imports: [BrowserModule, FormsModule],
    providers: [DataService],
    bootstrap: [AppComponent]
})
class AppModule { }
platformBrowserDynamic().bootstrapModule(AppModule);
```

The components *FirstComponent* and *SecondComponent* used in the module will be added next. Listing 27.3 shows the definition of the *FirstComponent*. It uses the *SecondComponent* and the *DataService* defined in Listing 27.1:

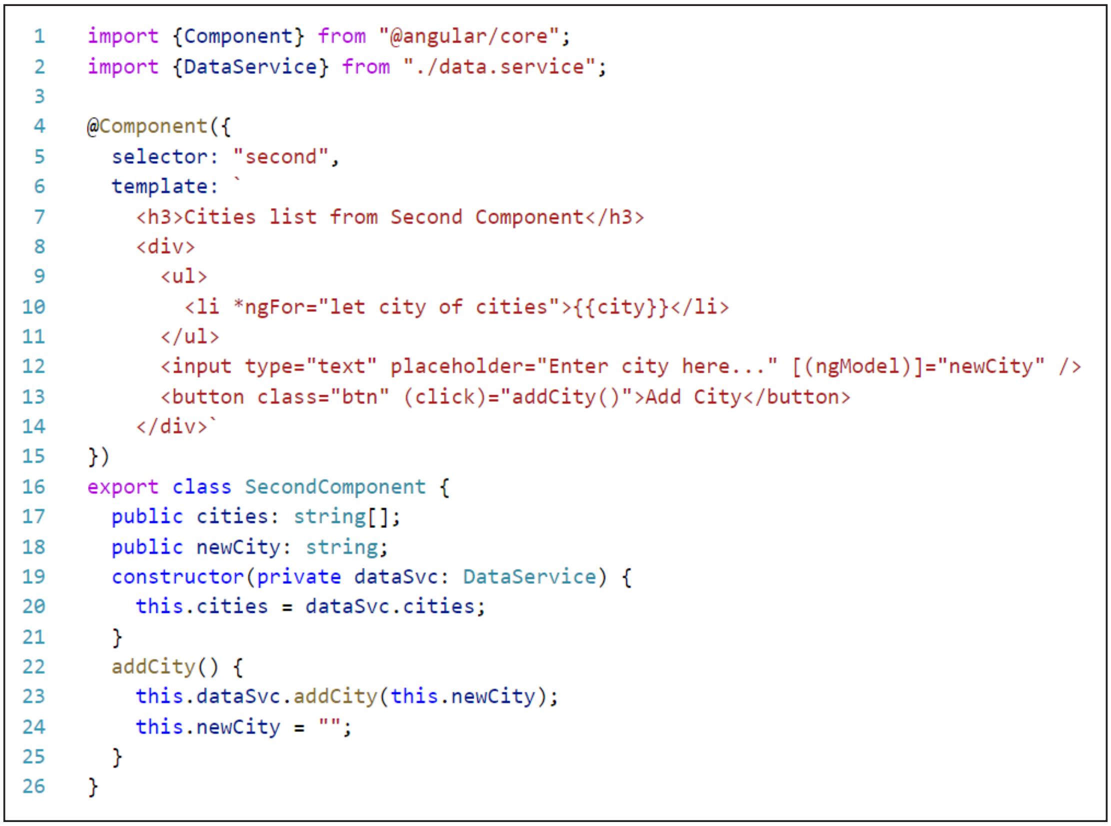
Listing 27.3: *Code of FirstComponent*



```
import {Component} from "@angular/core";
import {DataService} from "./data.service";

@Component({
  selector: "first",
  template: `
    <h3>Cities list from First Component</h3>
    <div>
      <ul>
        <li *ngFor="let city of cities">{{city}}</li>
      </ul>
    </div>
    <input type="text" placeholder="Enter city here..." [(ngModel)]="newCity" />
    <button class="btn" (click)="addCity()">Add City</button>
    <second></second>`
})
export class FirstComponent{
  public cities: string[];
  public newCity: string;
  constructor(private dataSvc: DataService){
    this.cities = dataSvc.cities;
  }
  addCity(){
    this.dataSvc.addCity(this.newCity);
    this.newCity = "";
  }
}
```

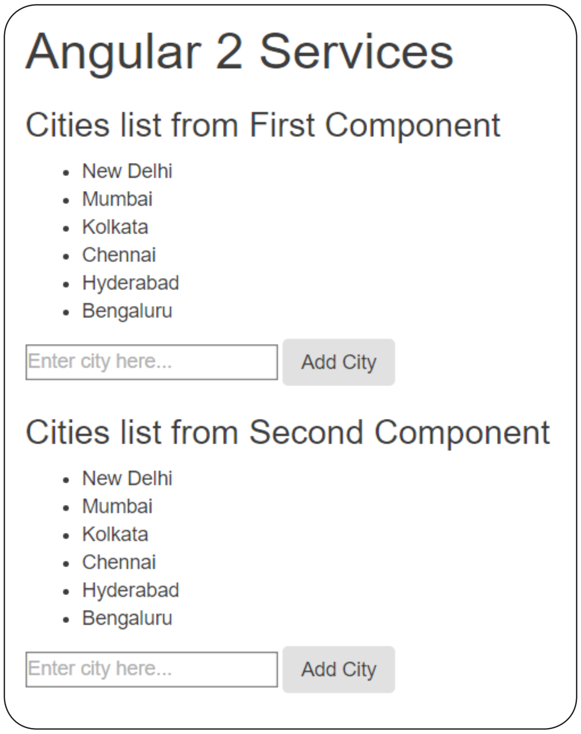
The component *SecondComponent* also has a similar definition, except it doesn't have the HTML tag of the *SecondComponent,* and it doesn't register the *DataService* in its *providers* array. The *SecondComponent* is shown below:

Listing 27.4: *Code of SecondComponent*



```
import {Component} from "@angular/core";
import {DataService} from "./data.service";

@Component({
  selector: "second",
  template: `
    <h3>Cities list from Second Component</h3>
    <div>
      <ul>
        <li *ngFor="let city of cities">{{city}}</li>
      </ul>
      <input type="text" placeholder="Enter city here..." [(ngModel)]="newCity" />
      <button class="btn" (click)="addCity()">Add City</button>
    </div>`
})
export class SecondComponent {
  public cities: string[];
  public newCity: string;
  constructor(private dataSvc: DataService) {
    this.cities = dataSvc.cities;
  }
  addCity() {
    this.dataSvc.addCity(this.newCity);
    this.newCity = "";
  }
}
```

The output of this code is shown in Figure 27.1:



Angular 2 Services

Cities list from First Component

- New Delhi
- Mumbai
- Kolkata
- Chennai
- Hyderabad
- Bengaluru

Cities list from Second Component

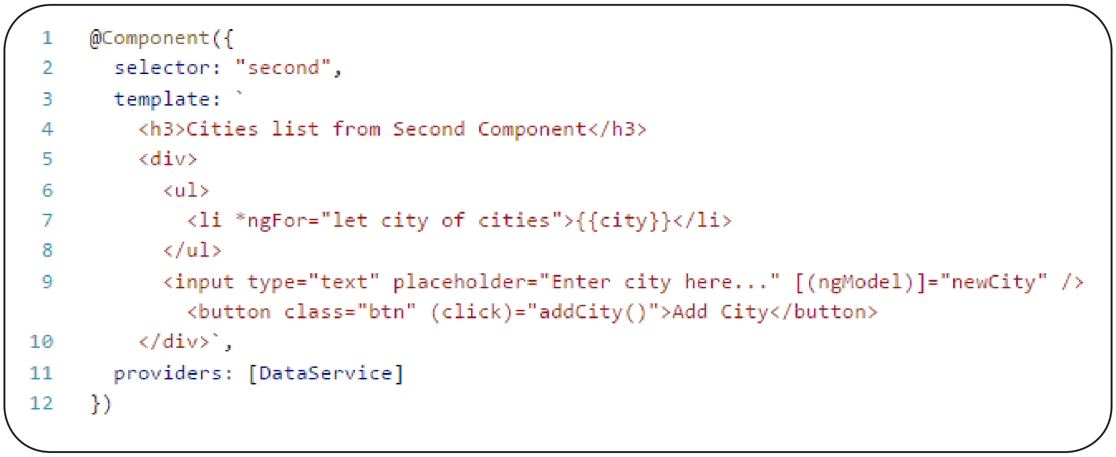
- New Delhi
- Mumbai
- Kolkata
- Chennai
- Hyderabad
- Bengaluru

Figure 27.1: *Two components using the same service*

If the *FirstComponent* adds a city, it immediately appears in the *SecondComponent* and vice-versa. This happens because the array *cities* is referred in both the components by reference, and the service is used as singleton.

Now make a small change to this by registering the service *DataService* again in the *SecondComponent*. Listing 27.5 shows the *Component* annotation of *SecondComponent* after making this change:

Listing 27.5: *Modified annotation of SecondComponent*



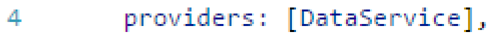
```
@Component({
  selector: "second",
  template: `
    <h3>Cities list from Second Component</h3>
    <div>
      <ul>
        <li *ngFor="let city of cities">{{city}}</li>
      </ul>
      <input type="text" placeholder="Enter city here..." [(ngModel)]="newCity" />
        <button class="btn" (click)="addCity()">Add City</button>
    </div>`,
  providers: [DataService]
})
```

After making this change, upon running the application, you will see that adding a city in *FirstComponent* does not add it to the *SecondComponent,* and vice-versa. If the service is registered again in a component, the framework will create a new instance for the context (a component or a sub-module) where it is registered for the second time.

Controlling the Way an Object is created for a Service

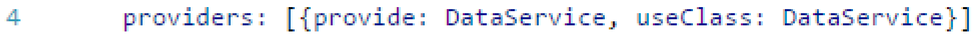
USING A DIFFERENT CLASS FOR SERVICE

Listing 27.2 used the service registered at the module level. The providers array was defined as:



```
providers: [DataService],
```

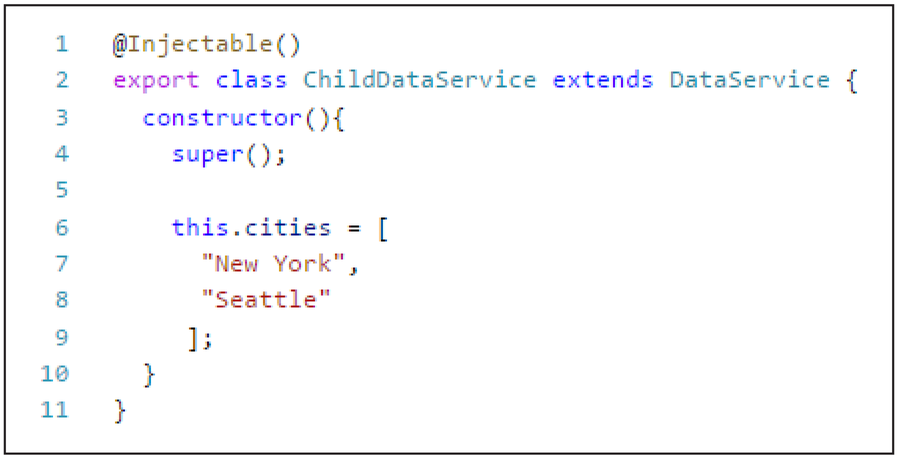
..which is the short cut of the following statement:



```
providers: [{provide: DataService, useClass: DataService}]
```

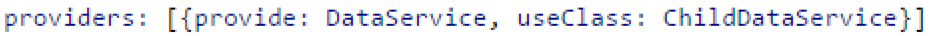
Here, the same class is used as both token and map. It is possible to use one class as the token, and another class inherited from the token class, as the service. Derive a class from the *DataService* class. Snippet 27.6 defines the derived class:

Listing 27.6: *Code of ChildDataService*



```
@Injectable()
export class ChildDataService extends DataService {
  constructor(){
    super();

    this.cities = [
      "New York",
      "Seattle"
    ];
  }
}
```

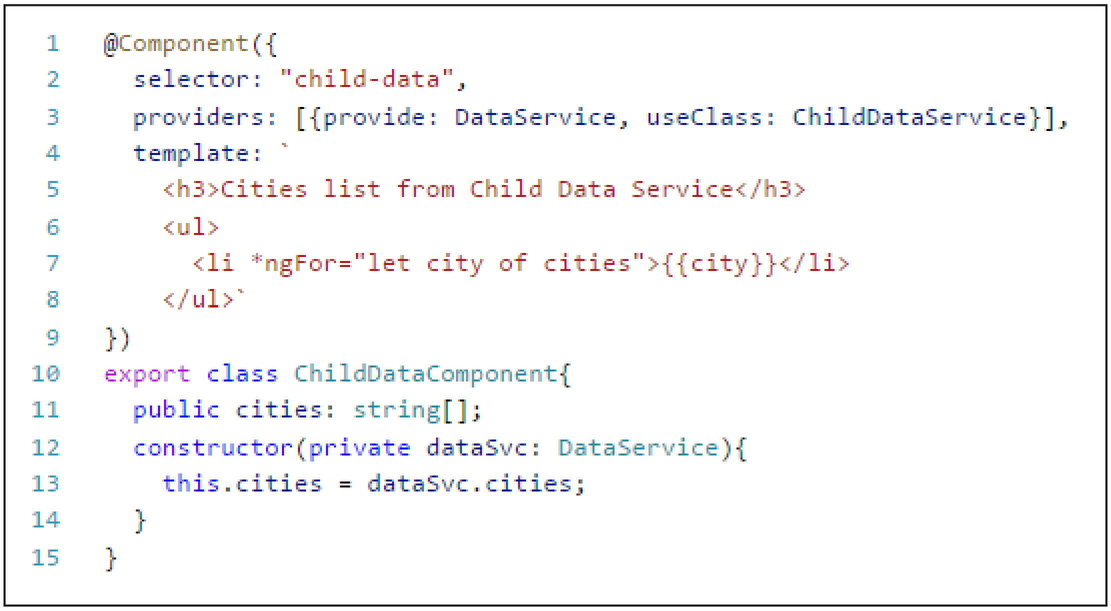
The child class overrides the values assigned in the array of the parent class with a new set of values. Now the class *ChildDataService* can be registered as the class for the token *DataService*. The following snippet shows how to do this:



```
providers: [{provide: DataService, useClass: ChildDataService}]
```

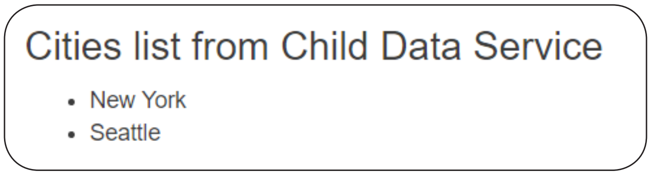
The following component shown in Listing 27.7 uses this way of service registration:

Listing 27.7: *Code of ChildDataComponent, it uses ChildDataService*



```
@Component({
  selector: "child-data",
  providers: [{provide: DataService, useClass: ChildDataService}],
  template: `
    <h3>Cities list from Child Data Service</h3>
    <ul>
      <li *ngFor="let city of cities">{{city}}</li>
    </ul>`
})
export class ChildDataComponent{
  public cities: string[];
  constructor(private dataSvc: DataService){
    this.cities = dataSvc.cities;
  }
}
```

When this component renders on the page, it displays the overridden data in the *ChildDataService*.



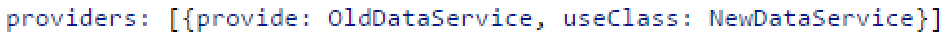
Cities list from Child Data Service

- New York
- Seattle

Figure 27.2: *Output of component using ChildDataService*

REPLACING AN OLDER SERVICE WITH A NEWER SERVICE

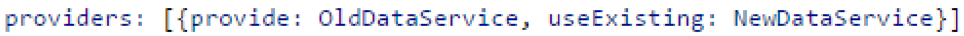
The mechanism used in listing 27.6 can be used to override an older service implementation with a new implementation. The application can override the old service with a new one using the providers option. For example, if the application used a service (*OldDataService),* but a new service (named *NewDataService), with the same structure and better access to the data, is available*. The following snippet shows the statement to override the service:



```
providers: [{provide: OldDataService, useClass: NewDataService}]
```

This is same as the syntax used earlier. The benefit of this approach is that the user need not go on searching for occurrences where the old service is used. The new service can be registered at an upper level so that it replaces the old service in the entire module, the component subtree, or the application.

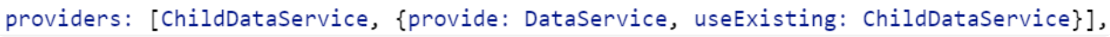
Of note: if the class *NewDataService* is already registered as a service, the application will have two different objects of the class: one from the original registration and the other from the *useClass.* This behavior can be changed by using the *useExisting* property in the provider object, in place of *useClass*. The following snippet does this:



```
providers: [{provide: OldDataService, useExisting: NewDataService}]
```

Before using *useExisting,* the target class has to be registered as a service. Otherwise it will fail saying "There are no providers for NewDataService".

To see this in action, you can modify the *providers* property in the listing 27.7 to use the *useExisting* option to register the *ChildDataService* as shown below:



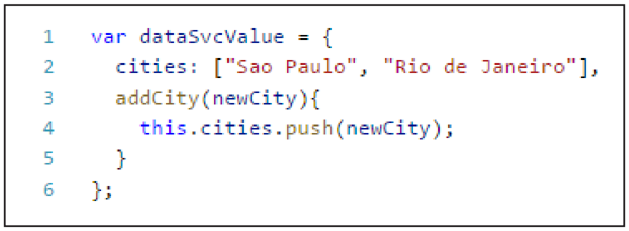
```
providers: [ChildDataService, {provide: DataService, useExisting: ChildDataService}],
```

Otherwise, you can take it as an assignment to write a new class and see it in action using the *useClass* and *useExisting* options.

PROVIDING VALUES FOR SERVICES

Sometimes, the application may try to use a fixed value for a service, rather than asking Angular to create an object using the class. In such a case, the service can be registered using the *useValue* property of the provider registration, and assign this object. Listing 27.8 shows an example of providing a value for the *DataService*:

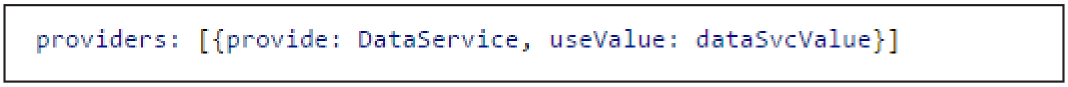
Listing 27.8: *A sample value for DataService*



```
var dataSvcValue = {
  cities: ["Sao Paulo", "Rio de Janeiro"],
  addCity(newCity){
    this.cities.push(newCity);
  }
};
```

This can be registered in the component or module as shown in listing 27.9.

Listing 27.9: *providers using useValue*

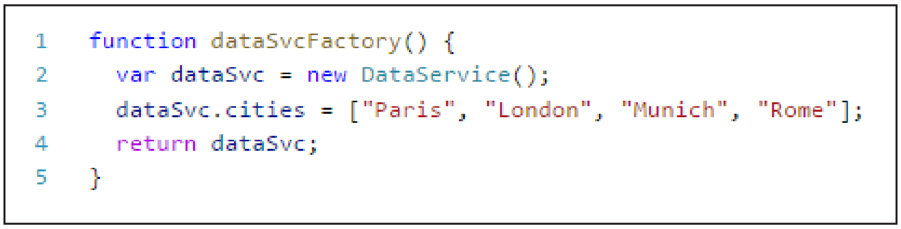


```
providers: [{provide: DataService, useValue: dataSvcValue}]
```

Using Factory Functions for Services

Factory pattern is commonly used to abstract the logic of creating an object depending upon certain values in the application, and the environment. Angular 2 supports this pattern through the *useFactory* property in the *providers* option. A function creating an object of the class can be registered as the factory function for the service. Listing 27.10 creates a simple factory function for *DataService,* like any other class or object, the factory function can be created anywhere and imported into the file creating module:

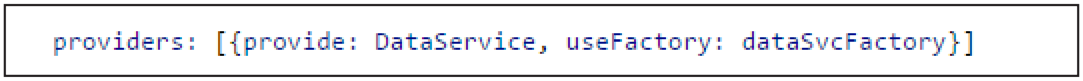
Listing 27.10: *A factory function returning an object of DataService*



```
function dataSvcFactory() {
  var dataSvc = new DataService();
  dataSvc.cities = ["Paris", "London", "Munich", "Rome"];
  return dataSvc;
}
```

Listing 27.11 specifies this method in the providers option.

Listing 27.11: *providers using useFactory*



```
providers: [{provide: DataService, useFactory: dataSvcFactory}]
```

Conclusion

This chapter demonstrated how Services can be registered in different ways in Angular 2 and how their objects can be controlled using different options available in the provider registration. These features make the DI system in Angular extremely powerful and flexible. Some of these features help in mocking services in unit tests as well. We will cover these features in the chapters on unit testing.

Chapter 28

Observables and Reactive Programming

AN APPLICATION CAN RECEIVE information from a number of different sources. The source could be an array, a backend API, a real-time data stream, or an event from a UI element. The responses received from the sources are sometimes predictable, and sometimes dynamic. Some of them work synchronously, while others work asynchronously. Regardless, the application receives some data from these sources and uses it to provide functionality.

There are different approaches to handle each of these cases. Ideally, all of them can be handled in the same way. Reactive programming provides us with a way to do that.

Reactive programming is programming with event streams. Any source of information can be viewed as an event stream. For example, consider iterating through an array. It can be viewed as an array emitting the values one by one, and the consumer of the array listening to it as long as the array has data. The consumer can receive this data and use it in the way needed.

The event stream in reactive programming consists of a series of events from a single source. Each of these events may either pass or fail. The stream doesn't prevent the next event from executing when an event fails. At the end of the stream, it sends a signal indicating that the stream has completed execution.

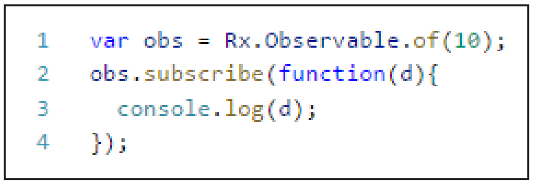
RxJS is the JavaScript library that provides reactiveness in JavaScript. Using RxJS, any event can be converted to an event stream and have a single API to handle any type of event. More information on the paradigm of reactive programming and the RxJS library can be found on the *official site of RxJS*.

Observables in RxJS

RxJS defines several objects to work with event streams. Any event stream in RxJS is viewed as Observable, and it notifies an Observer object whenever an event occurs. If you are familiar with promises, you can imagine Observable as an extension to promise. A promise gets the result once, whereas an Observable can get the result from its source multiple times. This section will explore a few ways of creating observables and using them.

Let's write the simplest possible observable to return a single value when it is asked to send values. Listing 28.1 shows an example:

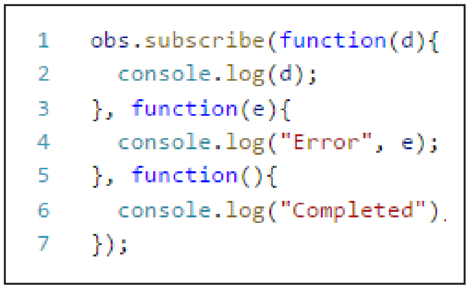
Listing 28.1: *A simple observable*



```
var obs = Rx.Observable.of(10);
obs.subscribe(function(d){
  console.log(d);
});
```

Here, the numeric value 10 is converted into an observable using the *of* operator of the *Observable* API. This value is returned to the *success* callback of the *subscribe* method. The *subscribe* method accepts two more callbacks: one for error, and the other to indicate that the sequence is complete. Let's add two more callbacks to the subscribe method:

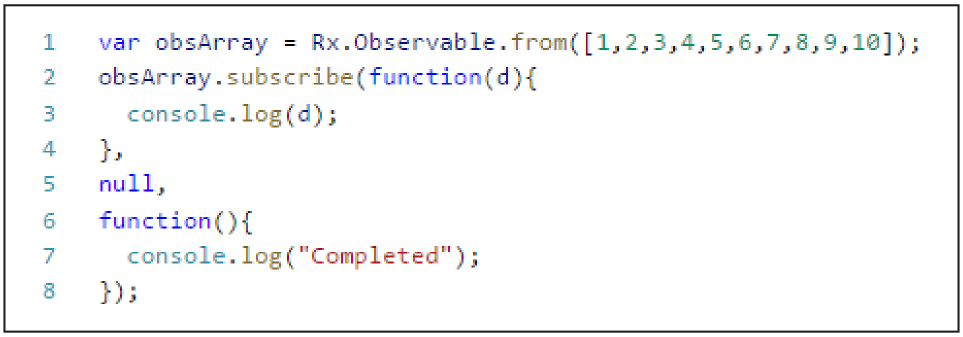
Listing 28.2: *Observable with callbacks to success, error and completed*



```
obs.subscribe(function(d){
  console.log(d);
}, function(e){
  console.log("Error", e);
}, function(){
  console.log("Completed")
});
```

Upon running this snippet, the value 10 and completed message "Completed" are displayed. Though it works, the observable with a single value is not a true demonstration of what it can offer. The next example shows an observable derived from an array. The *from* operator of observable helps convert an array into an observable. Listing 28.3 demonstrates an example:

Listing 28.3: *Observable from array*

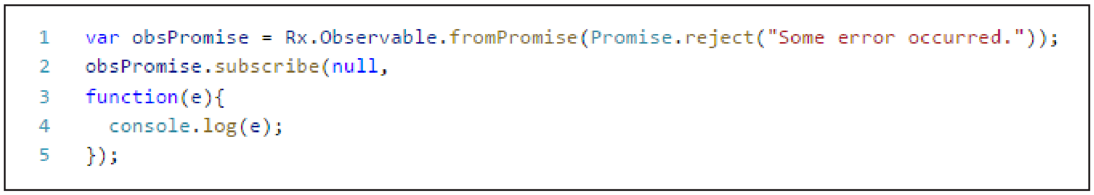


```
var obsArray = Rx.Observable.from([1,2,3,4,5,6,7,8,9,10]);
obsArray.subscribe(function(d){
  console.log(d);
},
null,
function(){
  console.log("Completed");
});
```

The *subscribe* method would be called for every item in the array. We are deliberately passing *null* for the failure callback, as the observable will not fail in this case. Once all elements in the array are processed, it calls the completed callback.

Promises can be converted into observables, too. Listing 28.4 converts an ES6 promise into an observable:

Listing 28.4: *Observable from promise*



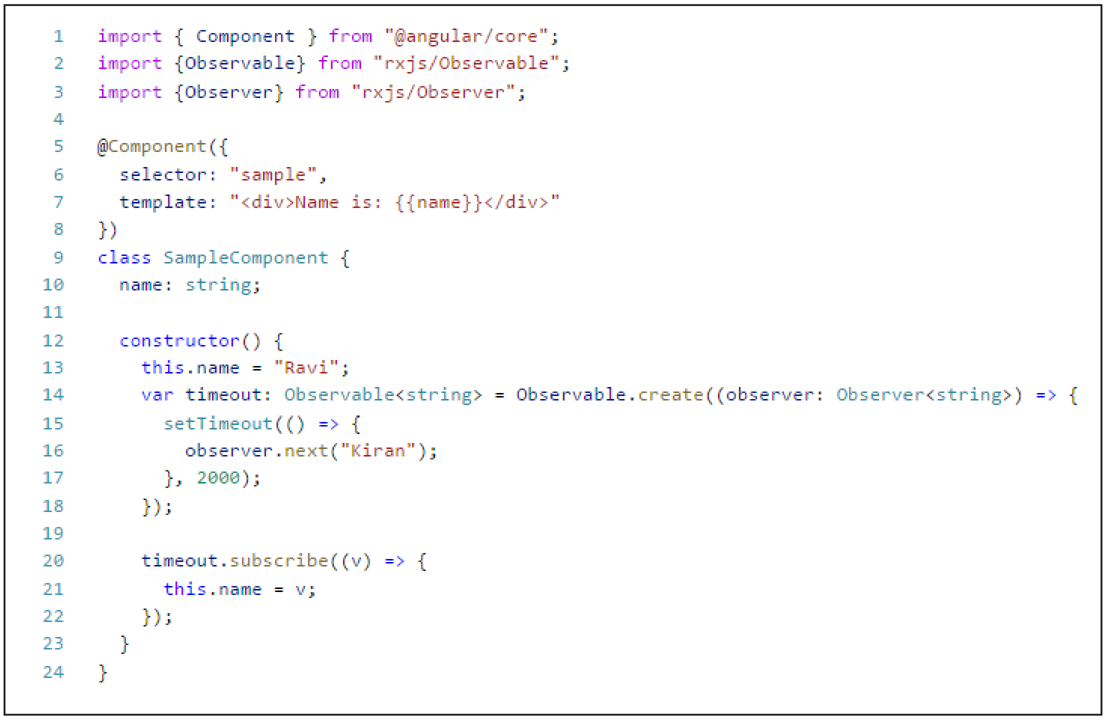
```
var obsPromise = Rx.Observable.fromPromise(Promise.reject("Some error occurred."));
obsPromise.subscribe(null,
function(e){
  console.log(e);
});
```

Listing 28.4 has a static promise which fails with a result immediately. So the error callback of the *subscribe* method would be called. This is the reason the success callback is not implemented in the snippet.

Using Observables in Angular 2

Observables can be used seamlessly in Angular 2 applications. The framework uses observables quite extensively to work with HTTP APIs (will be discussed in Chapter 29). To see how observables can be used in Angular 2, let's wrap *setTimeout* around an observable and use it in a component. Listing 28.5 shows code of the component:

Listing 28.5: *A sample component using observable*



```
import { Component } from "@angular/core";
import {Observable} from "rxjs/Observable";
import {Observer} from "rxjs/Observer";

@Component({
  selector: "sample",
  template: "<div>Name is: {{name}}</div>"
})
class SampleComponent {
  name: string;

  constructor() {
    this.name = "Ravi";
    var timeout: Observable<string> = Observable.create((observer: Observer<string>) => {
      setTimeout(() => {
        observer.next("Kiran");
      }, 2000);
    });

    timeout.subscribe((v) => {
      this.name = v;
    });
  }
}
```

The above snippet does the following:

- Creates an observable over *setTimeout* using the *Observable.create* method
- The *setTiemout* emits a value after 2 seconds using the method *observer.next*
- The observable object *timeout* gets a notification when the value is emitted and it sets the new value to the property *name*

The value displayed in the data binding expression changes as soon as the value is assigned in the subscribe callback.

Creating a chat application using Angular 2 and RxJS

Following these basic examples of RxJS, let's move on to build an application to understand it better. This section will create a simple chat application to use some of the features discussed, and to introduce you to some additional APIs.

Understanding the Application Design

The chat application will be about the interaction of a user with a bot. The bot will have a fixed set of messages, and will respond with a random message from this list to every message it receives from the user. The application will have the following pieces:

- A component named *AppComponent*, to receive messages from the user and to display the list of messages
- A service named *ChatBotService*, to play the role of a bot
- Another service named *ChattingService*, to make the communication happen

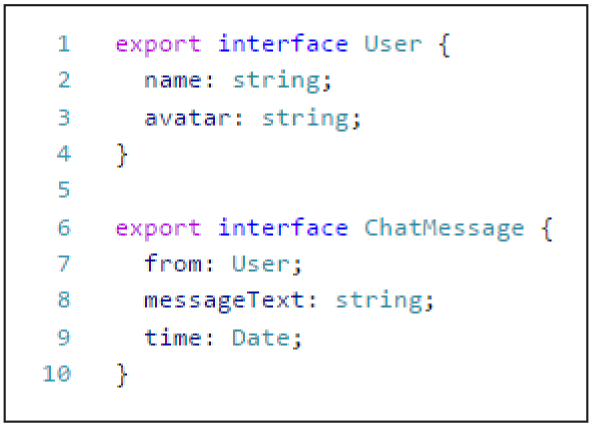
The services will make use of observables for communication. Here is how the communication will happen between the services and the component:

- When the *AppComponent* receives a message from a user, it sends it to the *ChattingService* using a method
- The *ChattingService* sends this message to the *ChatBotService* using a method
- The *ChatBotService* picks a random message from the list and emits it as next value in the observable
- The *ChattingService* subscribes to the observable provided by the *ChatBotService* and it emits the message its own observable
- The *AppComponent* subscribes to the observable of the *ChattingService* and displays the message on the view as soon as it is received

Setting up Model and Data

Now that it is clear how the application is divided and the way it works, it is time to implement it. The basic requirements for the application are two interfaces, defining types of user, and chat message objects. The *User* interface will have two properties: name and avatar. The *ChatMessage* interface will have three properties to hold the values of sending user, message tex,t and time of the message. Add a new file named *chat.model.ts* and add the following interfaces to it:

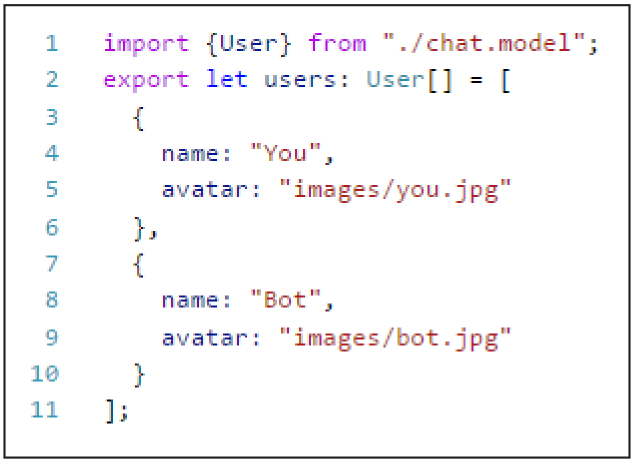
Listing 28.6: *Interfaces for user and chat message*



```
export interface User {
  name: string;
  avatar: string;
}

export interface ChatMessage {
  from: User;
  messageText: string;
  time: Date;
}
```

The sample chat application will have just two users. Details of these users is stored in an array. Listing 28.7 shows a snippet of the array:

Listing 28.7: *Static array of chat users*



```
import {User} from "./chat.model";
export let users: User[] = [
  {
    name: "You",
    avatar: "images/you.jpg"
  },
  {
    name: "Bot",
    avatar: "images/bot.jpg"
  }
];
```

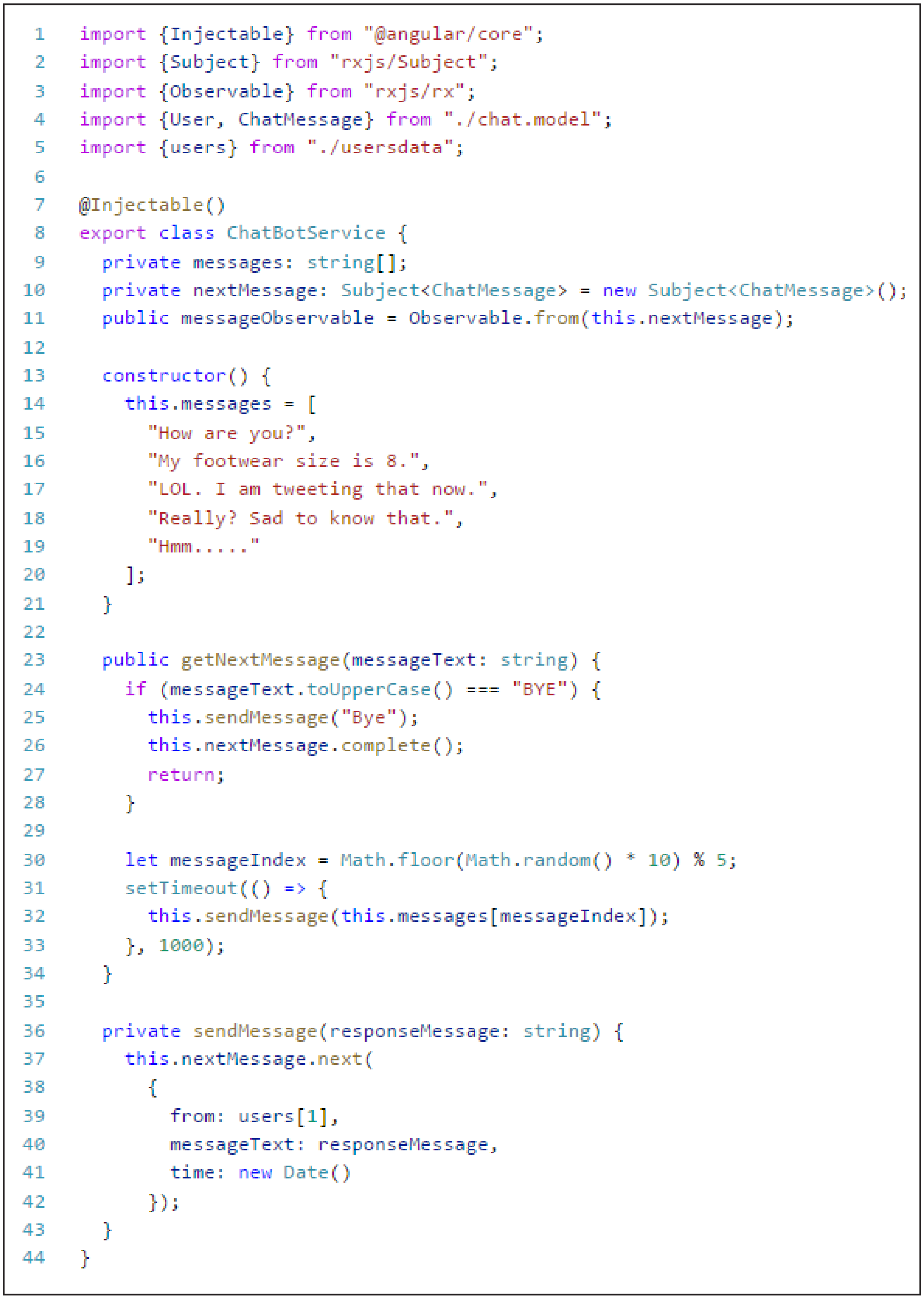
The images for these users are chosen from the list of authorized photos available in *UI Faces*, but may be replaced them with your choice of images.

Creating the *ChatBot* Service

As mentioned previously, the bot service includes a set of messages and will receive a message and subsequently respond (with a randomly chosen message).

The *Subject* class of RxJS provides a pub/sub API to send and receive messages. The sample will use this class to communicate between different pieces. The *Subject* object will be wrapped around an *Observable* object. The *Observable* object will be made a public property in the class so that a consumer can subscribe to it using the object. Listing 28.8 shows the code of this service:

Listing 28.8: *Code of ChatBotService*



```
import {Injectable} from "@angular/core";
import {Subject} from "rxjs/Subject";
import {Observable} from "rxjs/rx";
import {User, ChatMessage} from "./chat.model";
import {users} from "./usersdata";

@Injectable()
export class ChatBotService {
  private messages: string[];
  private nextMessage: Subject<ChatMessage> = new Subject<ChatMessage>();
  public messageObservable = Observable.from(this.nextMessage);

  constructor() {
    this.messages = [
      "How are you?",
      "My footwear size is 8.",
      "LOL. I am tweeting that now.",
      "Really? Sad to know that.",
      "Hmm....."
    ];
  }

  public getNextMessage(messageText: string) {
    if (messageText.toUpperCase() === "BYE") {
      this.sendMessage("Bye");
      this.nextMessage.complete();
      return;
    }

    let messageIndex = Math.floor(Math.random() * 10) % 5;
    setTimeout(() => {
      this.sendMessage(this.messages[messageIndex]);
    }, 1000);
  }

  private sendMessage(responseMessage: string) {
    this.nextMessage.next(
      {
        from: users[1],
        messageText: responseMessage,
        time: new Date()
      });
  }
}
```

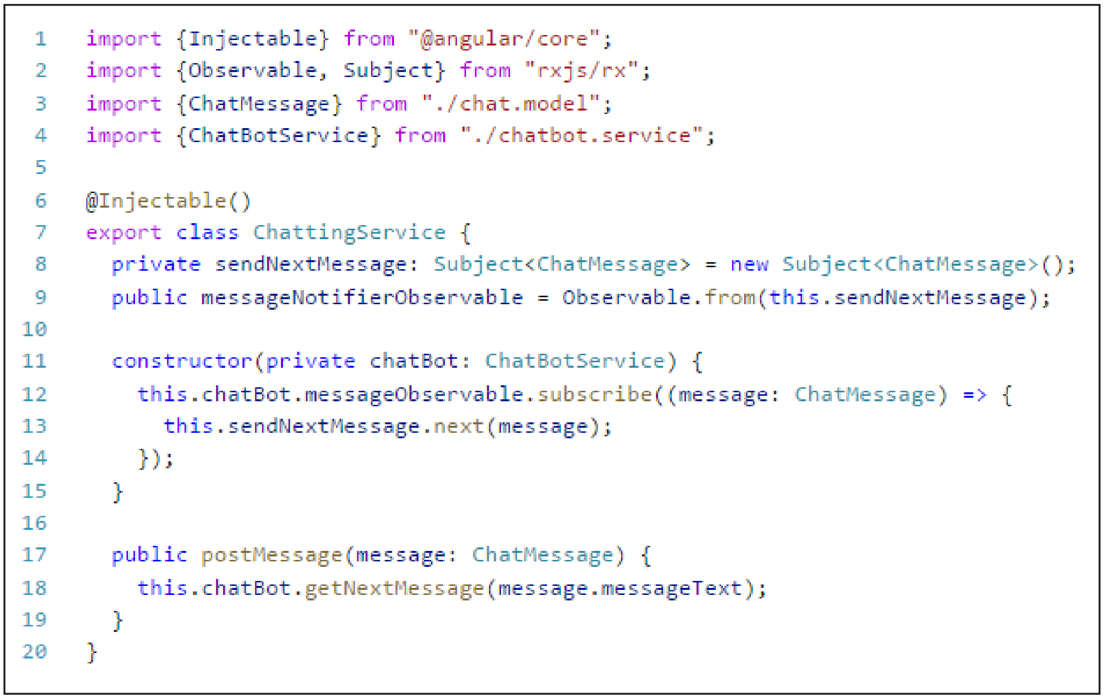
The above service exports the following members:

- **getNextMessage:** This method receives a message and passes the message, in response to the subject *nextMessage,* after a delay of 1 second. If the message received is "bye", then it responds with a "Bye" and completes the subject by calling the *complete* method. The subject sends a completed flag to its subscribers, and doesn't send any messages after this point.
- **messageObservable:** An observable object wraps the *nextMessage* subject. The consumers of the bot must subscribe to the *messageObservable* object to get the response messages.

Creating the *ChattingService*

This service plays the role of a mediator. It abstracts the bot and the component won't know anything beyond this service. The following snippet shows the code of this service:

Listing 28.9: *Code of ChattingService*



```
import {Injectable} from "@angular/core";
import {Observable, Subject} from "rxjs/rx";
import {ChatMessage} from "./chat.model";
import {ChatBotService} from "./chatbot.service";

@Injectable()
export class ChattingService {
  private sendNextMessage: Subject<ChatMessage> = new Subject<ChatMessage>();
  public messageNotifierObservable = Observable.from(this.sendNextMessage);

  constructor(private chatBot: ChatBotService) {
    this.chatBot.messageObservable.subscribe((message: ChatMessage) => {
      this.sendNextMessage.next(message);
    });
  }

  public postMessage(message: ChatMessage) {
    this.chatBot.getNextMessage(message.messageText);
  }
}
```

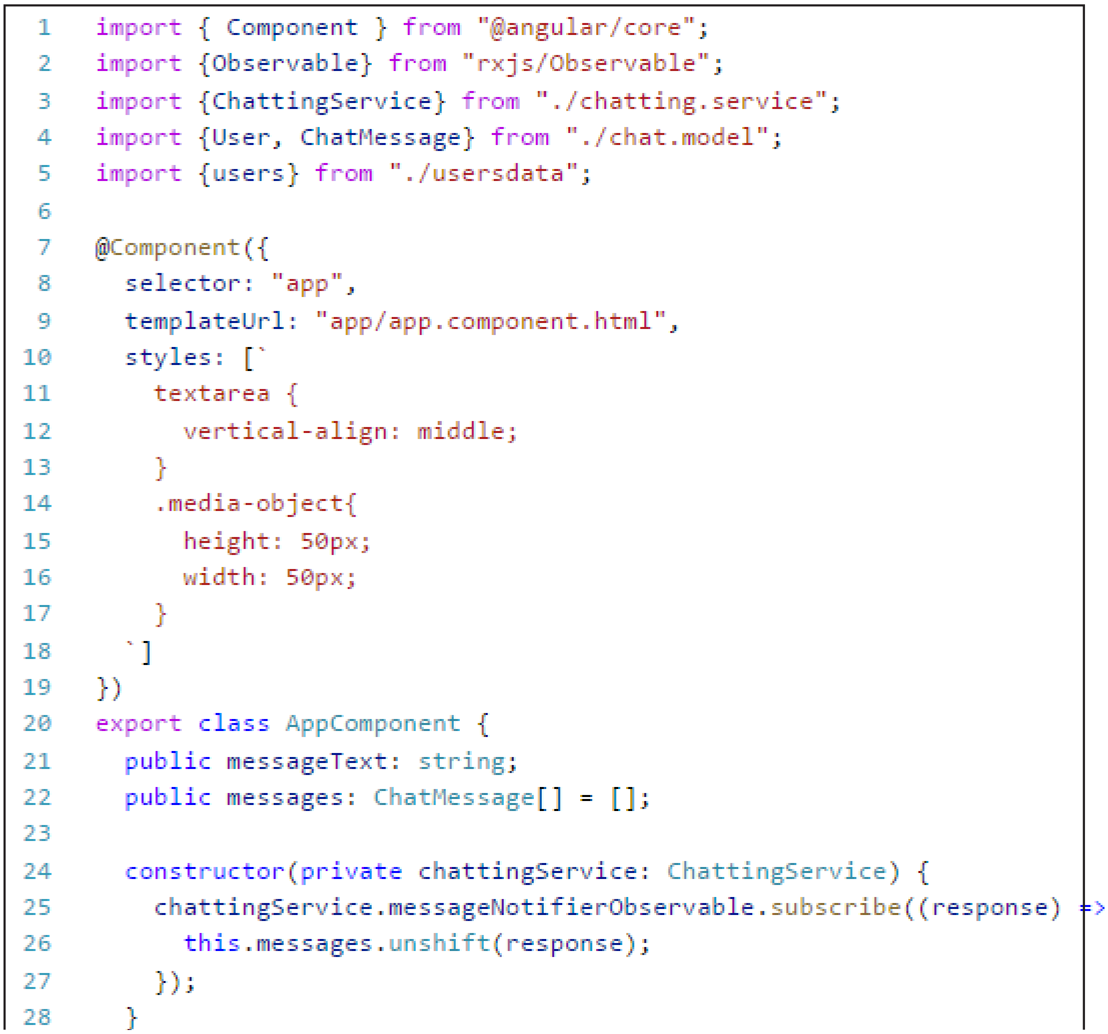
This service has the following public members:

- *messageNotifierObservable*: A wrapper around the subject *sendNextMessage*. Constructor of the service subscribes to the *messageObservable* of the bot and passes the received message to the subject *sendNextMessage*. The component must subscribe to this observable to get the subsequent messages.
- *postMessage*: This method receives a message from the component and passes it to the bot.

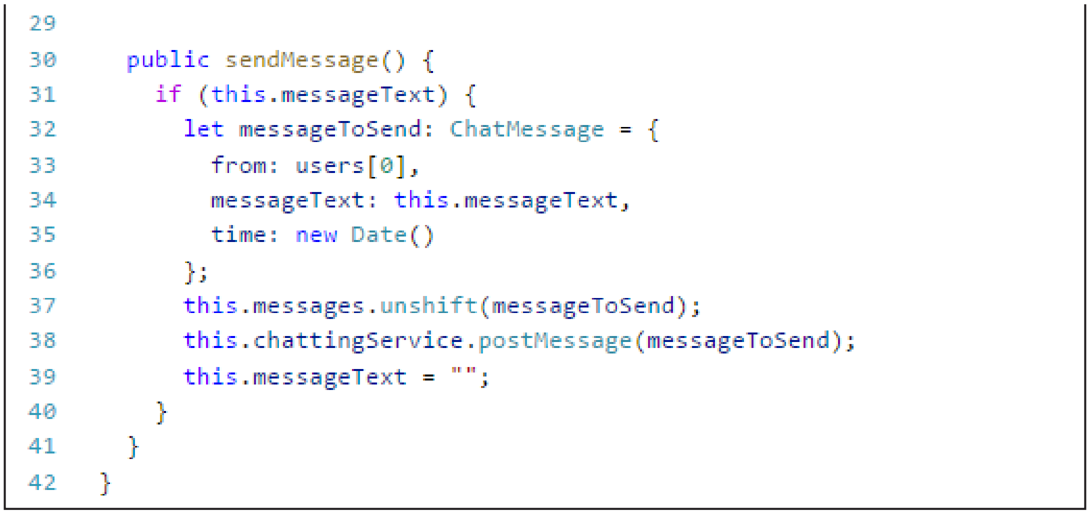
Creating the *AppComponent*

The last piece of our application is the component which will allow the user will interact with the bot. This is the only component in the application, so the application will use it for the bootstrap process. Listing 28.10 shows code of the *AppComponent*.

Listing 28.10: *Code of AppComponent*



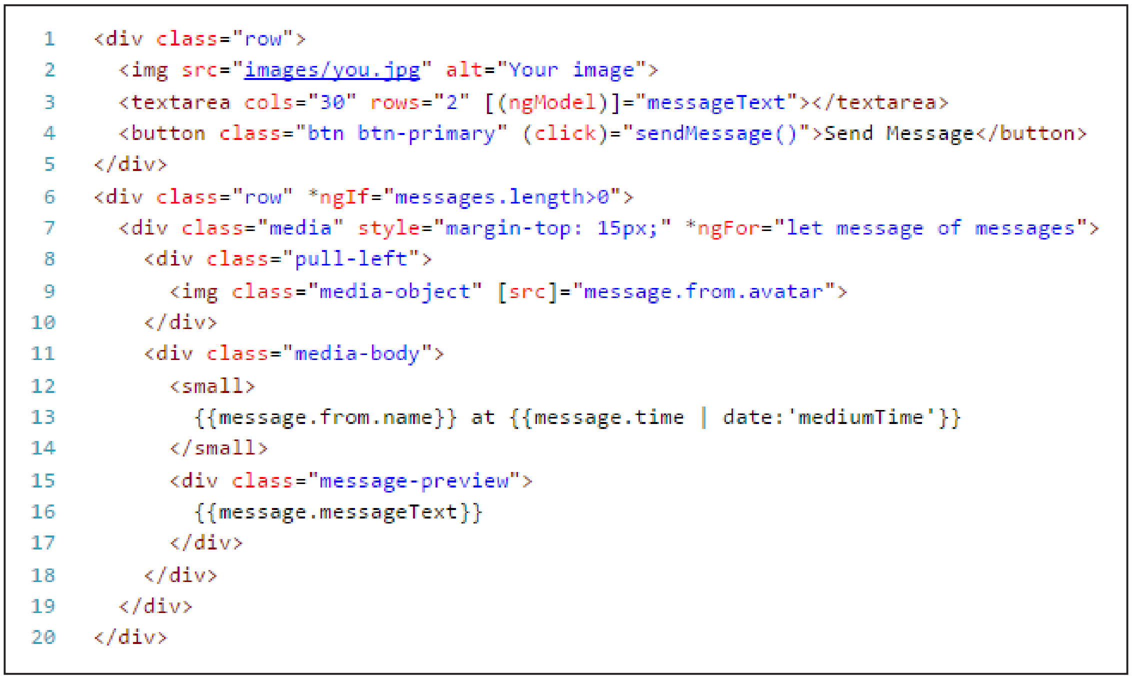
```
import { Component } from "@angular/core";
import {Observable} from "rxjs/Observable";
import {ChattingService} from "./chatting.service";
import {User, ChatMessage} from "./chat.model";
import {users} from "./usersdata";

@Component({
  selector: "app",
  templateUrl: "app/app.component.html",
  styles: [`
    textarea {
      vertical-align: middle;
    }
    .media-object{
      height: 50px;
      width: 50px;
    }
  `]
})
export class AppComponent {
  public messageText: string;
  public messages: ChatMessage[] = [];

  constructor(private chattingService: ChattingService) {
    chattingService.messageNotifierObservable.subscribe((response) =>
      this.messages.unshift(response);
    });
  }
```



```

  public sendMessage() {
    if (this.messageText) {
      let messageToSend: ChatMessage = {
        from: users[0],
        messageText: this.messageText,
        time: new Date()
      };
      this.messages.unshift(messageToSend);
      this.chattingService.postMessage(messageToSend);
      this.messageText = "";
    }
  }
}
```

Listing 28.11 shows the HTML template of the component:

Listing 28.11: *Template of AppComponent*



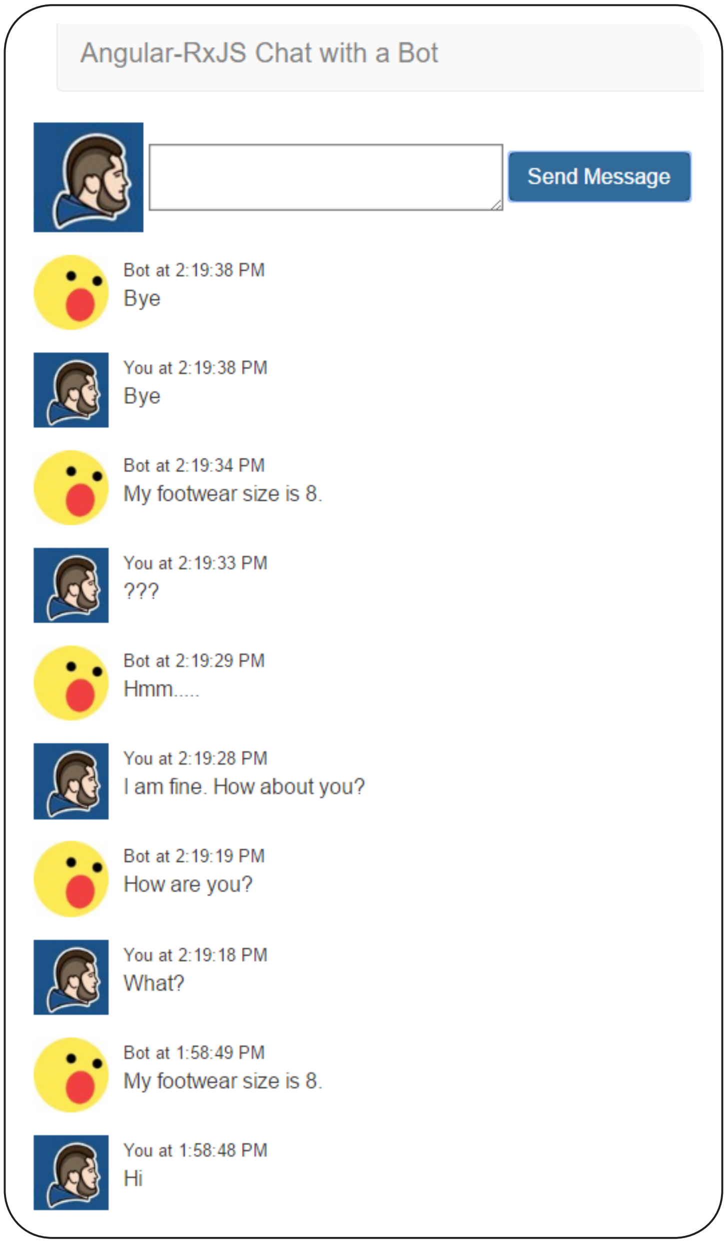
```
<div class="row">
  <img src="images/you.jpg" alt="Your image">
  <textarea cols="30" rows="2" [(ngModel)]="messageText"></textarea>
  <button class="btn btn-primary" (click)="sendMessage()">Send Message</button>
</div>
<div class="row" *ngIf="messages.length>0">
  <div class="media" style="margin-top: 15px;" *ngFor="let message of messages">
    <div class="pull-left">
      <img class="media-object" [src]="message.from.avatar">
    </div>
    <div class="media-body">
      <small>
        {{message.from.name}} at {{message.time | date:'mediumTime'}}
      </small>
      <div class="message-preview">
        {{message.messageText}}
      </div>
    </div>
  </div>
</div>
```

The component performs the following tasks:

- Accepts a message from the user in a textarea and sends it to the *ChattingService*
- Subscribes to the *messageNotifierObservable* of the *ChattingService* to get the response message

- Adds the messages to the array *messages* and displays them on the UI

Figure 28.1 shows a conversation with the bot:



Bot at 2:19:38 PM
Bye

You at 2:19:38 PM
Bye

Bot at 2:19:34 PM
My footwear size is 8.

You at 2:19:33 PM
???

Bot at 2:19:29 PM
Hmm.....

You at 2:19:28 PM
I am fine. How about you?

Bot at 2:19:19 PM
How are you?

You at 2:19:18 PM
What?

Bot at 1:58:49 PM
My footwear size is 8.

You at 1:58:48 PM
Hi

Figure 28.1: *An instance of the messages in a chat bot application*

Conclusion

RxJS is a rich library that provides a unified way to handle any type of data stream. Angular 2 uses it quite extensively and encourages the use of observables. This chapter demonstrated a simple chat application to show how seamlessly RxJS can be used with Angular 2. This duo can be used to build some powerful applications.

Chapter 29 Send and Receive Data with HTTP

ALL CLIENT APPLICATIONS HAVE a primary need to interact with a back-end data service. The application would perform its operations based on the data, and operations provided by the back-end service. In JavaScript applications, such services are invoked using AJAX and the format of the result is obtained in JSON format. Most of the modern browsers support the HTTP-based APIs XMLHttpRequest and JSONP to communicate with the services hosted on the same domain, or in a different domain. The applications, in general, use the wrappers around these APIs to avoid dealing with the lower level configurations and rather focus on the business logic.

Angular 2 provides wrappers around these APIs, which produces observables in response. The applications can subscribe to the observables and consume the result when it is available.

This chapter will discuss how to use the APIs of Angular 2 to perform CRUD operations over both XHR and JSONP.

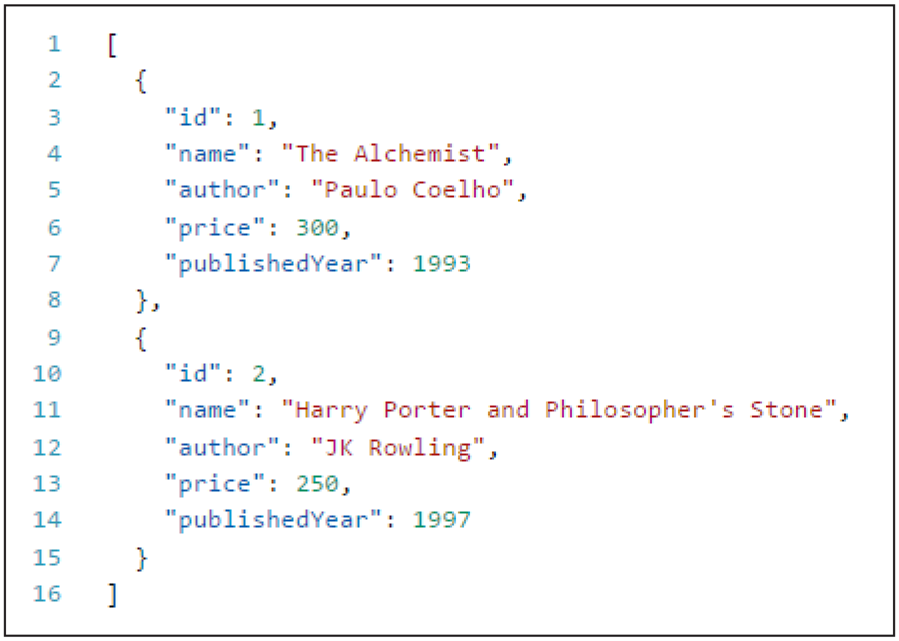
CRUD Operations Using HTTP API

Using a set of services available in Angular 2 and CRUD operations, this section will demonstrate how to fetch all books, add a new book, modify an existing book, and delete a book. The REST APIs are built using the Koa framework of Node.js. A static list of books is available in a JSON file. The sample application will read this data and perform rest of the operations in the memory. So, once the Node.js server is restarted, the changes made will be lost and the data on the screen will be same as the data in the file.

Building the service

First, let's build the service. As mentioned earlier, the application has a JSON file containing a list of books. The name of the file is **books.json.** Listing 29.1 shows two books from this file:

Listing 29.1: *An excerpt from books.json*



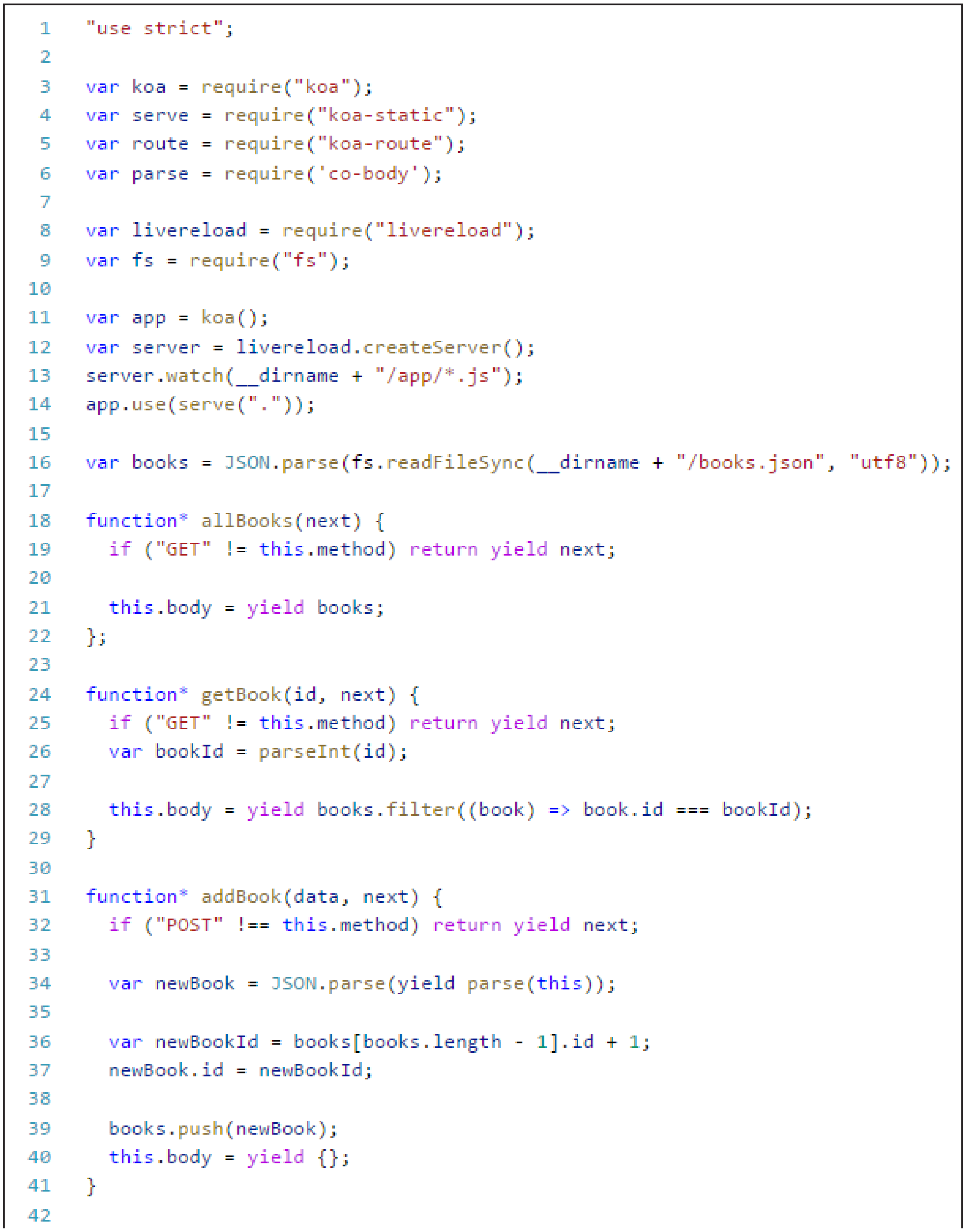
```
[
  {
    "id": 1,
    "name": "The Alchemist",
    "author": "Paulo Coelho",
    "price": 300,
    "publishedYear": 1993
  },
  {
    "id": 2,
    "name": "Harry Porter and Philosopher's Stone",
    "author": "JK Rowling",
    "price": 250,
    "publishedYear": 1997
  }
]
```

In addition to the NPM packages installed in Chapter 5 (while setting up the environment), several more packages are required to build the APIs. The following commands install these packages:

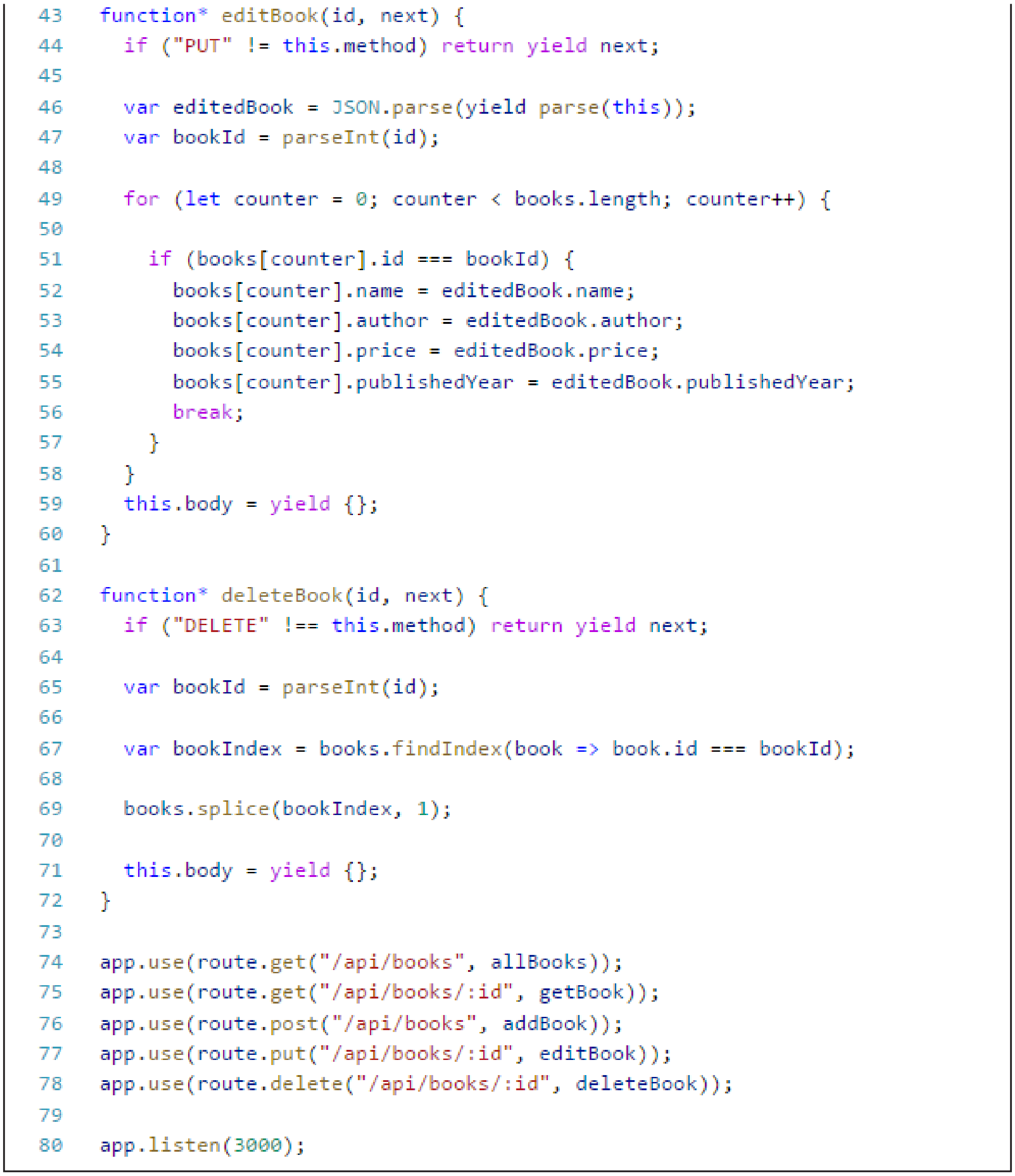
- npm install co-body --save
- npm install koa-route --save

The code of the server file is available in the file *server.js,* located in the folder *server* in the downloadable code of this chapter. You can copy the contents of this file and replace it in your server.js file. The following listing shows code of this file:

Listing 29.2: *Modified server.js*



```
"use strict";

var koa = require("koa");
var serve = require("koa-static");
var route = require("koa-route");
var parse = require('co-body');

var livereload = require("livereload");
var fs = require("fs");

var app = koa();
var server = livereload.createServer();
server.watch(__dirname + "/app/*.js");
app.use(serve("."));

var books = JSON.parse(fs.readFileSync(__dirname + "/books.json", "utf8"));

function* allBooks(next) {
  if ("GET" != this.method) return yield next;

  this.body = yield books;
};

function* getBook(id, next) {
  if ("GET" != this.method) return yield next;
  var bookId = parseInt(id);

  this.body = yield books.filter((book) => book.id === bookId);
}

function* addBook(data, next) {
  if ("POST" !== this.method) return yield next;

  var newBook = JSON.parse(yield parse(this));

  var newBookId = books[books.length - 1].id + 1;
  newBook.id = newBookId;

  books.push(newBook);
  this.body = yield {};
}

```



```
function* editBook(id, next) {
  if ("PUT" != this.method) return yield next;

  var editedBook = JSON.parse(yield parse(this));
  var bookId = parseInt(id);

  for (let counter = 0; counter < books.length; counter++) {

    if (books[counter].id === bookId) {
      books[counter].name = editedBook.name;
      books[counter].author = editedBook.author;
      books[counter].price = editedBook.price;
      books[counter].publishedYear = editedBook.publishedYear;
      break;
    }
  }
  this.body = yield {};
}

function* deleteBook(id, next) {
  if ("DELETE" !== this.method) return yield next;

  var bookId = parseInt(id);

  var bookIndex = books.findIndex(book => book.id === bookId);

  books.splice(bookIndex, 1);

  this.body = yield {};
}

app.use(route.get("/api/books", allBooks));
app.use(route.get("/api/books/:id", getBook));
app.use(route.post("/api/books", addBook));
app.use(route.put("/api/books/:id", editBook));
app.use(route.delete("/api/books/:id", deleteBook));

app.listen(3000);
```

You can test the APIs created before using them in the Angular application. For this, run the sample using the following command:

```
> npm run start
```

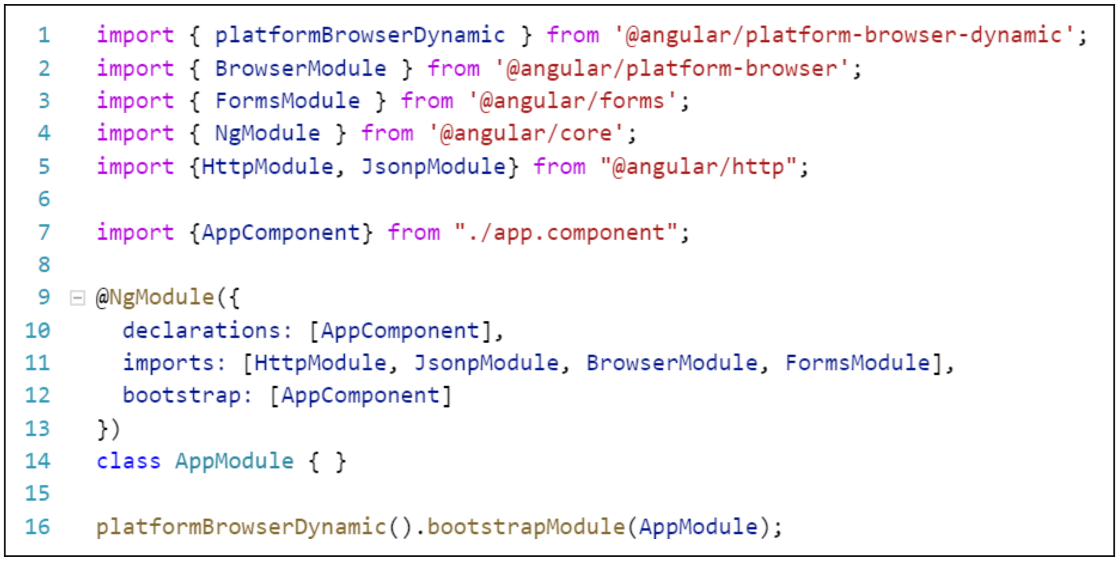
And open your favorite browser and change the URL to *http://localhost:3000/api/books*, ths should show the JSON payload containing the list of books on the browser. The next section is going to hit the same API using code.

Creating a Service to Interact with APIs

To consume these APIs in the Angular 2 application, a wrapper around the *XmlHttpRequest* object is needed. Angular 2 has the service *Http* wrapping up the low level logic of XHR and provides with an API that is easier to use. Methods of the service *Http* return RxJS Observables. The consuming application must subscribe to the observables to get their results and use them.

The *Http* service is defined in the package *@angular/http*. This package has an Angular 2 module named *HttpModule,* which gives access to the HTTP-related services defined in the module. This module must be imported in the application module to be able to use any of the HTTP based services. Listing 29.3 registers the providers while bootstrapping the Angular application:

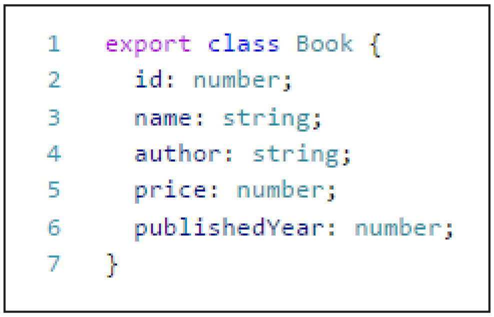
Listing 29.3: *Application module*



```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { NgModule } from '@angular/core';
import {HttpModule, JsonpModule} from "@angular/http";

import {AppComponent} from "./app.component";

@NgModule({
  declarations: [AppComponent],
  imports: [HttpModule, JsonpModule, BrowserModule, FormsModule],
  bootstrap: [AppComponent]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Before writing the service to read and write information on books, let's define a model class to represent the structure of the book object. This process makes good use of the types in TypeScript and makes the code less prone to errors. Listing 29.4 defines the class:

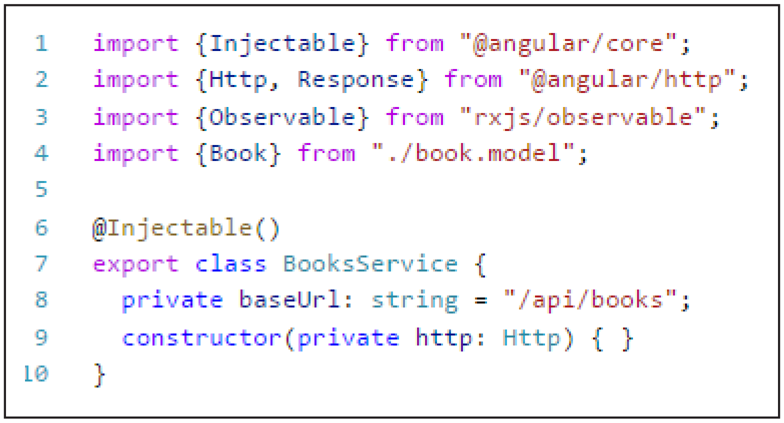
Listing 29.4: *Structure of Book objects, to be added to book.model.ts*



```
export class Book {
  id: number;
  name: string;
  author: string;
  price: number;
  publishedYear: number;
}
```

Let's define a service to interact with the APIs we created. The service needs a reference of the *Http* service. The code will be dealing with observables and it has to import the class *Observable* from RxJS. Listing 29.5 shows the skeleton of the service class:

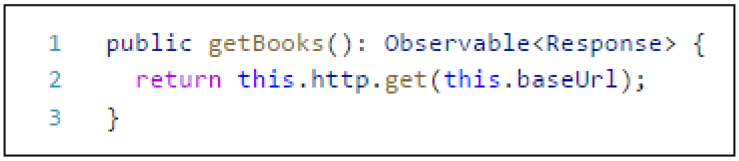
Listing 29.5: *Skeleton of BookService, to be added to books.service.ts*



```
import {Injectable} from "@angular/core";
import {Http, Response} from "@angular/http";
import {Observable} from "rxjs/observable";
import {Book} from "./book.model";

@Injectable()
export class BooksService {
  private baseUrl: string = "/api/books";
  constructor(private http: Http) { }
}
```

The base URL was assigned to a private variable to avoid usage of the strings repeatedly in the code. To fetch all books, an HTTP GET call must be made to the base URL itself. It will return an *Observable* object, which will call its subscribe callback when the asynchronous call is complete. The subscribe callback will be handled in a component, so the service method will return the observable object that it receives. Listing 29.6 shows the method to retrieve all books:

Listing 29.6: *getBooks method*

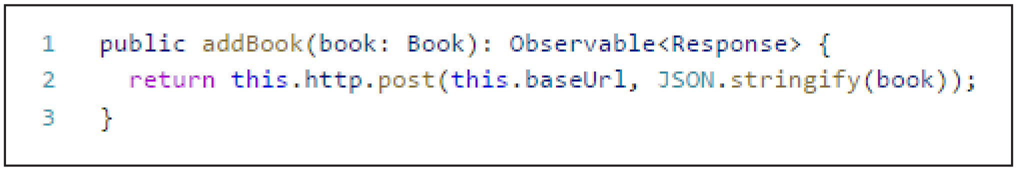


```
public getBooks(): Observable<Response> {
  return this.http.get(this.baseUrl);
}
```

The *get* method of the *Http* service returns a generic *Observable* object of *Response* type. The response object contains: status of the response, type of the response (basic, CORS, default, etc.), requested URL, response headers, methods to get the response data as text and as JSON, and other properties regarding HTTP response received from the request. The response object is passed to the callback function attached to the subscribe method, an example of which will follow shortly.

The *addBook* method accepts an object of the *Book* type we defined above and sends it to the *post* method of the *Http* service. This method, in turn, sends this object to the HTTP POST API to add the book to the list. This method also returns a generic object of *Observable<Response>* type. Listing 29.7 shows this method:

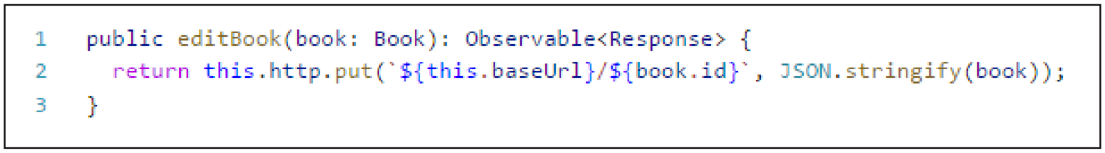
Listing 29.7: *addBook() method*



```
public addBook(book: Book): Observable<Response> {
  return this.http.post(this.baseUrl, JSON.stringify(book));
}
```

The method for editing a book accepts an object of the modified book and sends it to *put* method of the *Http* service. To modify the correct book, the API needs the ID of the book. This method also returns an object of type *Observable<Response>*. Listing 29.8 shows this method:

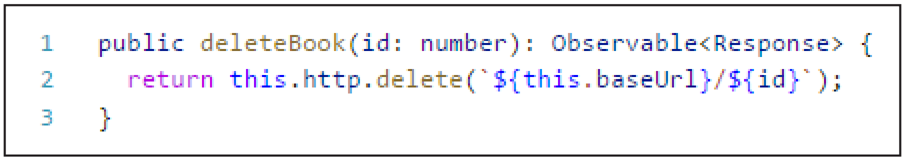
Listing 29.8: *editBook method*



```
public editBook(book: Book): Observable<Response> {
  return this.http.put(`${this.baseUrl}/${book.id}`, JSON.stringify(book));
}
```

The last method needed in the service is to delete a book. The API needs the book ID and, like other methods, the *http.delete* method returns an object of type *Observable<Response>*. Listing 29.9 defines this method:

Listing 29.9: *deleteBook method*

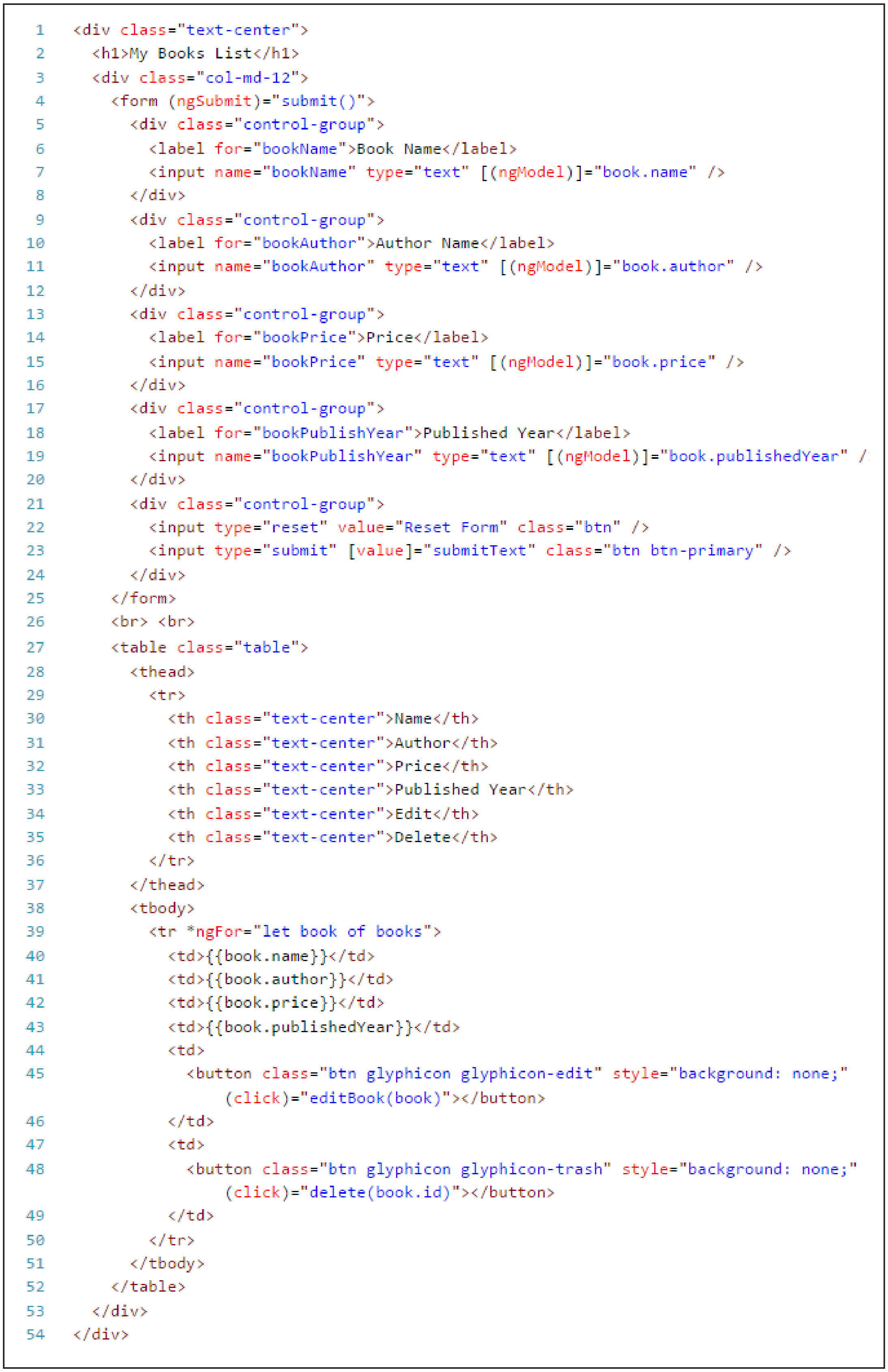


```
public deleteBook(id: number): Observable<Response> {
  return this.http.delete(`${this.baseUrl}/${id}`);
}
```

Component Consuming the BooksService

The application needs a component to use the above service and provide an interface for the user to work with the data. This component consists of a form and a table: the form will be used to add or edit a book; the table lists the books and provides buttons to delete a book and edit a book. Listing 29.10 shows the HTML template of the component:

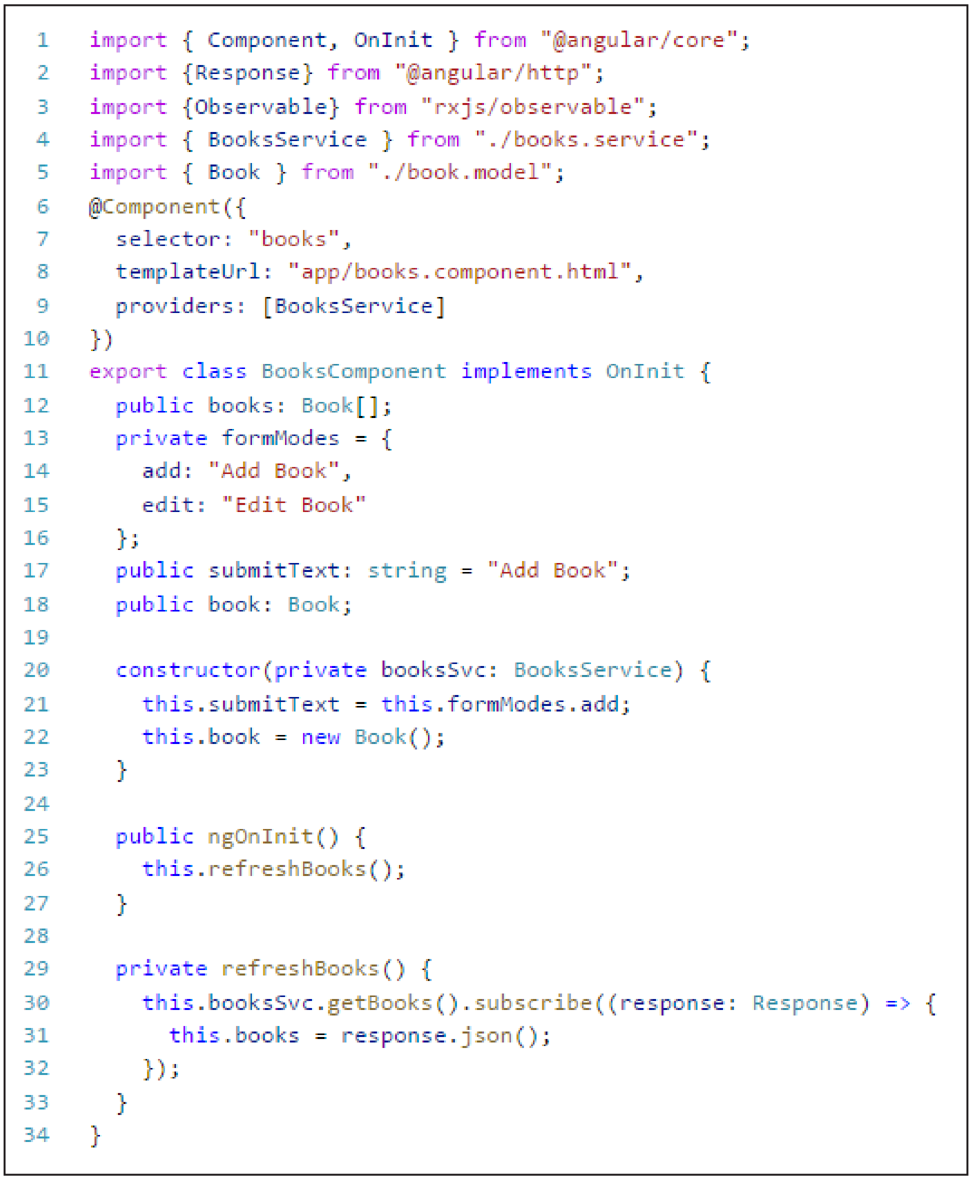
Listing 29.10: *Template of BooksComponent*



```
<div class="text-center">
  <h1>My Books List</h1>
  <div class="col-md-12">
    <form (ngSubmit)="submit()">
      <div class="control-group">
        <label for="bookName">Book Name</label>
        <input name="bookName" type="text" [(ngModel)]="book.name" />
      </div>
      <div class="control-group">
        <label for="bookAuthor">Author Name</label>
        <input name="bookAuthor" type="text" [(ngModel)]="book.author" />
      </div>
      <div class="control-group">
        <label for="bookPrice">Price</label>
        <input name="bookPrice" type="text" [(ngModel)]="book.price" />
      </div>
      <div class="control-group">
        <label for="bookPublishYear">Published Year</label>
        <input name="bookPublishYear" type="text" [(ngModel)]="book.publishedYear" /
      </div>
      <div class="control-group">
        <input type="reset" value="Reset Form" class="btn" />
        <input type="submit" [value]="submitText" class="btn btn-primary" />
      </div>
    </form>
    <br> <br>
    <table class="table">
      <thead>
        <tr>
          <th class="text-center">Name</th>
          <th class="text-center">Author</th>
          <th class="text-center">Price</th>
          <th class="text-center">Published Year</th>
          <th class="text-center">Edit</th>
          <th class="text-center">Delete</th>
        </tr>
      </thead>
      <tbody>
        <tr *ngFor="let book of books">
          <td>{{book.name}}</td>
          <td>{{book.author}}</td>
          <td>{{book.price}}</td>
          <td>{{book.publishedYear}}</td>
          <td>
            <button class="btn glyphicon glyphicon-edit" style="background: none;"
                (click)="editBook(book)"></button>
          </td>
          <td>
            <button class="btn glyphicon glyphicon-trash" style="background: none;"
                (click)="delete(book.id)"></button>
          </td>
        </tr>
      </tbody>
    </table>
  </div>
</div>
```

The class of the component obtains the object of the service through DI and loads all books when the component is initialized. To load the books, it calls the *getBooks* method of the service *BooksService*, subscribes to the observable returned from the method, and gets the result in the subscribed callback method. Listing 29.11 shows the imports required and defines the component with its *ngOnInit* lifecycle hook:

Listing 29.11: *Code of BooksComponent*



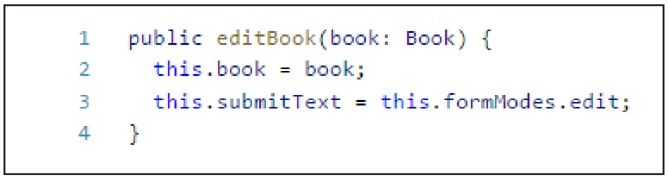
```
import { Component, OnInit } from "@angular/core";
import {Response} from "@angular/http";
import {Observable} from "rxjs/observable";
import { BooksService } from "./books.service";
import { Book } from "./book.model";
@Component({
  selector: "books",
  templateUrl: "app/books.component.html",
  providers: [BooksService]
})
export class BooksComponent implements OnInit {
  public books: Book[];
  private formModes = {
    add: "Add Book",
    edit: "Edit Book"
  };
  public submitText: string = "Add Book";
  public book: Book;

  constructor(private booksSvc: BooksService) {
    this.submitText = this.formModes.add;
    this.book = new Book();
  }

  public ngOnInit() {
    this.refreshBooks();
  }

  private refreshBooks() {
    this.booksSvc.getBooks().subscribe((response: Response) => {
      this.books = response.json();
    });
  }
}
```

A few things to note in the listing 29.11:

- The constructor assigns the field *submitText* with Add Book, as the form will be in add state by default. This text will be changed when a user edits a book. The *formModes* property in the *BooksComponent* class holds the texts corresponding to both the modes of the form; this is done to avoid repetition of the strings in the component
- The constructor of the component creates a new object of Book. The form of the component uses this object to accept values from the user
- The method *refreshBooks* calls the *json* method of the *Response* object. This gives the list of books received from the Koa REST API

When a user selects the edit option in the table, the selected book object must be set to the form, and the text of the submit button of the form must be changed to "Edit Book". If you observe the template of the component, the book object bound to a row is passed to the *editBook* method. The same object will be assigned to the component's private field *book*.

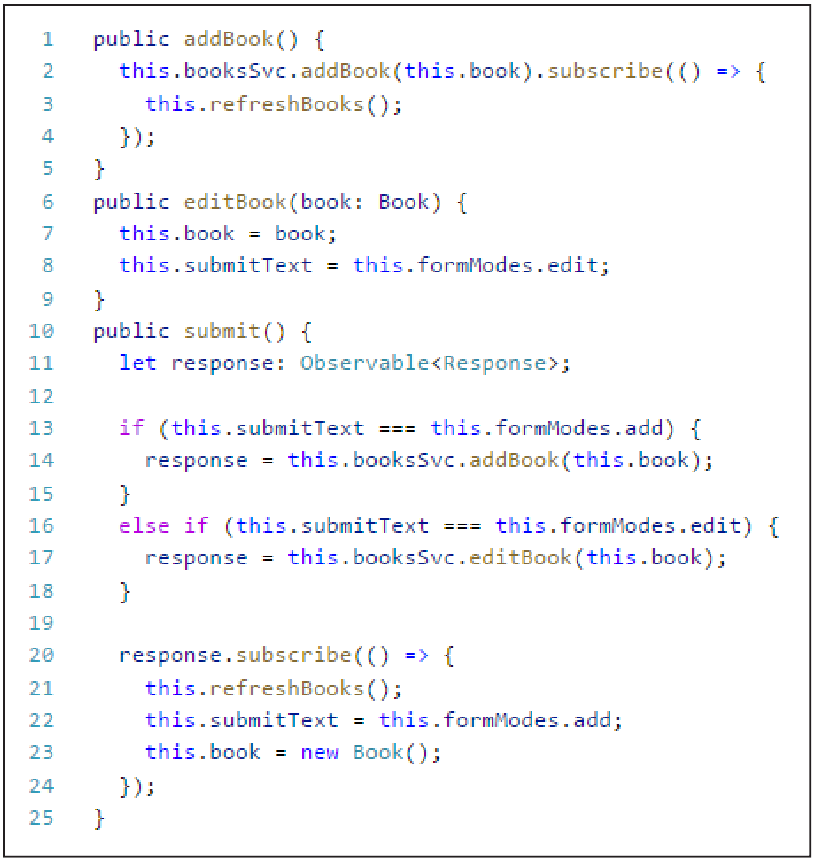
Listing 29.12: *editBook method*



```
public editBook(book: Book) {
  this.book = book;
  this.submitText = this.formModes.edit;
}
```

When a user submits the form, the component will call add or edit methods of *BooksService,* depending upon the text of the button. After receiving the response from the service, the form must be reset to its initial state. Listing 29.13 shows the methods (add, edit, and submit) of the component:

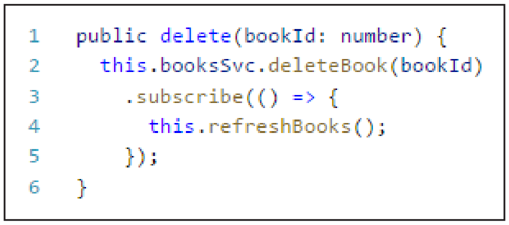
Listing 29.13: *addBook, editBook and submit methods*



```
public addBook() {
  this.booksSvc.addBook(this.book).subscribe(() => {
    this.refreshBooks();
  });
}
public editBook(book: Book) {
  this.book = book;
  this.submitText = this.formModes.edit;
}
public submit() {
  let response: Observable<Response>;

  if (this.submitText === this.formModes.add) {
    response = this.booksSvc.addBook(this.book);
  }
  else if (this.submitText === this.formModes.edit) {
    response = this.booksSvc.editBook(this.book);
  }

  response.subscribe(() => {
    this.refreshBooks();
    this.submitText = this.formModes.add;
    this.book = new Book();
  });
}
```

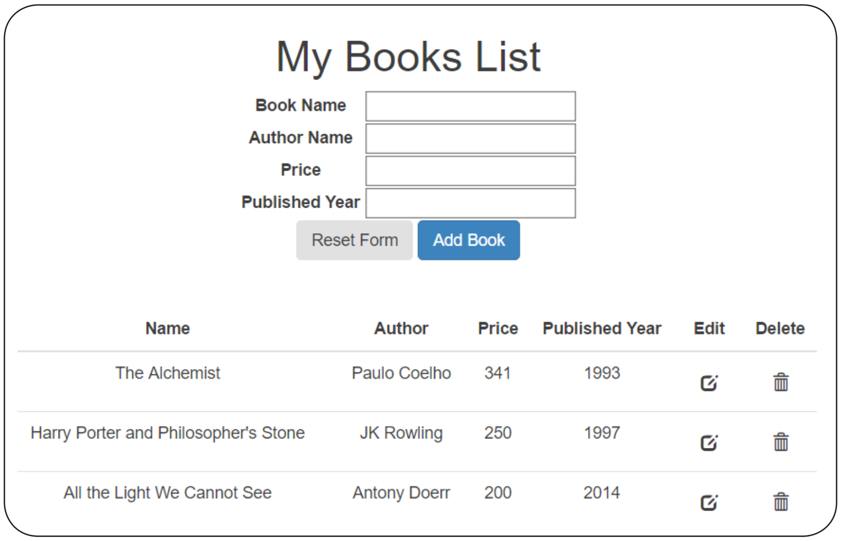
Finally, the *delete* method accepts the book ID and calls the service method to delete the book. Once the call is successful, it refreshes the data on the page. Listing 29.14 shows this method:

Listing 29.14: *delete method*



```
public delete(bookId: number) {
  this.booksSvc.deleteBook(bookId)
    .subscribe(() => {
      this.refreshBooks();
    });
}
```

At this stage, the application looks like the following:



My Books List

Book Name

Author Name

Price

Published Year

Name	Author	Price	Published Year	Edit	Delete
The Alchemist	Paulo Coelho	341	1993	✎	🗑
Harry Porter and Philosopher's Stone	JK Rowling	250	1997	✎	🗑
All the Light We Cannot See	Antony Doerr	200	2014	✎	🗑

Figure 29.1: *Books List page*

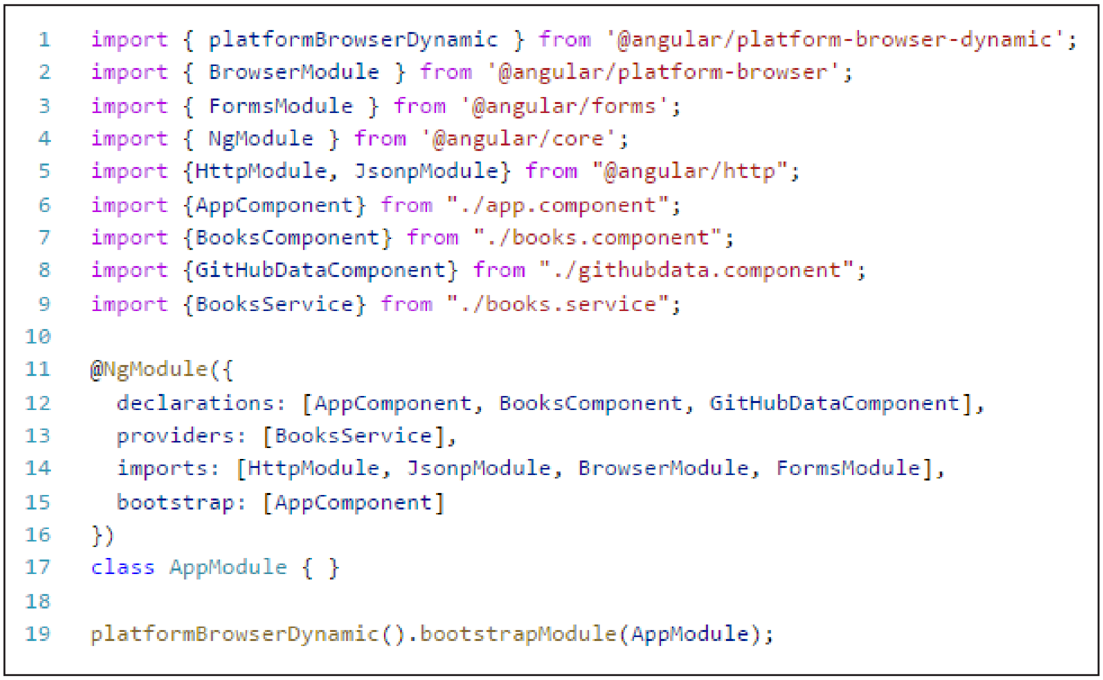
GitHub Repo list using JSONP

The APIs hosted on a different domain can be accessed either by enabling CORS (Cross Origin Resource Sharing) on the APIs or JSONP. If the API is CORS-enabled, *Http* service can call it and access it using the process in the previous section. If the API is not CORS-enabled, it can be consumed using JSONP.

The @angular/http module of Angular 2 includes the APIs to interact with the APIs using JSONP. All of the services created to support JSONP are in the module *JsonpModule*. This module must be imported in order to use JSONP in Angular 2.

Let's consume the GitHub API to get a list of repos by technology and language. As the GitHub API is hosted on a different domain, it must be consumed using JSONP. As the component must only display the list of repos, the JSONP operation is performed in the component itself. First, let's import the *JsonpModule* into the application module. Listing 29.15 shows the modified content of the main module:

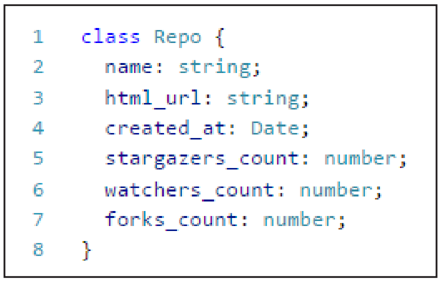
Listing 29.15: *Modified AppModule*



```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { NgModule } from '@angular/core';
import {HttpModule, JsonpModule} from "@angular/http";
import {AppComponent} from "./app.component";
import {BooksComponent} from "./books.component";
import {GitHubDataComponent} from "./githubdata.component";
import {BooksService} from "./books.service";

@NgModule({
  declarations: [AppComponent, BooksComponent, GitHubDataComponent],
  providers: [BooksService],
  imports: [HttpModule, JsonpModule, BrowserModule, FormsModule],
  bootstrap: [AppComponent]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Before calling the service, it is best to have a model class with the set of properties needed to display on the page. Listing 29.16 shows the *Repo* class with the required properties:

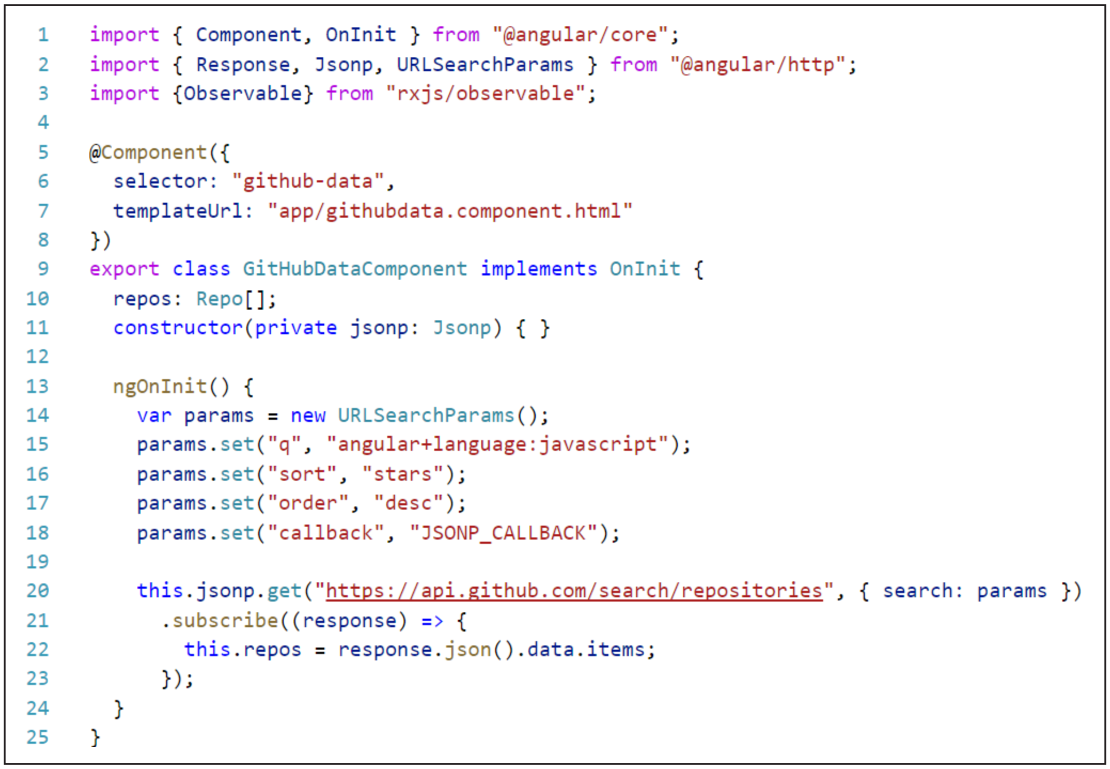
Listing 29.16: *The model class Repo*



```
class Repo {
  name: string;
  html_url: string;
  created_at: Date;
  stargazers_count: number;
  watchers_count: number;
  forks_count: number;
}
```

The component needs to implement the *ngOnInit* lifecycle hook to fetch the data and assign it to a field in the component. The GitHub API needs a couple of parameters to be sent with the request. The query string parameters can be built using the *URLSearchParams* class imported earlier. In addition to the input parameters with data, the API needs another parameter to recognize that it is a JSONP request. The *ngOnInit* method in the listing 29.17 adds these parameters and calls the API using the *Jsonp* service:

Listing 29.17: *Code of GithubDataComponent*

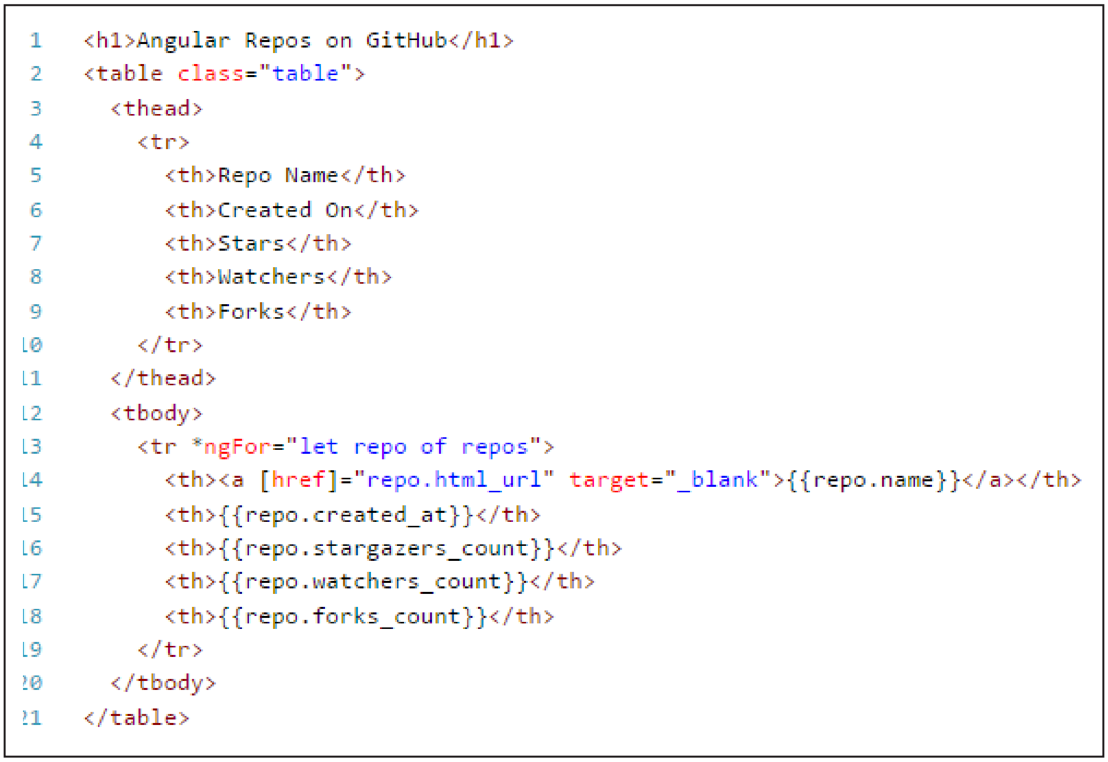


```
import { Component, OnInit } from "@angular/core";
import { Response, Jsonp, URLSearchParams } from "@angular/http";
import {Observable} from "rxjs/observable";

@Component({
  selector: "github-data",
  templateUrl: "app/githubdata.component.html"
})
export class GitHubDataComponent implements OnInit {
  repos: Repo[];
  constructor(private jsonp: Jsonp) { }

  ngOnInit() {
    var params = new URLSearchParams();
    params.set("q", "angular+language:javascript");
    params.set("sort", "stars");
    params.set("order", "desc");
    params.set("callback", "JSONP_CALLBACK");

    this.jsonp.get("https://api.github.com/search/repositories", { search: params })
      .subscribe((response) => {
        this.repos = response.json().data.items;
      });
  }
}
```

To make JSONP work, the module *JsonpModule* has to be added to the imports of the application module. It is already done in the listing 29.3, you may revisit the listing to see how the module is added.

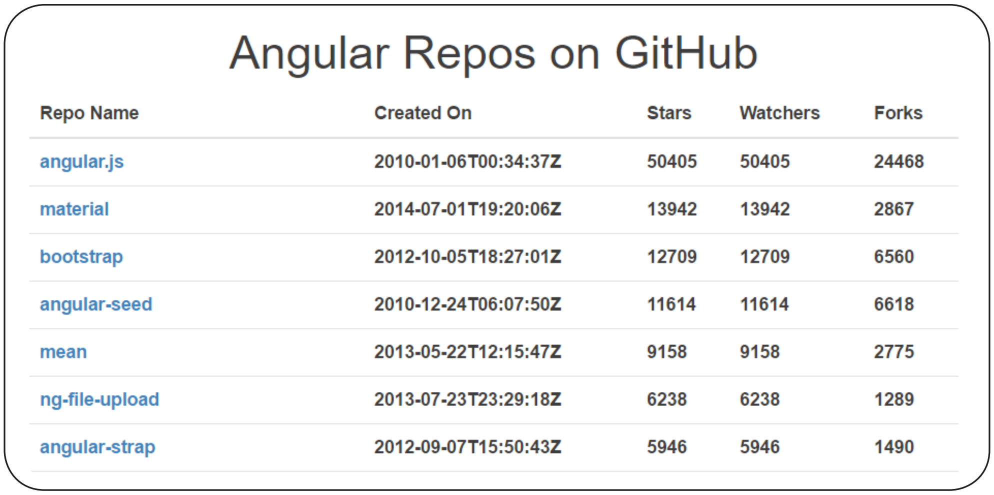
The template of the component is straightforward. It has a table with the fields to show the values received from the GitHub API. Listing 29.18 shows the template:

Listing 29.18: *Template of GithubDataComponent*



```
<h1>Angular Repos on GitHub</h1>
<table class="table">
  <thead>
    <tr>
      <th>Repo Name</th>
      <th>Created On</th>
      <th>Stars</th>
      <th>Watchers</th>
      <th>Forks</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let repo of repos">
      <th><a [href]="repo.html_url" target="_blank">{{repo.name}}</a></th>
      <th>{{repo.created_at}}</th>
      <th>{{repo.stargazers_count}}</th>
      <th>{{repo.watchers_count}}</th>
      <th>{{repo.forks_count}}</th>
    </tr>
  </tbody>
</table>
```

This page would look like the following figure:



Angular Repos on GitHub

Repo Name	Created On	Stars	Watchers	Forks
angular.js	2010-01-06T00:34:37Z	50405	50405	24468
material	2014-07-01T19:20:06Z	13942	13942	2867
bootstrap	2012-10-05T18:27:01Z	12709	12709	6560
angular-seed	2010-12-24T06:07:50Z	11614	11614	6618
mean	2013-05-22T12:15:47Z	9158	9158	2775
ng-file-upload	2013-07-23T23:29:18Z	6238	6238	1289
angular-strap	2012-09-07T15:50:43Z	5946	5946	1490

Figure 29.2: *Page showing Github repos*

Conclusion

Angular 2 comes with built-in support for querying the REST APIs over HTTP, as well as JSONP. This chapter examined how to perform CRUD operations using the services in Angular 2 with examples. A future chapter will discuss how to consume secured APIs.

Chapter 30

Using Built In pipes and Parameterized Pipes

PIPES HELP YOU REPEAT filtering or formatting of your output. They can be used in template expressions to improve the usability of views. Pipes transform displayed values within a template. A pipe takes in data as input and transforms it to a desired output as per the expectation of the data representation on the view. Pipes are more useful when the data is represented using currency, uppercase and lowercase, etc.

This chapter discusses the built-in and parameterized pipes provided in Angular 2. Some of the pipes discussed are Case pipes (uppercase and lowercase), Decimal, Currency, Percentage and JSON pipes.

The syntax of a pipe is very simple. An expression is followed by the pipe symbol (|), which is followed by the pipe name:

```
{{<EXPRESSION>|<PIPE_NAME>}}
```

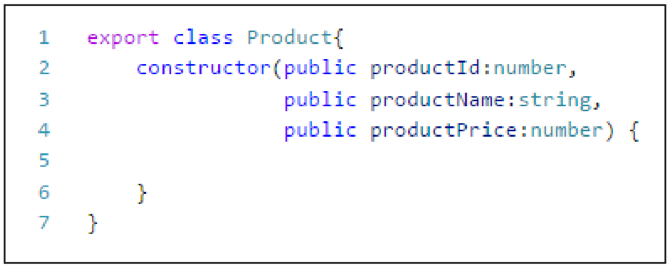
The EXPRESSION is a property declared in the Angular 2 component class which will be used for databinding. The PIPE_NAME can be a built-in or a custom pipe.

A Pipe can also accept any number of optional parameters separated by a colon (:). If a Pipe accepts multiple parameters, then each parameter is separated using a colon (:). The following syntax represents a parameterized pipe.

```
{{<EXPRESSION>|<PIPE_NAME>:Parameter1:Parameter2}}
```

To use the code-snippets of this chapter, create a new project using the steps discussed in chapter 5. A new TypeScript file product.model.ts will be added to the app folder in this project as shown in the following listing.

Listing 30.1: *The Product Model class*

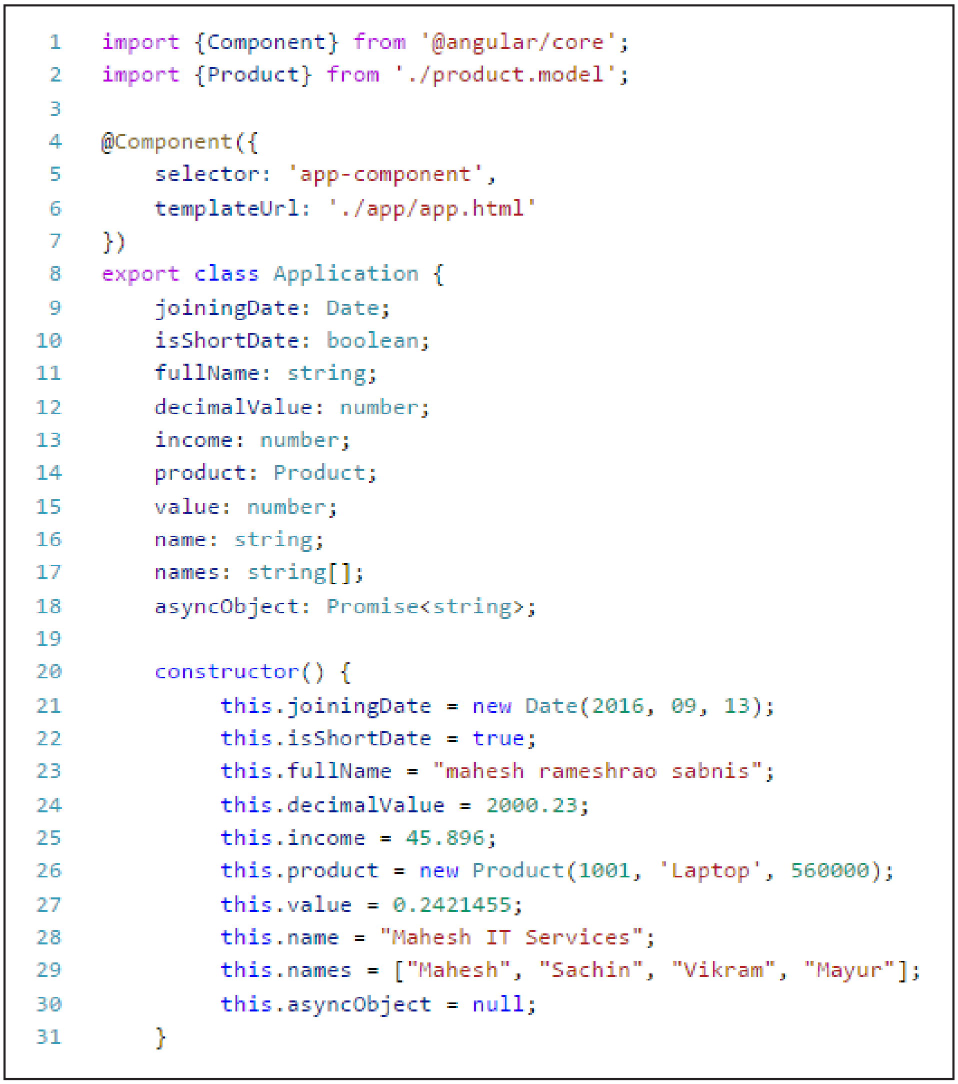


```
export class Product{
    constructor(public productId:number,
                public productName:string,
                public productPrice:number) {

    }
}
```

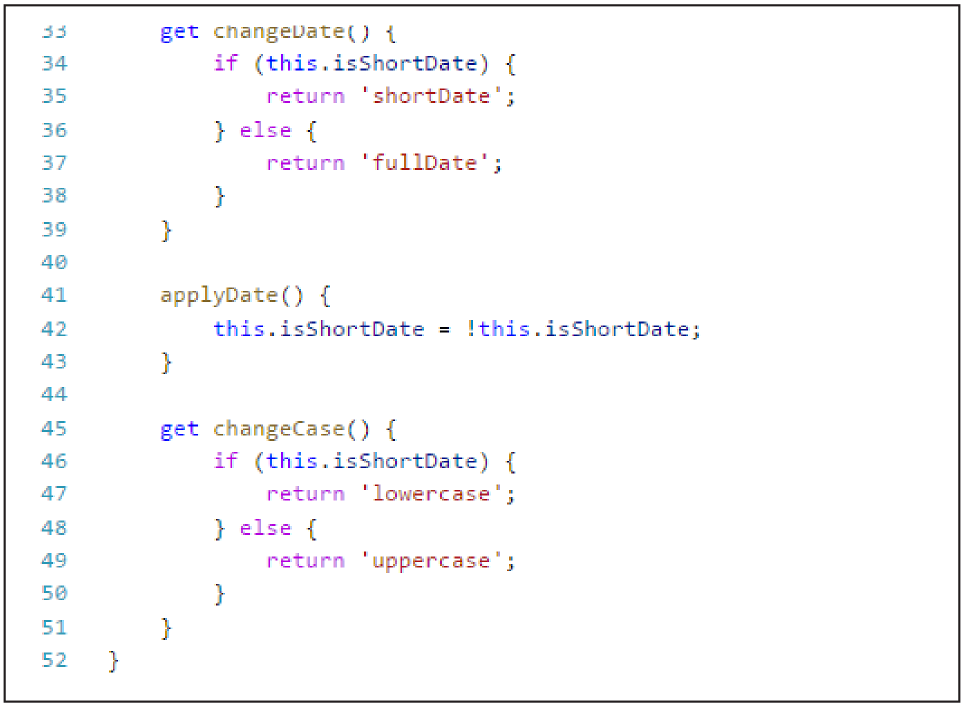
This class will be used while using the JSON pipe.

Add a component class application.components.ts in the app folder which will define properties to be exposed to the user interface. These properties will be used for databinding with pipes. The code for this typescript class is shown in Listing 30.2.

Listing 30.2: *The Component class with declarations*



```
import {Component} from '@angular/core';
import {Product} from './product.model';

@Component({
    selector: 'app-component',
    templateUrl: './app/app.html'
})
export class Application {
    joiningDate: Date;
    isShortDate: boolean;
    fullName: string;
    decimalValue: number;
    income: number;
    product: Product;
    value: number;
    name: string;
    names: string[];
    asyncObject: Promise<string>;

    constructor() {
        this.joiningDate = new Date(2016, 09, 13);
        this.isShortDate = true;
        this.fullName = "mahesh rameshrao sabnis";
        this.decimalValue = 2000.23;
        this.income = 45.896;
        this.product = new Product(1001, 'Laptop', 560000);
        this.value = 0.2421455;
        this.name = "Mahesh IT Services";
        this.names = ["Mahesh", "Sachin", "Vikram", "Mayur"];
        this.asyncObject = null;
    }
```



```
    get changeDate() {
        if (this.isShortDate) {
            return 'shortDate';
        } else {
            return 'fullDate';
        }
    }

    applyDate() {
        this.isShortDate = !this.isShortDate;
    }

    get changeCase() {
        if (this.isShortDate) {
            return 'lowercase';
        } else {
            return 'uppercase';
        }
    }
}
```

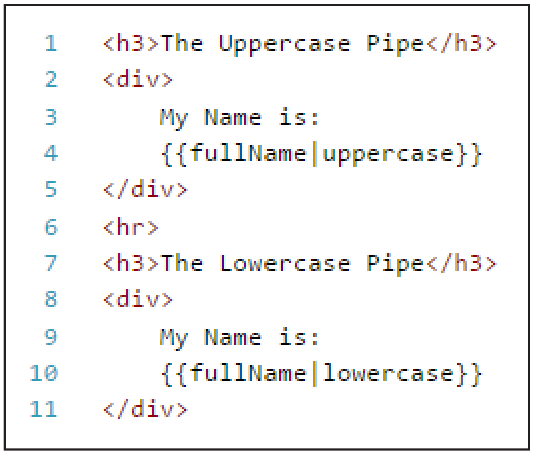
As explained in code, there are various properties declared with their values initialized in the constructor. We willuse these properties and functions later for databinding on the View.

Add a HTML page app.html in the **app** folder. This page will define HTML elements to test out Pipes and their output.

Uppercase and Lowercase pipes

These pipes as the name suggests, are used to change the case of the string input, and output it in uppercase or lowercase. Consider the following HTML code listing:

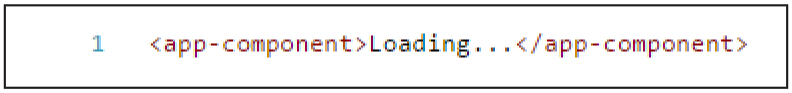
Listing 30.3: *Using uppercase and lowercase pipe*



```
<h3>The Uppercase Pipe</h3>
<div>
    My Name is:
    {{fullName|uppercase}}
</div>
<hr>
<h3>The Lowercase Pipe</h3>
<div>
    My Name is:
    {{fullName|lowercase}}
</div>
```

This listing uses the **fullName** property with a pipe applied on it. Add the following tag in index.html in the body section.

Listing 30.4



```
<app-component>Loading...</app-component>
```

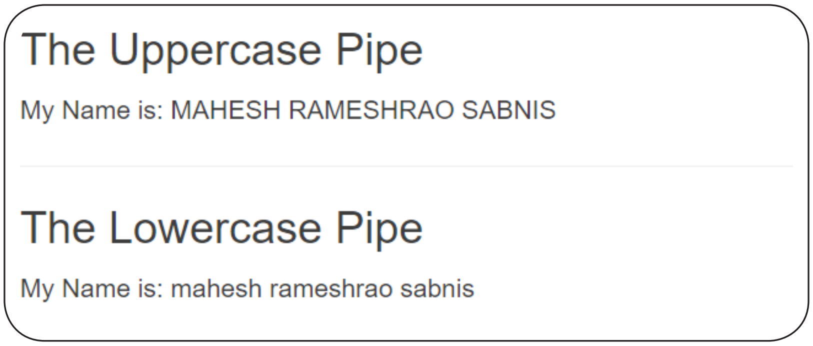
To view the output, right-click on the index.html in VS Code and select Open in Command Prompt. Run the following command on this command prompt

```
npm run start
```

Open the browser and enter the following URL:

http://localhost:3000

The following result will show the output for the Uppercase and Lowercase pipes.



The Uppercase Pipe

My Name is: MAHESH RAMESHRAO SABNIS

The Lowercase Pipe

My Name is: mahesh rameshrao sabnis

Figure 30.1: *The output of the uppercase and lowercase Pipe*

The number, currency and percent Pipe

While displaying numeric data on the view, one should consider the various flavors of numeric data representation. For example, numeric values can be displayed using decimal format, or sometimes using the currency formats for localization purposes.

THE NUMBER PIPE

The number pipe defines grouping and sizing for numbers. See the following syntax:

```
{{EXPRESSION|number[:digitInfo]}}
```

In this case, the EXPRESSION is a model property number to be displayed, and the digitInfo has the following digit formats

```
{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}
```

This expression has the following:

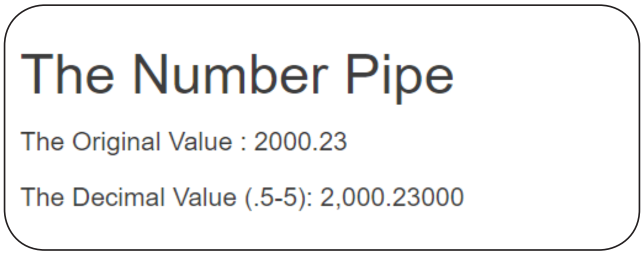
- **minIntegerDigits:** The minimum number of integer digits to use. The default is 1.
- **minFractionDigits:** The minimum number of digits after the fraction. The default is 0.
- **maxFractionDigits:** The maximum number of digits after the fraction. The default is 3.

There is a **decimalValue** property declared in the component class with a hard-coded value as 2000.23. We will use this property for databinding on the view as seen in the following listing:

Listing 30.5: *Using number pipe*

```
<h2>The Number Pipe </h2>
<p>The Original Value : {{decimalValue}}</p>
<p>The Decimal Value (.5-5): {{decimalValue | number:'.5-5'}}</p>
```

Here **number** pipe has been used with formatting as '.5-5'. After viewing it in browser, the output will be as follows:



The Number Pipe

The Original Value : 2000.23

The Decimal Value (.5-5): 2,000.23000

Figure 30.2: *The output of Number Pipe with formatting .5-5*

The Original value 2000.23 is displayed as 2,000.23000 (with 5 digits after the decimal point).

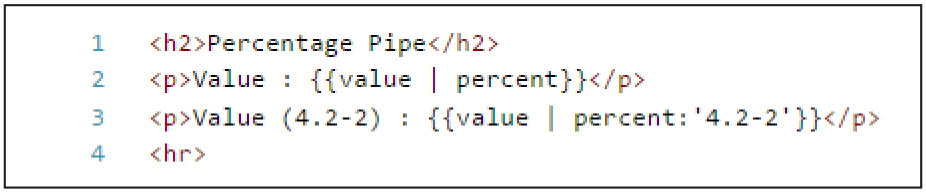
THE PERCENT PIPE

The percent pipe is derived from the number pipe, hence it retains all the formatting features of it. This pipe formats the output in the local percent format. The syntax of using percentage pipe is as following:

```
{{EXPRESSION|perent[:digitInfo]}}
```

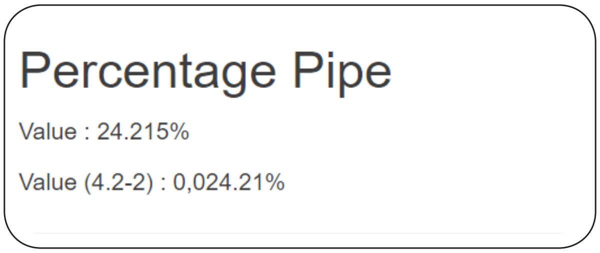
The following listing shows how to apply the percent pipe:

Listing 30.6: *Using percent pipe*



```
<h2>Percentage Pipe</h2>
<p>Value : {{value | percent}}</p>
<p>Value (4.2-2) : {{value | percent:'4.2-2'}}</p>
<hr>
```

There's a **value** property defined in the component class with the default as 0.2421455. After applying the percent pipe in Listing 30.6, the output will be as shown in Figure 30.3:



Percentage Pipe

Value : 24.215%

Value (4.2-2) : 0,024.21%

Figure 30.3: *The output of the percent pipe*

The first output displays the conversion of the value in local percent format, whereas the second output uses the formatting digits from the number pipe.

THE CURRENCY PIPE

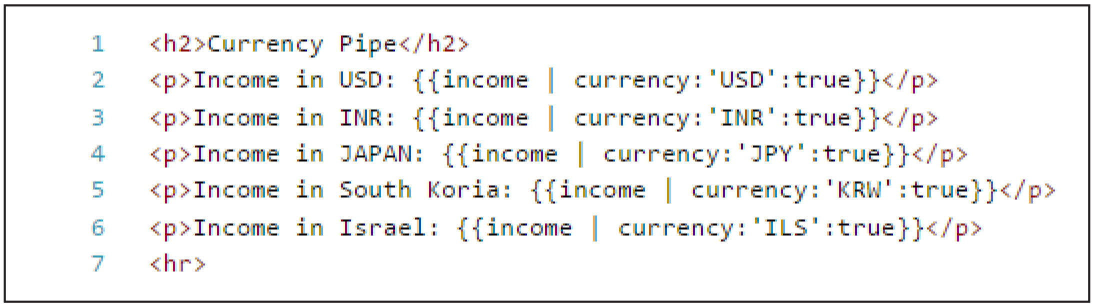
The currency pipe is used to format the number in a local currency format. The local currency name can be passed as a parameter to the currency pipe, hence we can use this pipe as a **parameterized pipe**. Since this pipe is derived from the number pipe, the same digit formatting can be used. The syntax for the percent pipe is as follows:

```
{{EXPRESSION|currency[:currencyCode[:symbolDisplay[:digitInfo]]]}}
```

In this syntax, the **currencyCode** is the ISO 4217 country code for the currency e.g. for US dollar we will use USD, and for Euro, we will use EUR. The **symbolDisplay** is a Boolean field indicating whether to use currency symbol e.g. $ or €, or use country code e.g. EUR or USD.

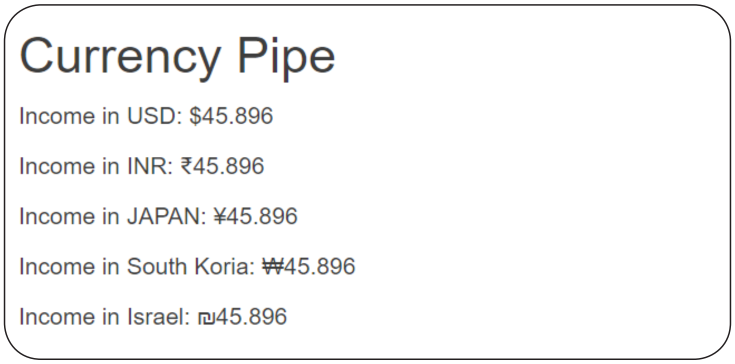
In the component, there's an **income** property with value as 45.896. This property will be used in the View as shown in listing 30.7.

Listing 30.7: *Using currency pipe*



```
<h2>Currency Pipe</h2>
<p>Income in USD: {{income | currency:'USD':true}}</p>
<p>Income in INR: {{income | currency:'INR':true}}</p>
<p>Income in JAPAN: {{income | currency:'JPY':true}}</p>
<p>Income in South Koria: {{income | currency:'KRW':true}}</p>
<p>Income in Israel: {{income | currency:'ILS':true}}</p>
<hr>
```

As you can see, the currencyCode and symbolDisplay are passed as parameters to the currency filter. The output of the filter is as shown in Figure 30.4:



Currency Pipe

Income in USD: $45.896

Income in INR: ₹45.896

Income in JAPAN: ¥45.896

Income in South Koria: ₩45.896

Income in Israel: ₪45.896

Figure 30.4: *Output of the currency pipe*

The slice pipe

The slice pipe is equivalent to the Array.prototype.slice() function of Array object in JavaScript, or like String.prototype.slice() which is used to slice a collection or string data. This pipe is more useful when we want index based data slicing before showing it on the view as output. The syntax of the slice pipe is as follows:

```
{{EXPRESSION|slice:start[:end]}}
```

While using slice pipe, the **start** index is mandatory, whereas the **end** index is optional. The default value for the **end** index is the last index of the input value of the EXPRESSION.

In the component, declare a name property and names array property as shown in listing 30.8:

Listing 30.8: *string and array declaration*

```
this.name = "Mahesh IT Services";
this.names = ["Mahesh", "Sachin", "Vikram", "Mayur"];
```

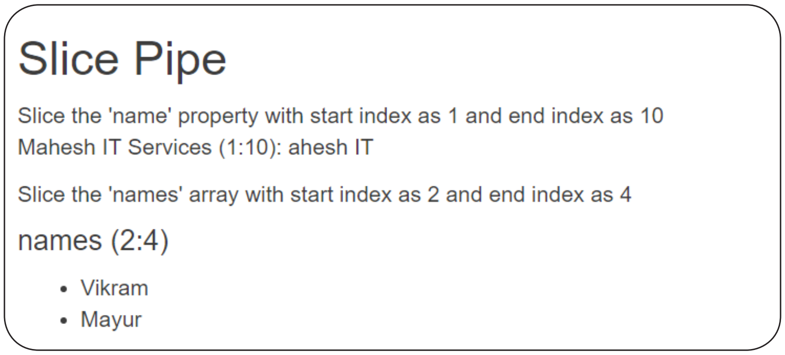
The slice pipe is used for displaying desired output of these properties.

The following HTML listing shows the use of slice pipe:

Listing 30.9: *Using slice pipe*

```
<h2>Slice Pipe</h2>
<p>{{name}} (1:10): {{name | slice:1:10}}</p>
<h4>names (2:4)</h4>
<ul>
    <li *ngFor="let n of names | slice:2:4">{{n}}</li>
</ul>
<hr>
```

Here start index and end index have values as 1 and 10 respectively for the **name** property. The **ngFor** directive has been used for slicing **names** array with start and end index as 2 and 4. This results to the following output:



Slice Pipe

Slice the 'name' property with start index as 1 and end index as 10

Mahesh IT Services (1:10): ahesh IT

Slice the 'names' array with start index as 2 and end index as 4

names (2:4)

- Vikram
- Mayur

Figure 30.5 Output of the slice pipe

The slice applied on string **Mahesh IT Services** with start index as 1, and end as 10 will return **ahesh IT .** The names array will be sliced from the start index.

The date pipe

The date pipe is used to format a date value as string based on the request format, which is passed as a parameter to it. The zone for the date and time is based on the settings of the end-user's machine settings. Here's the syntax of the date pipe

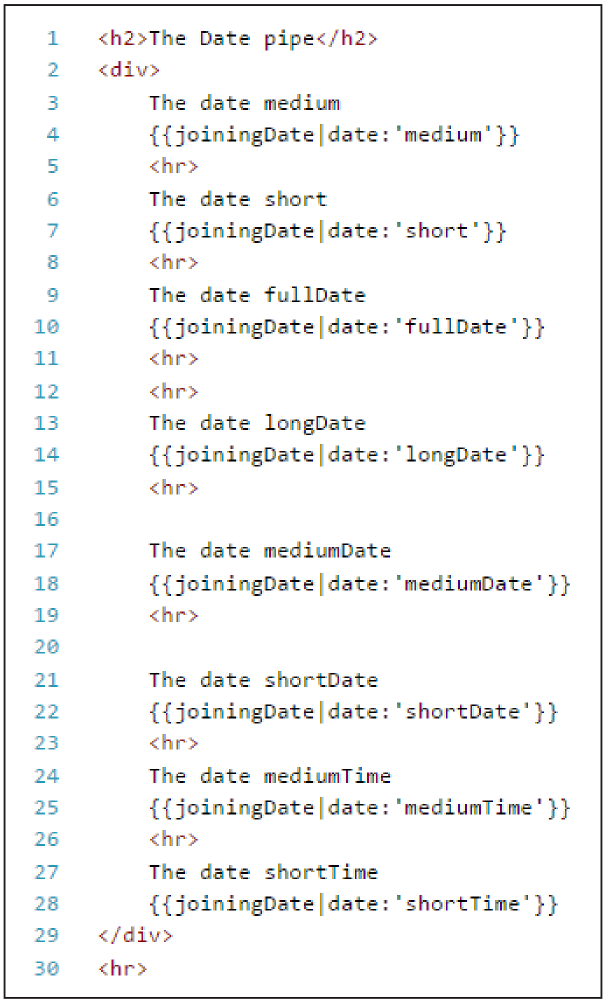
```
{{EXPRESSION|date[:format]}}
```

The format value is used to display the date with specific local format. The various formats available are as follows:

- 'medium'
- 'short'
- 'fullDate'
- 'longDate'
- 'mediumDate'
- 'shortDate'
- 'mediumTime'
- 'shortTime'

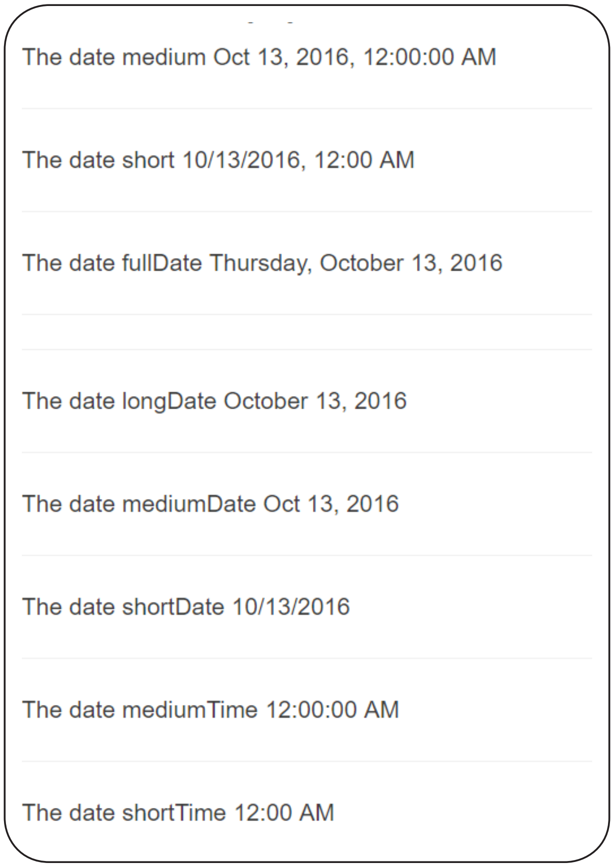
Listing 30.10 uses the date pipe on the joiningDate property of the component.

Listing 30.10: *Using date pipe*



```
<h2>The Date pipe</h2>
<div>
    The date medium
    {{joiningDate|date:'medium'}}
    <hr>
    The date short
    {{joiningDate|date:'short'}}
    <hr>
    The date fullDate
    {{joiningDate|date:'fullDate'}}
    <hr>
    <hr>
    The date longDate
    {{joiningDate|date:'longDate'}}
    <hr>

    The date mediumDate
    {{joiningDate|date:'mediumDate'}}
    <hr>

    The date shortDate
    {{joiningDate|date:'shortDate'}}
    <hr>
    The date mediumTime
    {{joiningDate|date:'mediumTime'}}
    <hr>
    The date shortTime
    {{joiningDate|date:'shortTime'}}
</div>
<hr>
```

The following image shows the output of the date pipe:



The date medium Oct 13, 2016, 12:00:00 AM

The date short 10/13/2016, 12:00 AM

The date fullDate Thursday, October 13, 2016

The date longDate October 13, 2016

The date mediumDate Oct 13, 2016

The date shortDate 10/13/2016

The date mediumTime 12:00:00 AM

The date shortTime 12:00 AM

Figure 30.6: *Output of the date pipe*

The JSON pipe

This is the simplest of all pipes in Angular 2. This pipe accepts an object, and outputs it in JSON format. The JSON pipe has the following syntax:

```
{{EXPRESSION|json}}
```

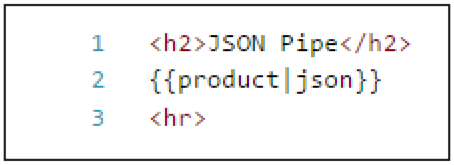
In the component, there is a **product** property as shown in listing 30.11:

Listing 30.11: *Product Object declaration*

```
this.product = new Product(1001, 'Laptop', 560000);
```

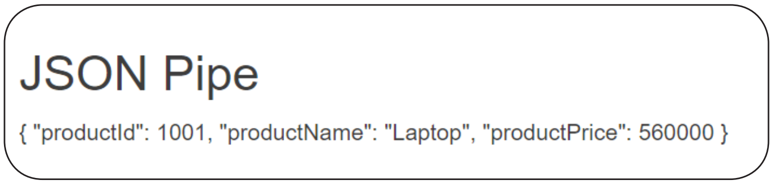
This property is used in an expression on the View as shown in Listing 30.12:

Listing 30.12: *Using json pipe*



```
<h2>JSON Pipe</h2>
{{product|json}}
<hr>
```

In the output, the product will be JSON formatted as shown in the following image:



JSON Pipe

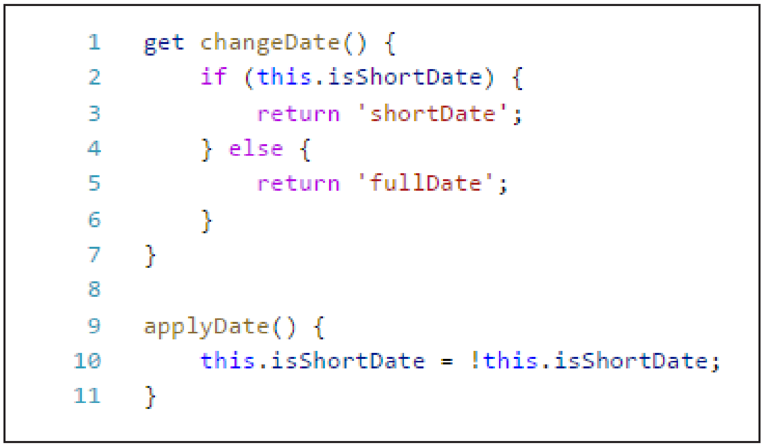
{ "productId": 1001, "productName": "Laptop", "productPrice": 560000 }

Figure 30.7: *The output of the json pipe*

Dynamically changing pipe

We can dynamically change a pipe applied on a property bound to the UI. In the component class, declare an **isShortDate** Boolean property and a **joiningDate** string property. For the joiningDate, we can now toggle through the format value for the date pipe.

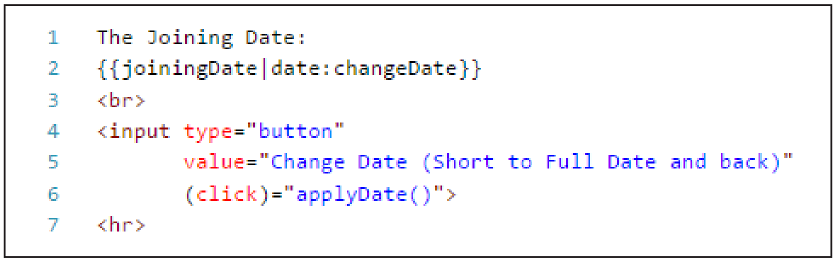
Listing 30.13: *Code to change and apply date*



```
get changeDate() {
    if (this.isShortDate) {
        return 'shortDate';
    } else {
        return 'fullDate';
    }
}

applyDate() {
    this.isShortDate = !this.isShortDate;
}
```

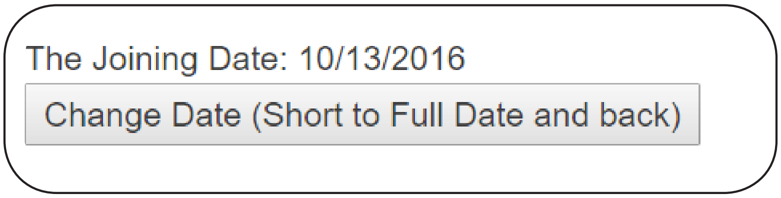
Use the applyDate function to toggle the format value of the date filter, using **changeDate** property parameter on the date pipe. See Listing 30.14:

Listing 30.14: *Using dynamically changing pipe*



```
The Joining Date:
{{joiningDate|date:changeDate}}
<br>
<input type="button"
       value="Change Date (Short to Full Date and back)"
       (click)="applyDate()">
<hr>
```

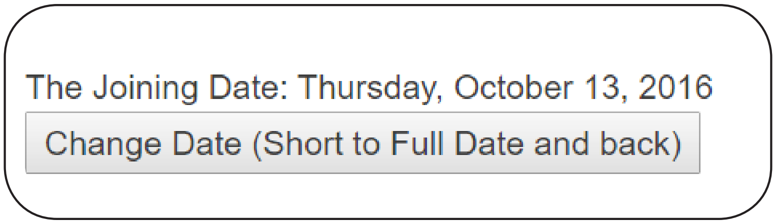
View the page in the browser, the default result will be as follows:



The Joining Date: 10/13/2016

Figure 30.8: *Dynamically changing pipe*

By default, it outputs the short date. Click on the Change Date button, and as a result of the dynamically changing pipe, the output will change to the following:



The Joining Date: Thursday, October 13, 2016

Figure 30.9: *Output of the dynamically changing pipe*

This output as seen in Figure 30.9 now displays the full date.

Conclusion

Pipes in Angular 2 is a powerful feature to define data databinding more effectively. Pipes allow us to filter the outcome of our expression on the View by applying transformation to display data in better ways.

Chapter 31
Angular 2 Custom Pipes

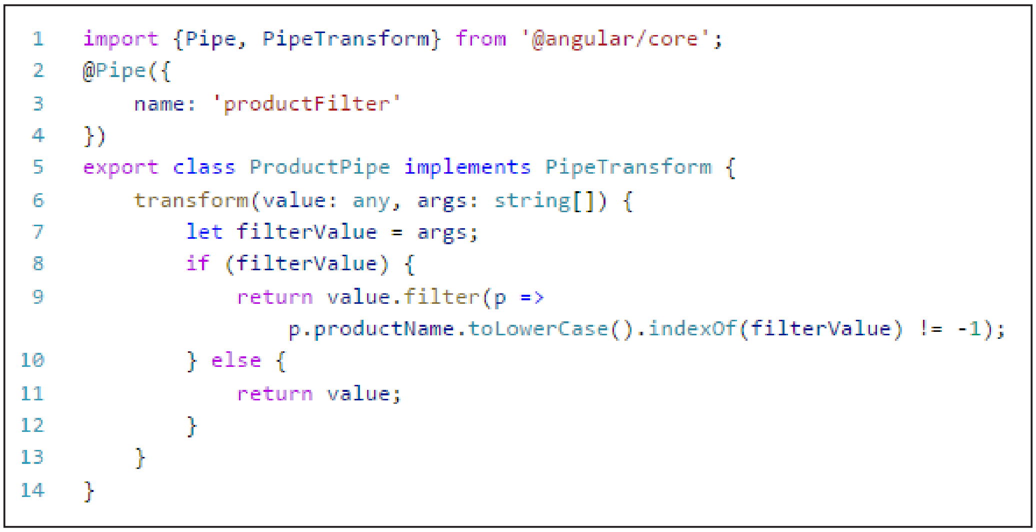
IN CHAPTER 30, WE examined built-in pipes in Angular 2. Pipes offer a powerful mechanism to transform data, leading to a better UI experience. In Angular 2, we can even define custom pipes to provide custom transformation.

In this chapter, we will implement a custom pipe to filter data from an array bound to the view. The **@Pipe** metadata decorator is used to create a custom pipe and the name of the custom pipe is passed to this decorator. The pipe class implements the **PipeTransform** interface, which provides the **transform** method. This method must be implemented to write logic for the custom pipe.

The transform method accepts the value to be processed as an input parameter. The value of the parameter is used for implementing logic of the custom pipe. The **transform** method can be implemented *without* implementing the **PipeTransform** interface in custom pipe. But implementing this interface gives tooling support for the signature of the transform method, which can be quite useful in large projects.

We will use the code from the previous chapter for implementing a custom pipe. In the project, add a new file named custom.pipe.ts. In this file, add the following code shown in listing 31.1:

Listing 31.1: *The code for custom filter*

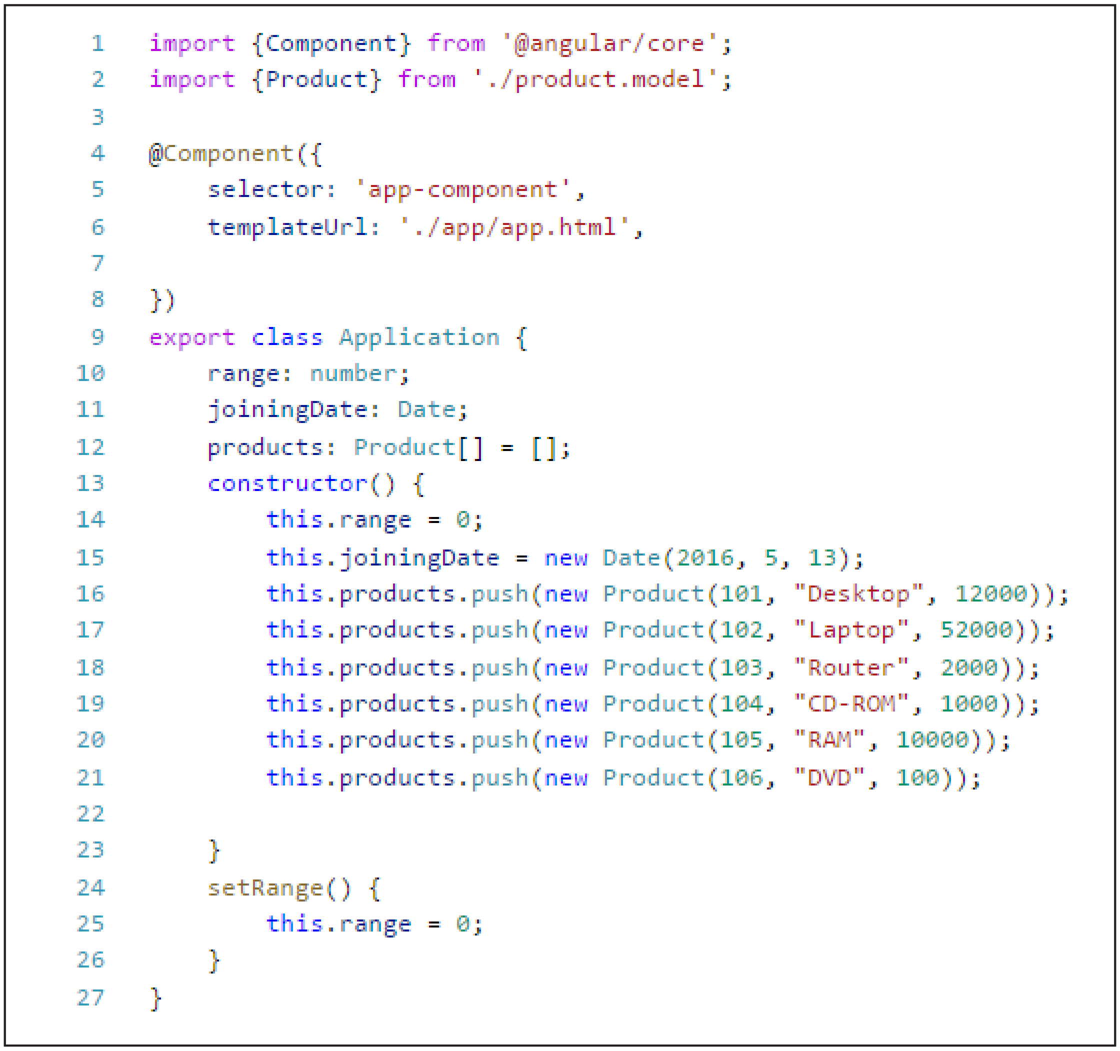


```
import {Pipe, PipeTransform} from '@angular/core';
@Pipe({
    name: 'productFilter'
})
export class ProductPipe implements PipeTransform {
    transform(value: any, args: string[]) {
        let filterValue = args;
        if (filterValue) {
            return value.filter(p =>
                p.productName.toLowerCase().indexOf(filterValue) != -1);
        } else {
            return value;
        }
    }
}
```

This code uses the Pipe decorator **productFilter.** The ProductPipe class implements PipeTransform interface to act as a pipe. As mentioned, implementing the PipeTransform interface is optional, but it is good to implement it as VS Code provides tooling support for the method *transform*. This pipe class implements the transform method, which accepts the **value** of type **any** and **args** of type string array. The code reads the ProductName property value matching with the **value** parameter, and returns all matching occurrences from the string array.

Add a new file applicationcomponents.ts in the app folder with the following code:

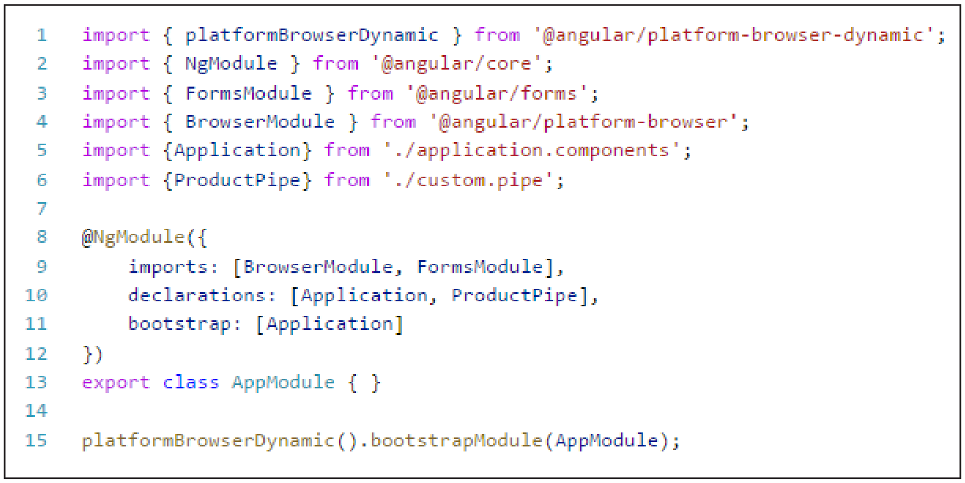
Listing 31.2: *The component code*



```
import {Component} from '@angular/core';
import {Product} from './product.model';

@Component({
    selector: 'app-component',
    templateUrl: './app/app.html',

})
export class Application {
    range: number;
    joiningDate: Date;
    products: Product[] = [];
    constructor() {
        this.range = 0;
        this.joiningDate = new Date(2016, 5, 13);
        this.products.push(new Product(101, "Desktop", 12000));
        this.products.push(new Product(102, "Laptop", 52000));
        this.products.push(new Product(103, "Router", 2000));
        this.products.push(new Product(104, "CD-ROM", 1000));
        this.products.push(new Product(105, "RAM", 10000));
        this.products.push(new Product(106, "DVD", 100));

    }
    setRange() {
        this.range = 0;
    }
}
```

As shown in listing 31.2, the Component class defines a products array. To be use the custom pipe in the current application, import it in the application module. In the app folder, add the main.ts file with the following code:

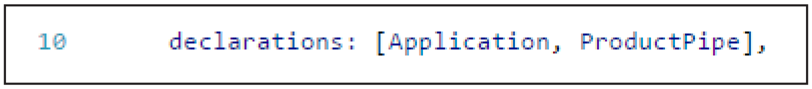
Listing 31.3: *Importing required dependencies and custom pipe*



```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';
import {Application} from './application.components';
import {ProductPipe} from './custom.pipe';

@NgModule({
    imports: [BrowserModule, FormsModule],
    declarations: [Application, ProductPipe],
    bootstrap: [Application]
})
export class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

The code shown above imports the custom.pipe file and uses it in the **declaration** of the NgModule using the following statement:

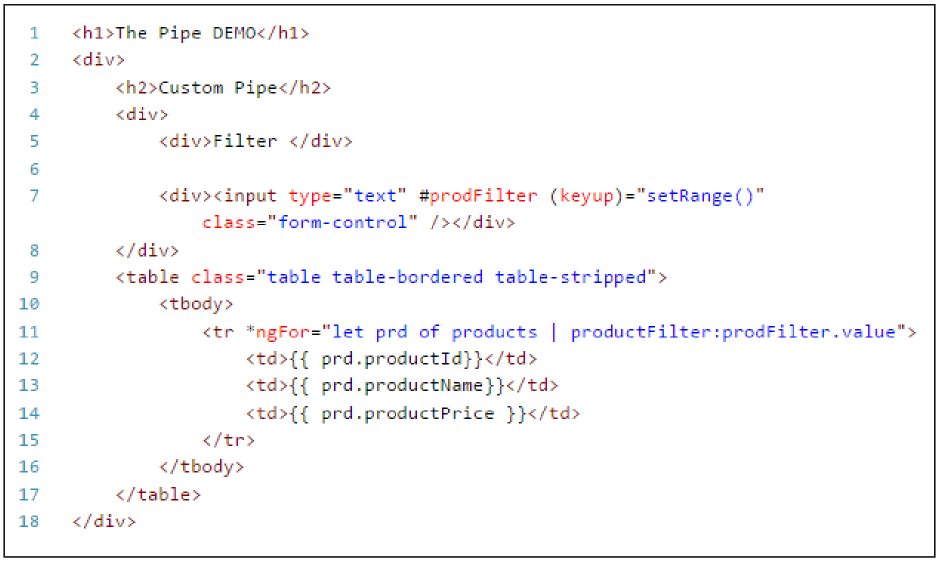
Listing 31.4: *Declaring custom pipe in NgModule*



```
    declarations: [Application, ProductPipe],
```

Add a new html file in the app folder of name app.html with the following HTML markup:

Listing 31.5: *Using custom pipe*



```
<h1>The Pipe DEMO</h1>
<div>
    <h2>Custom Pipe</h2>
    <div>
        <div>Filter </div>

        <div><input type="text" #prodFilter (keyup)="setRange()"
            class="form-control" /></div>
    </div>
    <table class="table table-bordered table-stripped">
        <tbody>
            <tr *ngFor="let prd of products | productFilter:prodFilter.value">
                <td>{{ prd.productId}}</td>
                <td>{{ prd.productName}}</td>
                <td>{{ prd.productPrice }}</td>
            </tr>
        </tbody>
    </table>
</div>
```

Listing 31.5 demonstrates how to use the custom pipe prodFilter on the table row, which is generating rows based on the products array. This pipe accepts a parameter as a value entered into the input text element, with the variable as #prodFilter. The value in the input text element is read when the keyup event is fired.

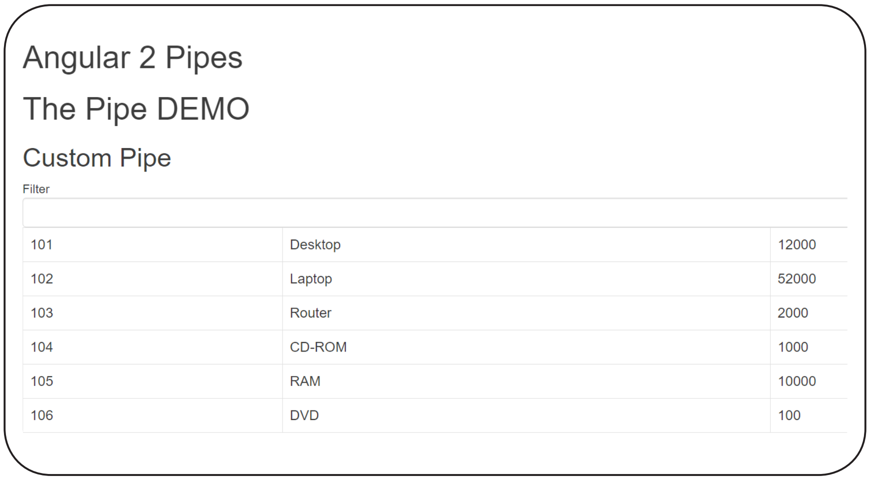
Run the application from the command prompt using the following command:

```
npm run start
```

Enter the following URL in the browser:

http://localhost:3000

Figure 31.1 shows the following output:



Angular 2 Pipes

The Pipe DEMO

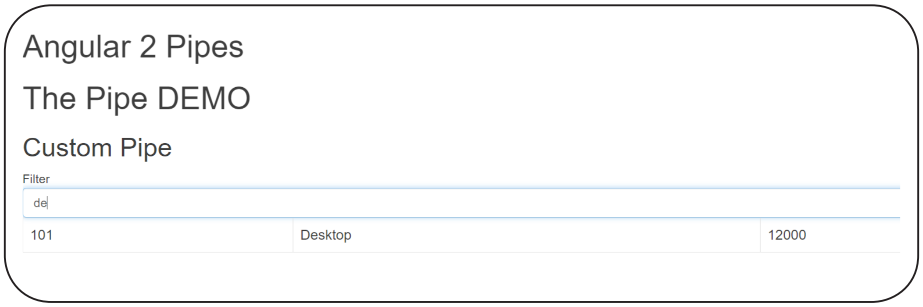
Custom Pipe

Filter

101	Desktop	12000
102	Laptop	52000
103	Router	2000
104	CD-ROM	1000
105	RAM	10000
106	DVD	100

Figure 31.1: *The first output*

Enter some data in the textbox, for example: **de**, and the following result will be displayed:



Angular 2 Pipes

The Pipe DEMO

Custom Pipe

Filter

de

101	Desktop	12000

Figure 31.2: *Output after execution of the custom pipe.*

Conclusion

Custom pipes allow developers to implement their own logic to manage the output effectively, and as per the business requirements.

Chapter

32

Using Custom Components in Angular 2 Components and Services

CHAPTER 31 EXPLAINED HOW to create and use custom pipes in Angular 2. With the help of pipes, data can be presented in an effective manner to the end-user. Although pipes can be used in a user interface for presenting data for databinding, they can also be used in components and services explicitly. The advantage is that the application can reuse the functionality of a pipe at various places to avoid the code repetition.

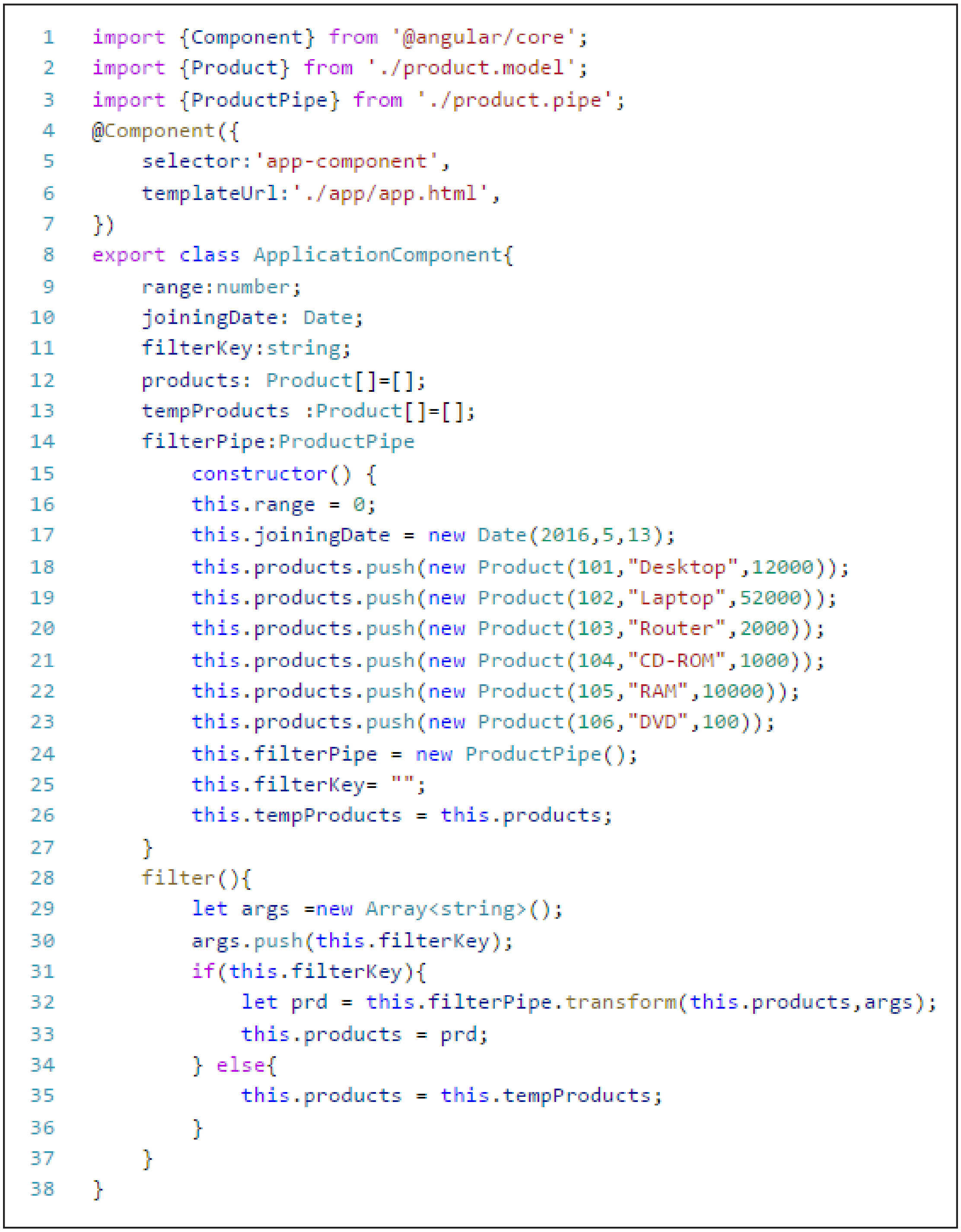
This chapter uses the custom ProductPipe created in Chapter 31. This pipe provides data filtration based on the name of the product. The custom pipe implements the *PipeTransform* interface its transform() method. This method contains logic for operations to be performed by the custom pipe.

NOTE: While using custom pipe explicitly in Angular 2 components and services, the transform method must be called explicitly.

Using a Custom Pipe in Component

Open the code from \Chapter 31 in Visual Studio Code and review the code for the ProductPipe. Since the custom pipe is used for filtering products, rename the file name custom.pipe.ts to product.pipe.ts. Since the *ApplicationComponent* is going to use the custom pipe, the code must be modified in the application.component.ts as shown in listing 32.1:

Listing 32.1: *The ApplicationComponent code*

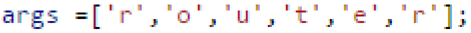


```
import {Component} from '@angular/core';
import {Product} from './product.model';
import {ProductPipe} from './product.pipe';
@Component({
    selector:'app-component',
    templateUrl:'./app/app.html',
})
export class ApplicationComponent{
    range:number;
    joiningDate: Date;
    filterKey:string;
    products: Product[]=[];
    tempProducts :Product[]=[];
    filterPipe:ProductPipe
        constructor() {
        this.range = 0;
        this.joiningDate = new Date(2016,5,13);
        this.products.push(new Product(101,"Desktop",12000));
        this.products.push(new Product(102,"Laptop",52000));
        this.products.push(new Product(103,"Router",2000));
        this.products.push(new Product(104,"CD-ROM",1000));
        this.products.push(new Product(105,"RAM",10000));
        this.products.push(new Product(106,"DVD",100));
        this.filterPipe = new ProductPipe();
        this.filterKey= "";
        this.tempProducts = this.products;
    }
    filter(){
        let args =new Array<string>();
        args.push(this.filterKey);
        if(this.filterKey){
            let prd = this.filterPipe.transform(this.products,args);
            this.products = prd;
        } else{
            this.products = this.tempProducts;
        }
    }
}
```

The code imports product.pipe to use the ProductPipe class. In the component class, the property filterKey is declared as a string. This property will be used to push the ProductName through an array to the *transform* method of the ProductPipe class. The filterPipe property is declared of the type ProductPipe, which will be used to access

ProductPipe methods explicitly in the component. tempProducts is a Product array used to store Products information.

In Listing 32.1, the filter() method declares a string array of the name *args*. This array contains each character of the ProductName entered using the filterKey property, e.g. if the ProductName is 'router', then the args will be as following:

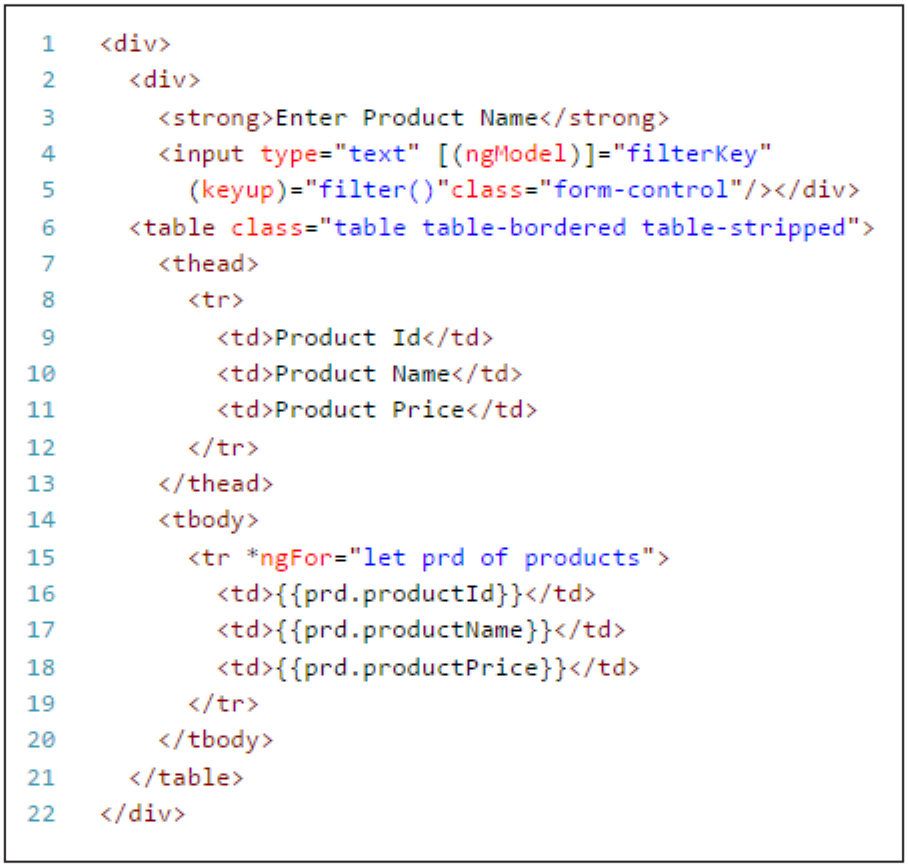


```
args =['r','o','u','t','e','r'];
```

This array will be passed as an input parameter to the transform method of the ProductPipe, along with the products array parameter. The transform method will be executed if the filterKey has a value, and will return an array of filtered products based on the matching character from the args array; otherwise, the products array will contain data from the tempProducts array.

Since a custom pipe is used in the component, the app.html must use it in a table. The app.html needs to be modified as shown in listing 32.2:

Listing 32.2: *The app.html*



```
<div>
  <div>
    <strong>Enter Product Name</strong>
    <input type="text" [(ngModel)]="filterKey"
      (keyup)="filter()"class="form-control"/></div>
  <table class="table table-bordered table-stripped">
    <thead>
      <tr>
        <td>Product Id</td>
        <td>Product Name</td>
        <td>Product Price</td>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let prd of products">
        <td>{{prd.productId}}</td>
        <td>{{prd.productName}}</td>
        <td>{{prd.productPrice}}</td>
      </tr>
    </tbody>
  </table>
</div>
```

The <input> element is bound with the filterKey property of the ApplicationComponent. The filter() method of the component is bound to the keyup event of the <input> element using event binding.

Since the code uses the data-binding features, the application must use FormsModule. To use the FormsModule, modify package.json for installing @angular/forms dependencies. The systemjs.config.js file must be modified to use forms module as shown in Chapter 17.

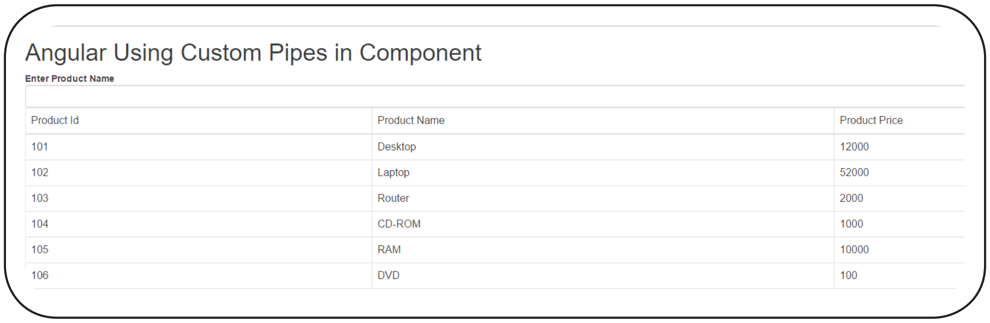
To run the application, right-click on index.html and select Open in Command Prompt. Enter the following command in the command prompt:

```
> npm run start
```

To view the result, enter the following url in the browser:

http://localhost:3000

The result will be displayed as follows:



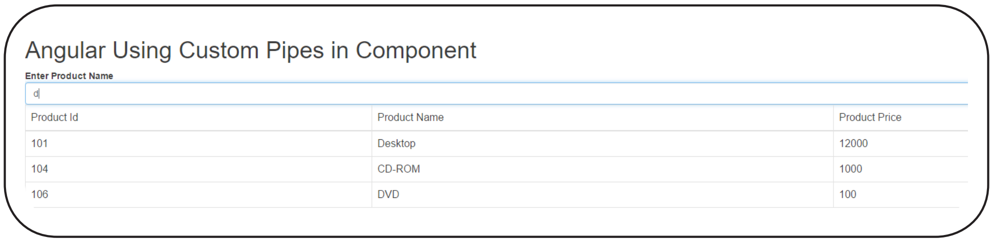
Angular Using Custom Pipes in Component

Enter Product Name

Product Id	Product Name	Product Price
101	Desktop	12000
102	Laptop	52000
103	Router	2000
104	CD-ROM	1000
105	RAM	10000
106	DVD	100

Figure 32.1: *List of products*

Enter the ProductName in the Text box with the label 'Enter Product Name'. For instance, enter 'd' in it and the Product will be filtered as shown in Figure 32.2.



Angular Using Custom Pipes in Component

Enter Product Name

d

Product Id	Product Name	Product Price
101	Desktop	12000
104	CD-ROM	1000
106	DVD	100

Figure 32.2: *List of Products filtered based on value in TextBox*

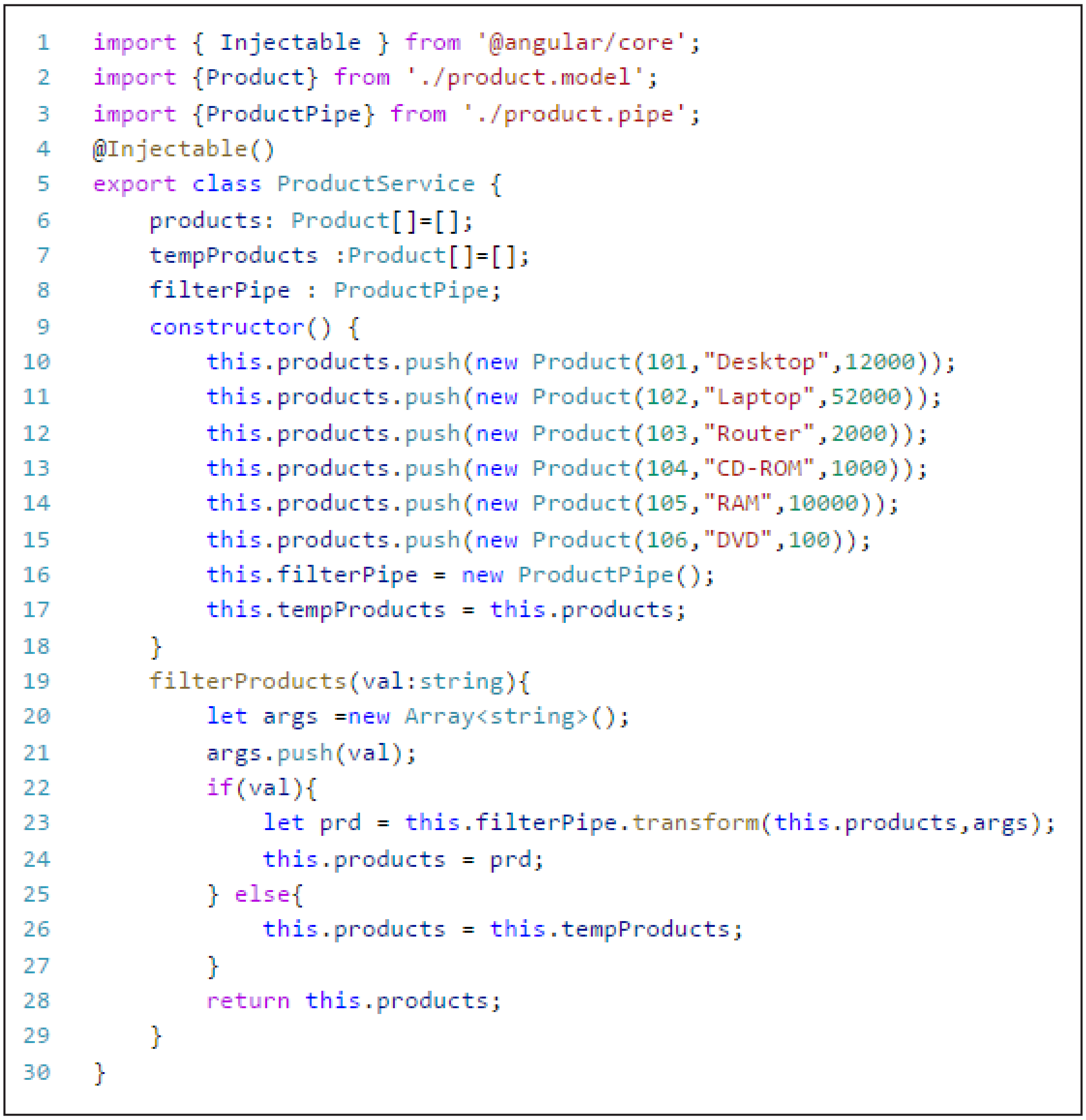
When the input is entered in the textbox, the keyup event is fired, calling the filter() method of the component. The method filters products with names, starting with the string entered in the textbox.

Using Custom Pipe in Service

To reiterate, a custom pipe may also be used in a Service. Chapters 25 and 27 explain the use and importance of Angular 2 Services.

In the app folder, add a new file named productservice.ts. This file will contain the products array and a method filterProducts() to filter the products by using the custom pipe. The code for this service is shown in listing 32.3.

Listing 32.3: *Product Service code*

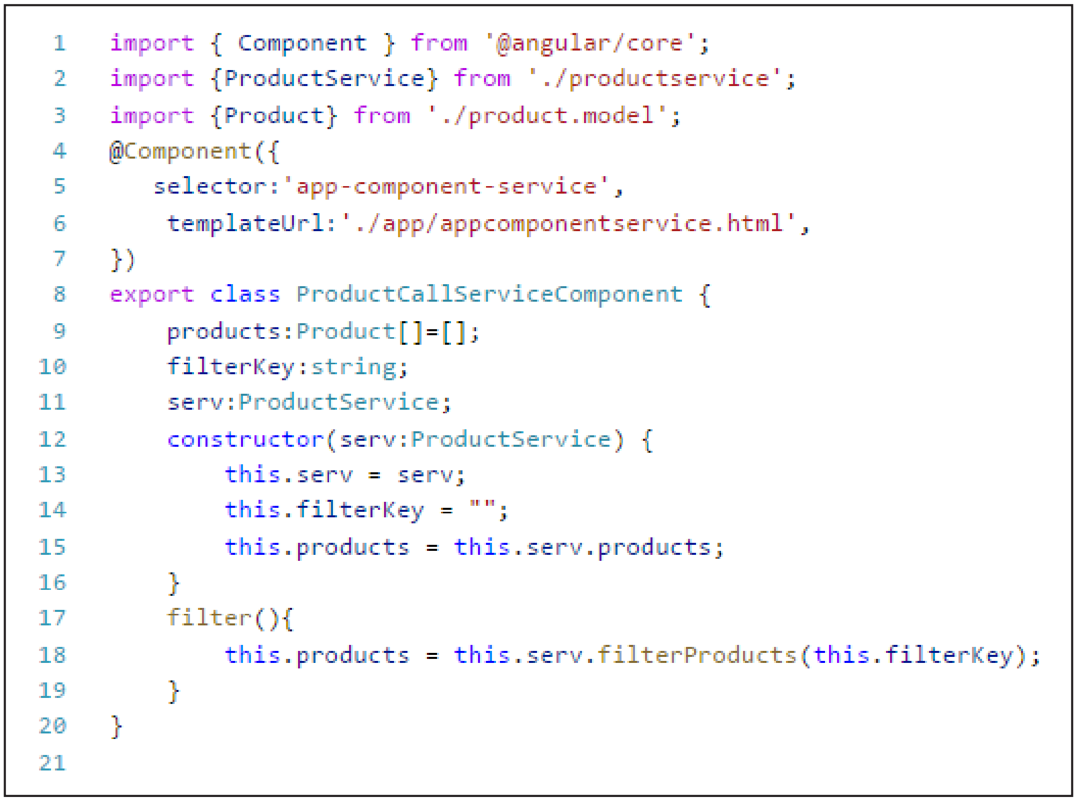


```
import { Injectable } from '@angular/core';
import {Product} from './product.model';
import {ProductPipe} from './product.pipe';
@Injectable()
export class ProductService {
    products: Product[]=[];
    tempProducts :Product[]=[];
    filterPipe : ProductPipe;
    constructor() {
        this.products.push(new Product(101,"Desktop",12000));
        this.products.push(new Product(102,"Laptop",52000));
        this.products.push(new Product(103,"Router",2000));
        this.products.push(new Product(104,"CD-ROM",1000));
        this.products.push(new Product(105,"RAM",10000));
        this.products.push(new Product(106,"DVD",100));
        this.filterPipe = new ProductPipe();
        this.tempProducts = this.products;
    }
    filterProducts(val:string){
        let args =new Array<string>();
        args.push(val);
        if(val){
            let prd = this.filterPipe.transform(this.products,args);
            this.products = prd;
        } else{
            this.products = this.tempProducts;
        }
        return this.products;
    }
}
```

The code in the service is very similar to the code of the Listing 31-1. The filterProducts() method calls the transform() method of the ProductPipe to filter products based on the ProductName, and returns the products array after finding the matching products.

The service must be called from a component. In the app folder, add a new file named app.callservice.component.ts with the code shown in listing 32.4.

Listing 32.4: *The component calling service*



```
import { Component } from '@angular/core';
import {ProductService} from './productservice';
import {Product} from './product.model';
@Component({
   selector:'app-component-service',
    templateUrl:'./app/appcomponentservice.html',
})
export class ProductCallServiceComponent {
    products:Product[]=[];
    filterKey:string;
    serv:ProductService;
    constructor(serv:ProductService) {
        this.serv = serv;
        this.filterKey = "";
        this.products = this.serv.products;
    }
    filter(){
        this.products = this.serv.filterProducts(this.filterKey);
    }
}

```

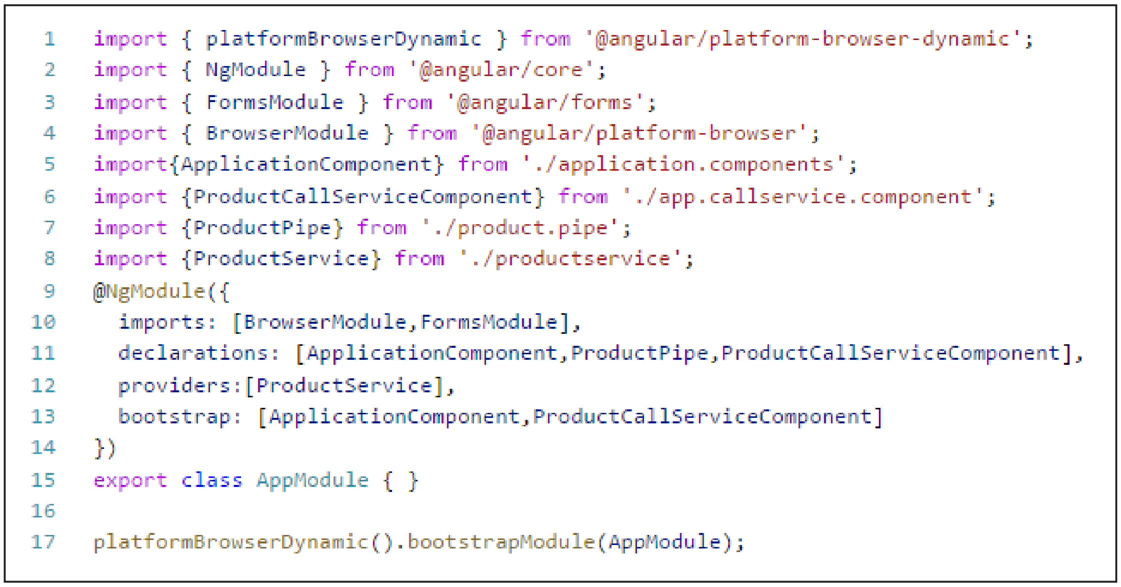
This code imports the ProductService and injects it in the constructor. The constructor calls the products array property from the service and stores all products from the service in the products property, declared in the component.

The filter method of the component class calls the filterProducts method of the service by passing the filterKey property value to it. This filterKey passes the ProductName to the filterProducts method, which will further filter products by matching ProductName to the filterKey and returns products.

In the app folder, add a new file named appcomponentservice.html with the same markup as the app.html file in listing 32-2. This html file is the template for the app-component-service selector declared in the ProductCallServiceComponent.

Since the application has two components, main.ts must be modified to load both components as shown in listing 32.5:

Listing 32.5: *main.ts with multiple bootstrap components*

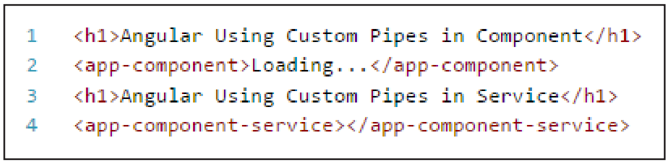


```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';
import{ApplicationComponent} from './application.components';
import {ProductCallServiceComponent} from './app.callservice.component';
import {ProductPipe} from './product.pipe';
import {ProductService} from './productservice';
@NgModule({
  imports: [BrowserModule,FormsModule],
  declarations: [ApplicationComponent,ProductPipe,ProductCallServiceComponent],
  providers:[ProductService],
  bootstrap: [ApplicationComponent,ProductCallServiceComponent]
})
export class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Listing 32.5 loads the components ApplicationComponent and ProductCallServiceComponents when the Angular 2 application is bootstrapped using the module AppModule.

The code in the <body> tag of the index.html which will use the multiple component selector is as shown in listing 32.5.

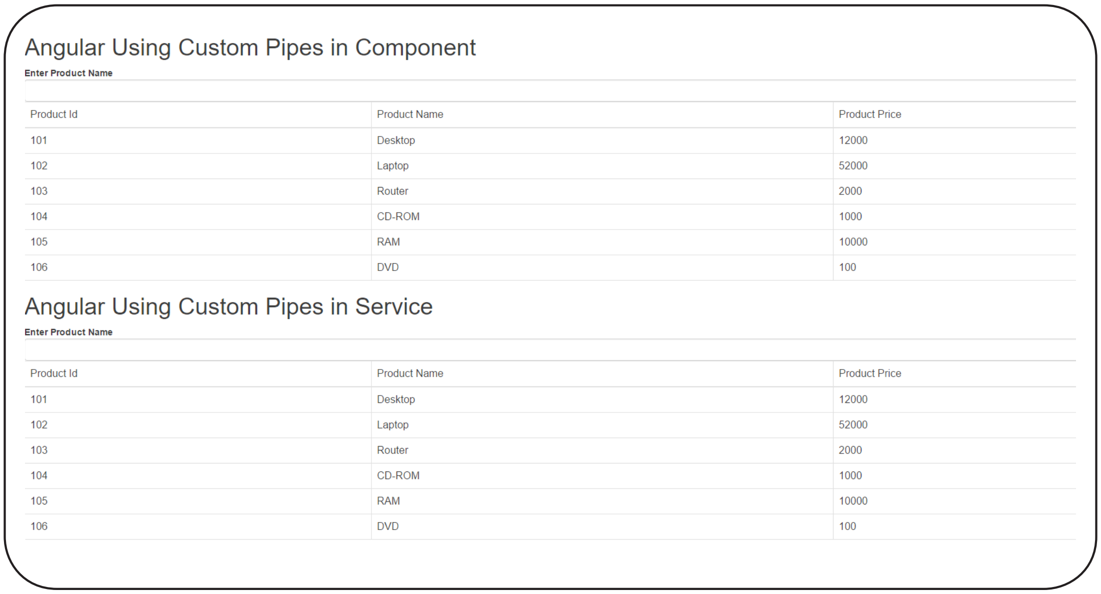
Listing 32.5: *index.htm*



```
<h1>Angular Using Custom Pipes in Component</h1>
<app-component>Loading...</app-component>
<h1>Angular Using Custom Pipes in Service</h1>
<app-component-service></app-component-service>
```

This piece of code will load html files assigned for the **templateUrl** property of the component.

Run the application in the browser. The result will be shown as in Figure 32.3.



Angular Using Custom Pipes in Component

Enter Product Name

Product Id	Product Name	Product Price
101	Desktop	12000
102	Laptop	52000
103	Router	2000
104	CD-ROM	1000
105	RAM	10000
106	DVD	100

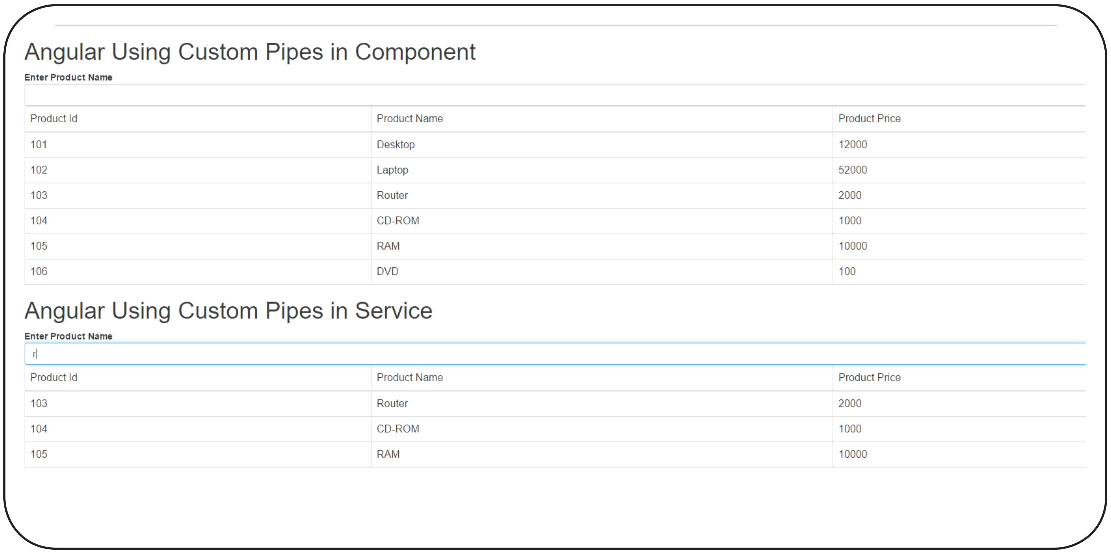
Angular Using Custom Pipes in Service

Enter Product Name

Product Id	Product Name	Product Price
101	Desktop	12000
102	Laptop	52000
103	Router	2000
104	CD-ROM	1000
105	RAM	10000
106	DVD	100

Figure 32.3: *The Use of all components*

Enter the ProductName in the Textbox labeled with 'Enter Product Name' under the header 'Angular Using Custom Pipes in Service'. If the textbox contains character 'r', then all products starting from 'r' will be shown:



Angular Using Custom Pipes in Component

Enter Product Name

Product Id	Product Name	Product Price
101	Desktop	12000
102	Laptop	52000
103	Router	2000
104	CD-ROM	1000
105	RAM	10000
106	DVD	100

Angular Using Custom Pipes in Service

Enter Product Name

r

Product Id	Product Name	Product Price
103	Router	2000
104	CD-ROM	1000
105	RAM	10000

Figure 32.4: *The custom pipe in service*

Conclusion

The use of custom pipes in components and services allows developer to reuse their own logic effectively.

Chapter

33

Understanding Routing

A SINGLE PAGE APPLICATION (SPA) requires just one page load, and thereafter all required elements can be loaded dynamically onto the page without the need to leave the page. SPA consists of many functionalities with limited or light-weight access to the server. All of the tasks are performed in a single page, without asking the server for a new page to render.

Sometimes presenting all of the functionalites on the same view without a change in the url can be confusing and may reduce the usability of the application. Client-side routing addresses this problem by providing a way to navigate between different parts of the application. With client-side routing, one can switch between different views without asking the server to refresh the whole page. The URL of the page changes when a user switches between the views. This URL can be bookmarked and one can return to the client-side rendered view directly, without having to navigate to this page again. When the context switches from one view to the other, an SPA sends requests for the template and related files of the new view through AJAX requests.

This chapter will explain how Angular 2 supports routing using a basic example. By the end of this chapter, you will be able to configure simple routes, creates links to the routes, and use them.

To follow along with this chapter, you can start with the development environment created in chapter 5. To this, install the @angular/forms package using the following command:

```
> npm install @angular/forms@2.0.0 –save
```

It also needs a couple of packages for the server. The following command installs them:

```
> npm install co-body koa-route --save
```

The form package and the http and router packages have to be registered with SystemJS to be able to refer it in the application. Add the following entries to the *map* configuration in the file *systemjs.config.js:*

```
"@angular/http": "node_modules/@angular/http",
```

"@angular/forms": "node_modules/@angular/forms",

"@angular/router": "node_modules/@angular/router"

And the following entries to the *packages* section in the file *systemjs.config.js*:

"@angular/http": { "main": "bundles/http.umd.js", "defaultExtension": "js" },

"@angular/forms": { "main": "bundles/forms.umd.js", "defaultExtension": "js" },

"@angular/router": { "main": "bundles/router.umd.js", "defaultExtension": "js" }

Copy the files server.js and books.json from the downloadable code of this chapter and paste them in the folder where you have set up your project.

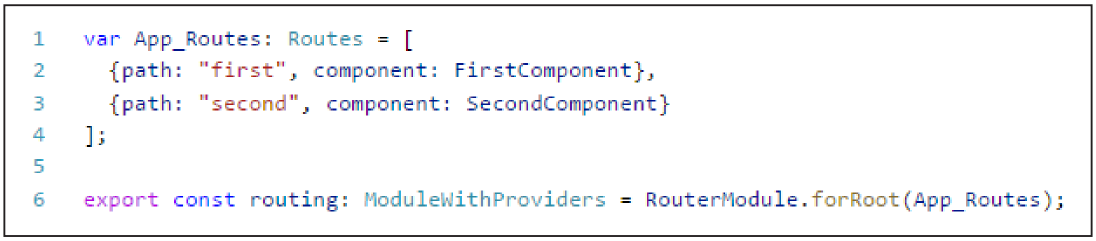
How to use Routes in Angular 2

To use routing in Angular 2, the routing module must be installed using npm. The following command installs this package:

```
> npm install @angular/router
```

The package is already installed in the setup project we created in Chapter 5. The set of routes of the application has to be added to a list. This list must be passed to *RouteModule*. Listing 33.1 shows the syntax of creating the routes:

Listing 33.1: *Configuring routes in Angular 2*

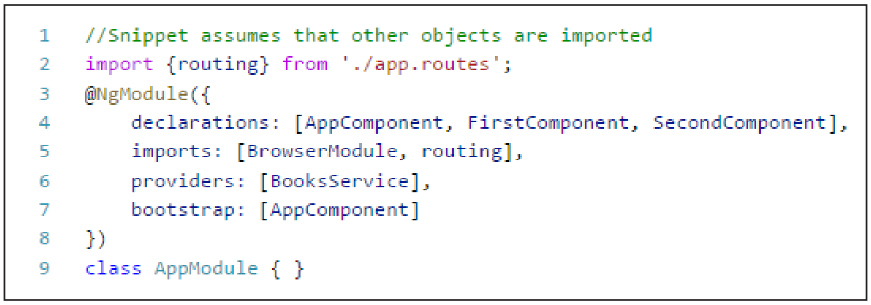


```
var App_Routes: Routes = [
  {path: "first", component: FirstComponent},
  {path: "second", component: SecondComponent}
];

export const routing: ModuleWithProviders = RouterModule.forRoot(App_Routes);
```

Notice that the *routing* variable is made constant in the above snippet. It is done to prevent any modifications to the routes.

As shown here, every entry in the routes array is an object with two properties containing relative paths to reach the route as well as the component to be rendered in the route. The last statement in listing 33.1 creates a module from the list of routes. The application module has to import this module. Upon importing this module, the application gets access to all exported members of the RouteModule, and the routes defined in the above snippet are configured for the application. Listing 33.2 imports the routing module inside application module:

Listing 33.2: *Importing routing module*



```
//Snippet assumes that other objects are imported
import {routing} from './app.routes';
@NgModule({
    declarations: [AppComponent, FirstComponent, SecondComponent],
    imports: [BrowserModule, routing],
    providers: [BooksService],
    bootstrap: [AppComponent]
})
class AppModule { }
```

As an Angular 2 application starts with a component, the first component must have a placeholder in its template. The placeholder is a *router-outlet* component; this component is defined in the router module and is available after importing the routing module.

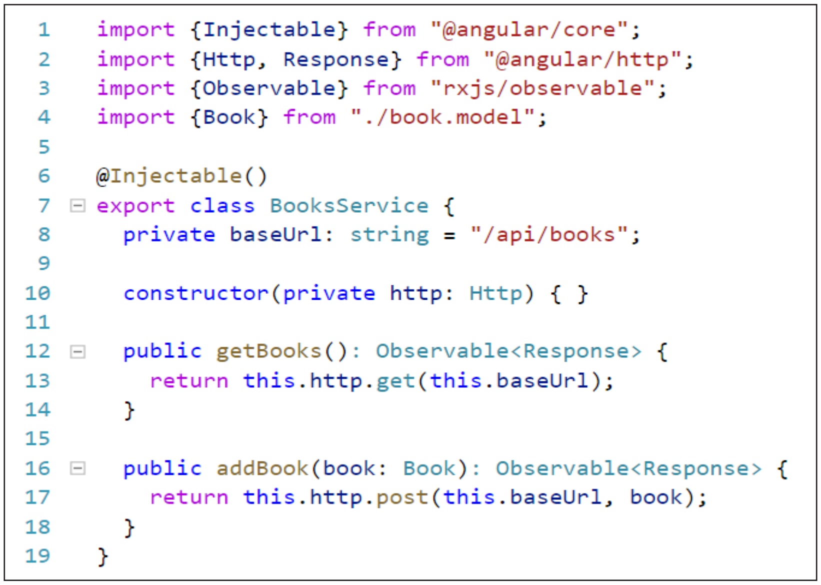
These are some basic steps to get started with routing in Angular 2. Let's build an app to use these features and to explore several others.

Building an App Using Routing

BUILDING COMPONENTS AND SERVICE

Looking at the example of a book shelf, the application consists of two routes: one to view the list of books and another to add a new book. The application interacts with a REST API built using Koa.js. Refer to the sample code of this chapter to view the APIs. An angular 2 service should interact with this API. Listing 33.3 shows code of the class *BooksService,* which interacts with the APIs:

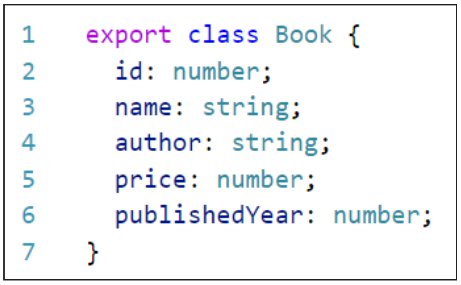
Listing 33.3: *Code of BooksService, to be placed in the file*



```
import {Injectable} from "@angular/core";
import {Http, Response} from "@angular/http";
import {Observable} from "rxjs/observable";
import {Book} from "./book.model";

@Injectable()
export class BooksService {
  private baseUrl: string = "/api/books";

  constructor(private http: Http) { }

  public getBooks(): Observable<Response> {
    return this.http.get(this.baseUrl);
  }

  public addBook(book: Book): Observable<Response> {
    return this.http.post(this.baseUrl, book);
  }
}
```

To represent the structure of a book in TypeScript, we need a model class for book. The following snippet shows the *Book* class, it has to be added to the file book.model.ts:

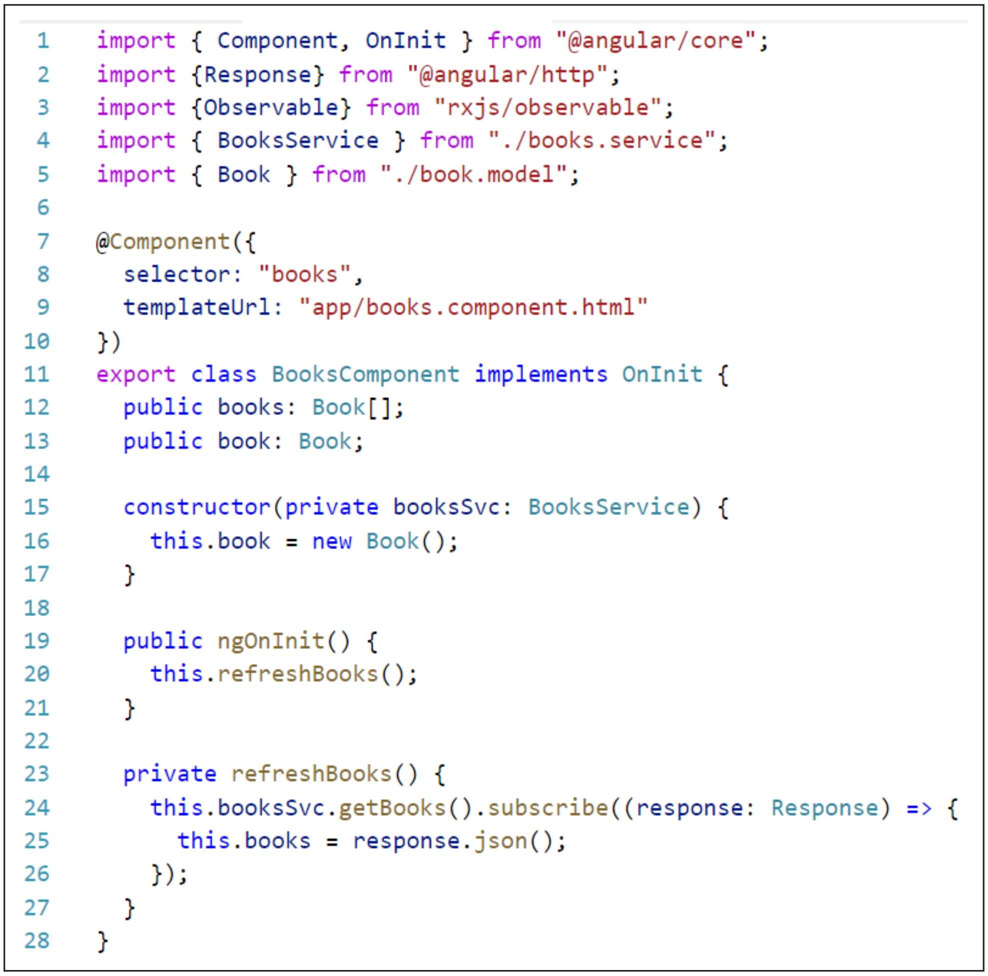
Listing 33.4: *The Book class*



```
export class Book {
  id: number;
  name: string;
  author: string;
  price: number;
  publishedYear: number;
}
```

The first page of the application has to display the list of books available. A component is required to consume the BooksService to get the data and add it to the component instance, to make it accessible to the UI. Add a file named books.component.ts. Listing 33.5 shows the TypeScript class of the component, it has to be added to the newly created file:

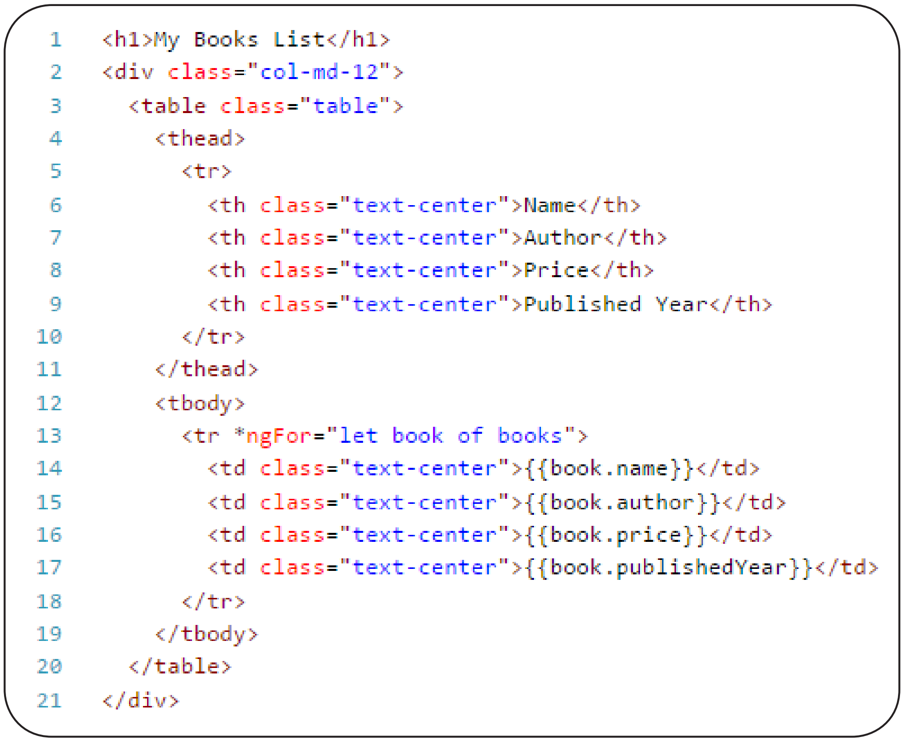
Listing 33.5: *Code of BooksComponent*



```
import { Component, OnInit } from "@angular/core";
import {Response} from "@angular/http";
import {Observable} from "rxjs/observable";
import { BooksService } from "./books.service";
import { Book } from "./book.model";

@Component({
  selector: "books",
  templateUrl: "app/books.component.html"
})
export class BooksComponent implements OnInit {
  public books: Book[];
  public book: Book;

  constructor(private booksSvc: BooksService) {
    this.book = new Book();
  }

  public ngOnInit() {
    this.refreshBooks();
  }

  private refreshBooks() {
    this.booksSvc.getBooks().subscribe((response: Response) => {
      this.books = response.json();
    });
  }
}
```

The template of this page has a table displaying a list of books. The *ngFor* structural directive is used to iterate through the list of books and display them in different rows of the table. Listing 33.5 shows the markup of the template file:

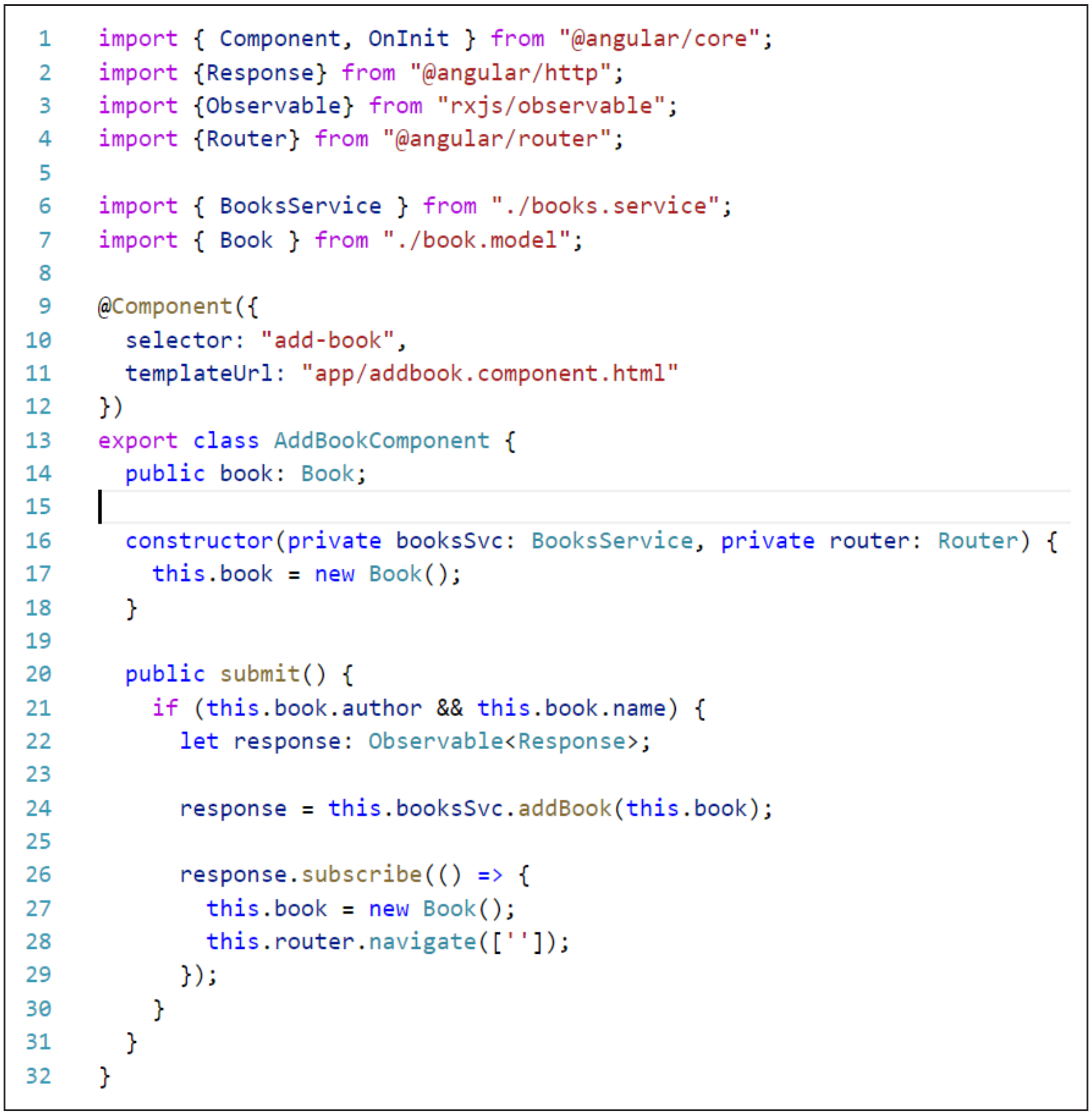
Listing 33.6: *Template of BooksComponent*



```
<h1>My Books List</h1>
<div class="col-md-12">
  <table class="table">
    <thead>
      <tr>
        <th class="text-center">Name</th>
        <th class="text-center">Author</th>
        <th class="text-center">Price</th>
        <th class="text-center">Published Year</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let book of books">
        <td class="text-center">{{book.name}}</td>
        <td class="text-center">{{book.author}}</td>
        <td class="text-center">{{book.price}}</td>
        <td class="text-center">{{book.publishedYear}}</td>
      </tr>
    </tbody>
  </table>
</div>
```

The other page in the application will add a new book. This page will have a form with four fields, accepting values for different properties of the book. It uses the *BooksService* to call the Koa.js service to add the book. Listing 33.7 shows the class of the component:

Listing 33.7: *Code of AddBookComponent*



```
import { Component, OnInit } from "@angular/core";
import {Response} from "@angular/http";
import {Observable} from "rxjs/observable";
import {Router} from "@angular/router";

import { BooksService } from "./books.service";
import { Book } from "./book.model";

@Component({
  selector: "add-book",
  templateUrl: "app/addbook.component.html"
})
export class AddBookComponent {
  public book: Book;

  constructor(private booksSvc: BooksService, private router: Router) {
    this.book = new Book();
  }

  public submit() {
    if (this.book.author && this.book.name) {
      let response: Observable<Response>;

      response = this.booksSvc.addBook(this.book);

      response.subscribe(() => {
        this.book = new Book();
        this.router.navigate(['']);
      });
    }
  }
}
```

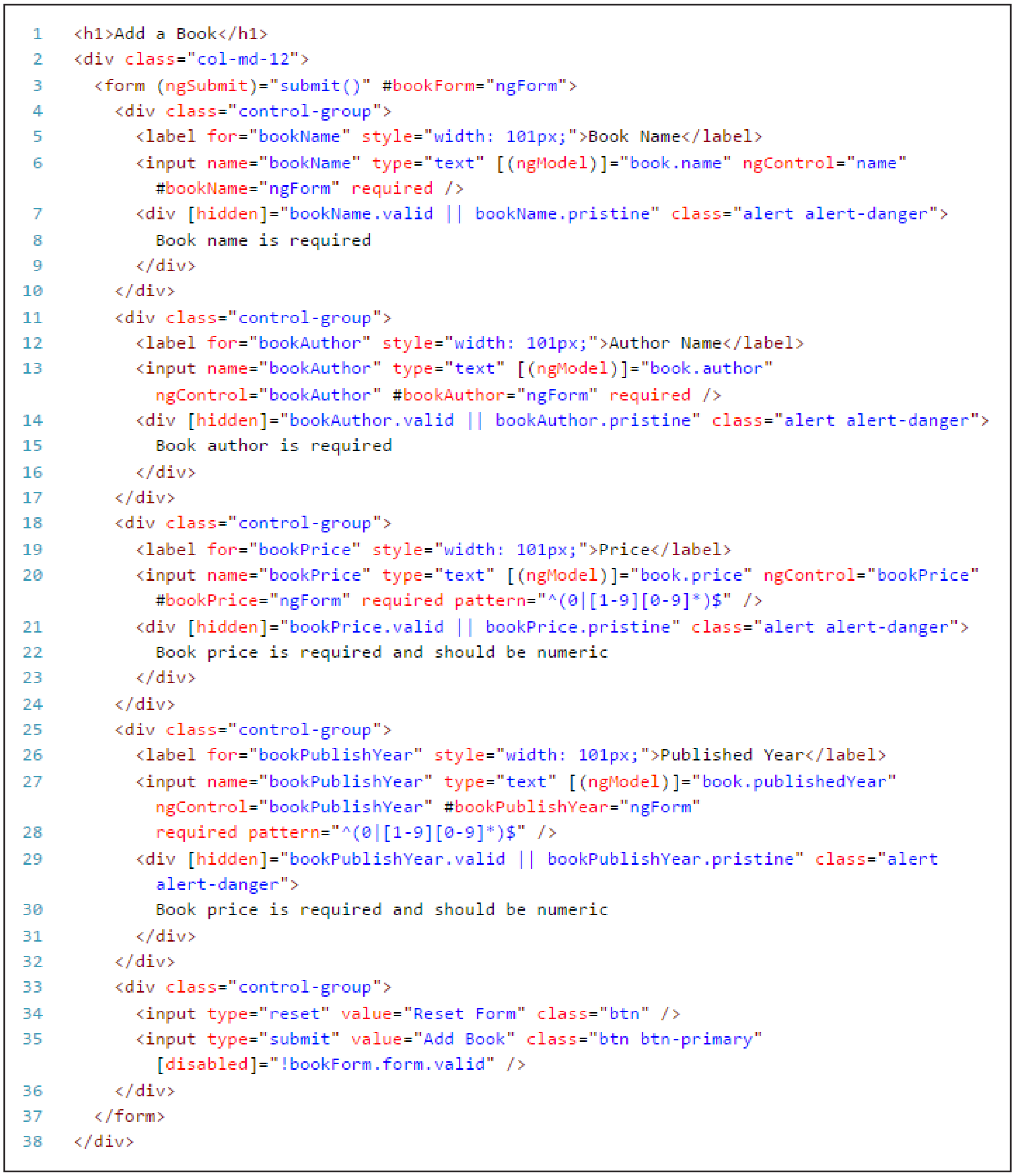
The constructor of the *AddBookComponent* accepts the *Router* service in addition to the *BooksService*. This service will be used later in the next section.

As this page has a form, the data received from the user must be validated to prevent insertion of invalid data in the service object. This component uses the template-driven forms approach for the form and their validations. Please refer to Chapter 26 to learn more about template-driven forms. The data received in this component needs the following set of validations:

- Required field validations on all the fields
- Pattern validation on price and published year fields, to check if they are numeric

Listing 33.8 shows the template of the component:

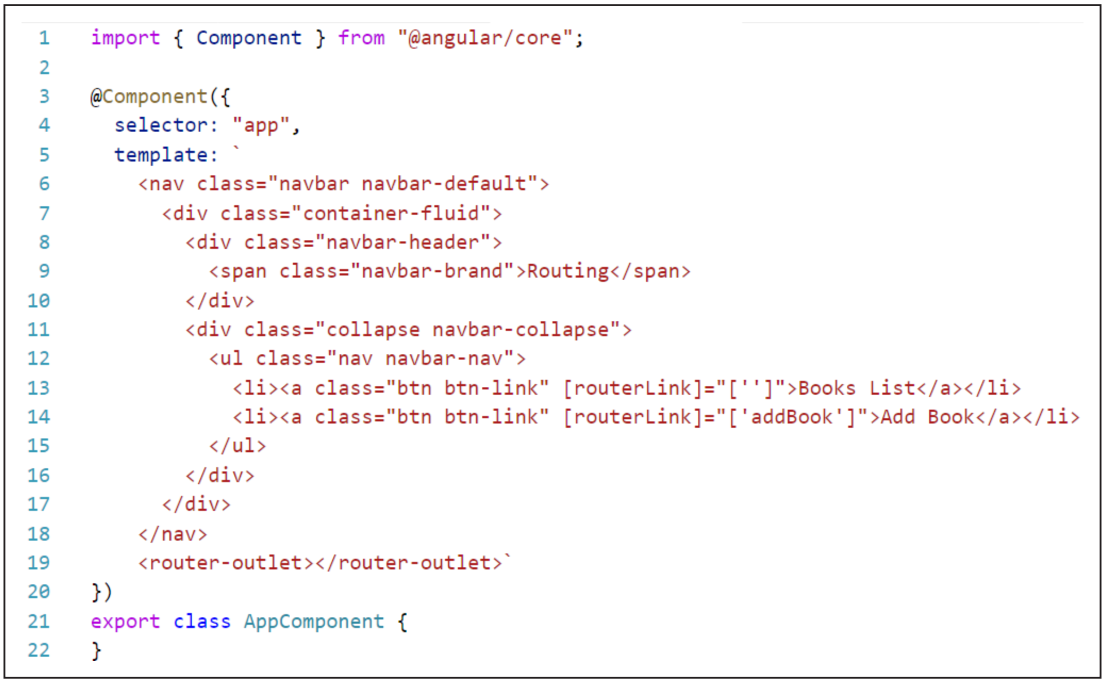
Listing 33.8: *Template of AddBookComponent*



```
<h1>Add a Book</h1>
<div class="col-md-12">
  <form (ngSubmit)="submit()" #bookForm="ngForm">
    <div class="control-group">
      <label for="bookName" style="width: 101px;">Book Name</label>
      <input name="bookName" type="text" [(ngModel)]="book.name" ngControl="name"
        #bookName="ngForm" required />
      <div [hidden]="bookName.valid || bookName.pristine" class="alert alert-danger">
        Book name is required
      </div>
    </div>
    <div class="control-group">
      <label for="bookAuthor" style="width: 101px;">Author Name</label>
      <input name="bookAuthor" type="text" [(ngModel)]="book.author"
        ngControl="bookAuthor" #bookAuthor="ngForm" required />
      <div [hidden]="bookAuthor.valid || bookAuthor.pristine" class="alert alert-danger">
        Book author is required
      </div>
    </div>
    <div class="control-group">
      <label for="bookPrice" style="width: 101px;">Price</label>
      <input name="bookPrice" type="text" [(ngModel)]="book.price" ngControl="bookPrice"
        #bookPrice="ngForm" required pattern="^(0|[1-9][0-9]*)$" />
      <div [hidden]="bookPrice.valid || bookPrice.pristine" class="alert alert-danger">
        Book price is required and should be numeric
      </div>
    </div>
    <div class="control-group">
      <label for="bookPublishYear" style="width: 101px;">Published Year</label>
      <input name="bookPublishYear" type="text" [(ngModel)]="book.publishedYear"
        ngControl="bookPublishYear" #bookPublishYear="ngForm"
        required pattern="^(0|[1-9][0-9]*)$" />
      <div [hidden]="bookPublishYear.valid || bookPublishYear.pristine" class="alert
        alert-danger">
        Book price is required and should be numeric
      </div>
    </div>
    <div class="control-group">
      <input type="reset" value="Reset Form" class="btn" />
      <input type="submit" value="Add Book" class="btn btn-primary"
        [disabled]="!bookForm.form.valid" />
    </div>
  </form>
</div>
```

The last component to add to the application is the application component, which will be used to start the application. This component will have a navigation bar and will use routing to add a placeholder for the component to be loaded in it. Listing 33.9 shows the code of the application component:

Listing 33.9: *Code of AppComponent*

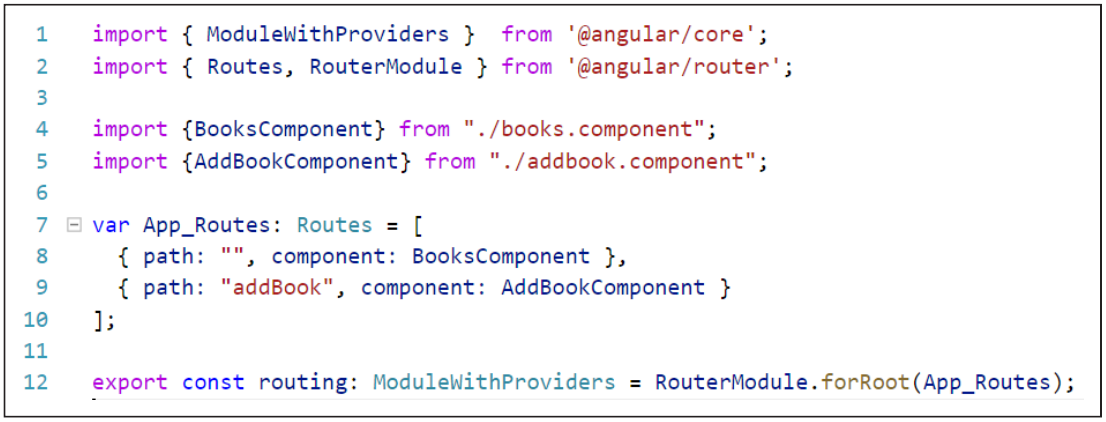


```
import { Component } from "@angular/core";

@Component({
  selector: "app",
  template: `
    <nav class="navbar navbar-default">
      <div class="container-fluid">
        <div class="navbar-header">
          <span class="navbar-brand">Routing</span>
        </div>
        <div class="collapse navbar-collapse">
          <ul class="nav navbar-nav">
            <li><a class="btn btn-link" [routerLink]="['']">Books List</a></li>
            <li><a class="btn btn-link" [routerLink]="['addBook']">Add Book</a></li>
          </ul>
        </div>
      </div>
    </nav>
    <router-outlet></router-outlet>`
})
export class AppComponent {
}
```

Adding Routing to Book Shelf

Next, add routes to the application. The demo application needs two routes for the two pages. Add a new file to the application and name it app.routes.ts. Add the code shown in listing 33.10 to this file:

Listing 33.10: *Routes*



```
import { ModuleWithProviders }  from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import {BooksComponent} from "./books.component";
import {AddBookComponent} from "./addbook.component";

var App_Routes: Routes = [
  { path: "", component: BooksComponent },
  { path: "addBook", component: AddBookComponent }
];

export const routing: ModuleWithProviders = RouterModule.forRoot(App_Routes);
```

Note that the first route added to the array *App_Routes* doesn't have a path. Angular will render the component configured in this route when no route path is specified in the URL. The module created from the routes list is exported, so that it can be registered with the application's module. An application module has to be created to start the application. Listing 33.11 creates this module:

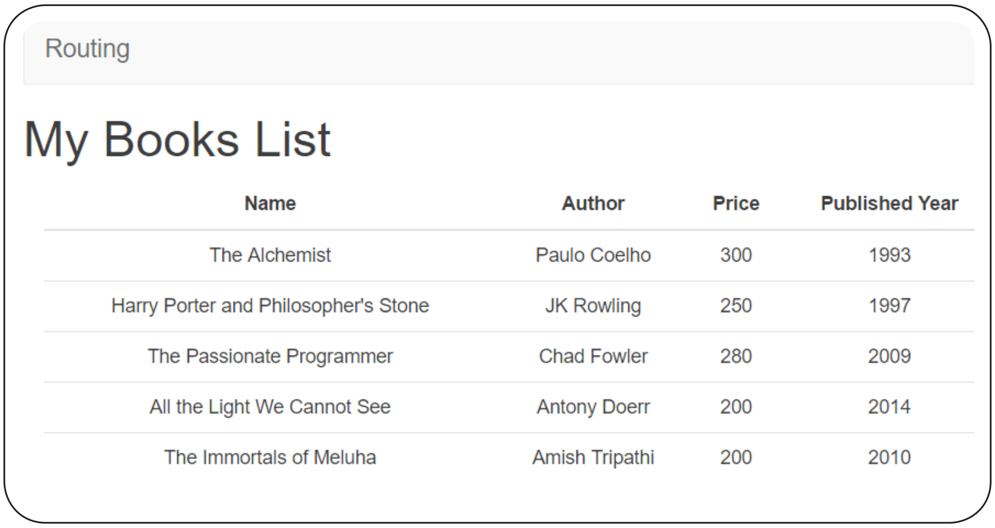
Listing 33.11: *Application module*

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { HttpModule } from "@angular/http";
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { BooksService } from "./books.service";
import { routing } from './app.routes';

@NgModule({
    declarations: [AppComponent],
    imports: [BrowserModule, HttpModule, FormsModule, routing],
    providers: [BooksService],
    bootstrap: [AppComponent]
})
class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

If you save all of the files and run the application, you will receive an error stating: Error during instantiation of LocationStrategy. To fix this, add a base path URL to the head section of index.html file. The following snippet shows this statement:

```
<base href="/" />
```

Upon refreshing the page after making the above change, the page renders a list of books. Figure 33.1 shows this:



Routing

My Books List

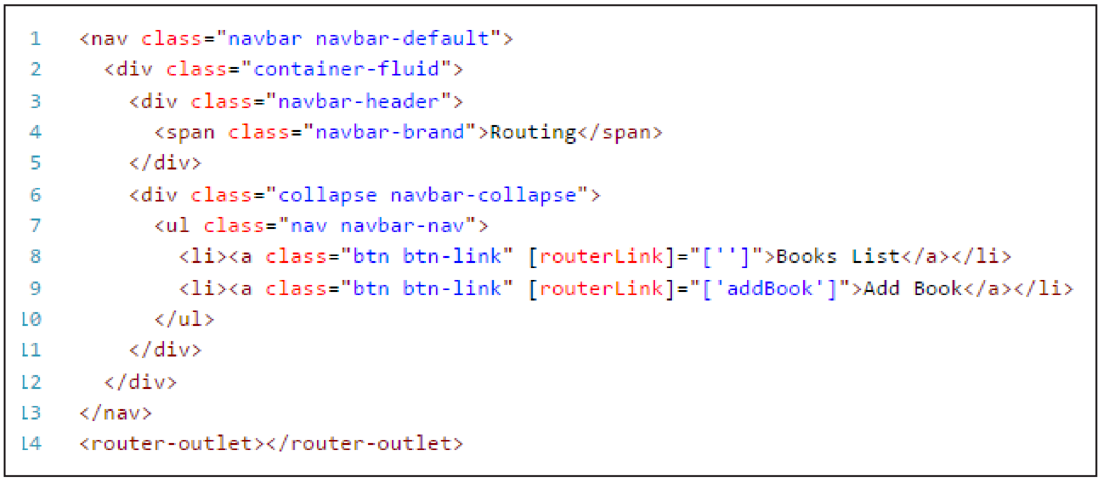
Name	Author	Price	Published Year
The Alchemist	Paulo Coelho	300	1993
Harry Porter and Philosopher's Stone	JK Rowling	250	1997
The Passionate Programmer	Chad Fowler	280	2009
All the Light We Cannot See	Antony Doerr	200	2014
The Immortals of Meluha	Amish Tripathi	200	2010

Figure 33.1: *Books list on a browser*

To view the *add book* page, change the route to http://localhost:3000/addBook. Changing the URL manually is an error prone process and it isn't user friendly too.

Let's add links to the *AppComponent* template to make this easier for us. Listing 33.12 shows the modified template of *AppComponent*:

Listing 33.12: *Navigation bar*



```
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-header">
      <span class="navbar-brand">Routing</span>
    </div>
    <div class="collapse navbar-collapse">
      <ul class="nav navbar-nav">
        <li><a class="btn btn-link" [routerLink]="['']">Books List</a></li>
        <li><a class="btn btn-link" [routerLink]="['addBook']">Add Book</a></li>
      </ul>
    </div>
  </div>
</nav>
<router-outlet></router-outlet>
```

Note the unordered list inside the third div element in the template. It has an unordered list with an anchor element specified in each of the list items. The anchor element has the *routerLink* directive, which is defined in the @angular/router module. This directive accepts a path and parameters to be supplied to the path, and also constructs the *href* attribute of the anchor element. Chapter 34 will cover use of parameters in routes. Now, if you run the application, you will see the navigation bar with the two links and you can click the links to switch between the two routes.

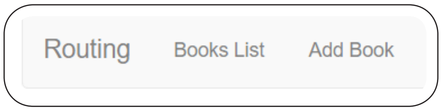
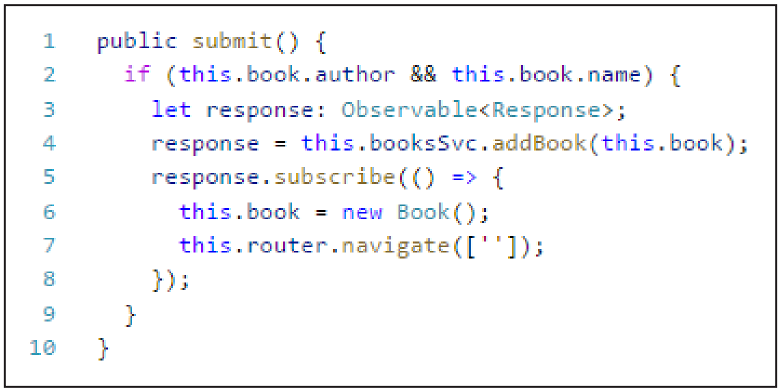


Figure 33.2: *Navigation bar*

In the add book page, redirect the user to books list page after adding a book. This can't be done in the view using the *routerLink* directive, as the book must be added to the list before redirecting (this can be achieved using the *Router* service of @angular/router module). This service is already injected into the component. With a method named *navigate,* a route can be passed to this method in the same format as we assigned the *routerLink* directive.

The navigation should happen after the book is added to the service. The statement for navigation has to be in the callback of the subscribe method. Listing 33.13 shows the modified *submit* method:

Listing 33.13: *Submit method with route redirection*



```
public submit() {
  if (this.book.author && this.book.name) {
    let response: Observable<Response>;
    response = this.booksSvc.addBook(this.book);
    response.subscribe(() => {
      this.book = new Book();
      this.router.navigate(['']);
    });
  }
}
```

Whenever a new book is added, the user is redirected to the list page.

Conclusion

Routing is a way to keep the application simpler and adds richness to the user experience. The next chapter will cover a several more features of routing in Angular 2.

Chapter 34

Parameterized Routes and Creating Sub-Routes

CHAPTER 33 EXAMINED HOW Angular 2 supports routing with some examples of using simple routes. These routes are made more dynamic by adding parameters. Angular 2 also supports nested routes, where a rendered component has a placeholder to load another component inside itself. This chapter focuses on these features.

To follow along with this chapter, keep a copy of the sample built in Chapter 33. This chapter extends the Books List sample built in the last chapter to include parameterized and nested routes, and it adds, reviews, and displays the list of reviews to demonstrate the features of routing. Before getting started, add a new folder inside the *app* folder and name it *reviews*. The steps discussed in this chapter will add code in this folder.

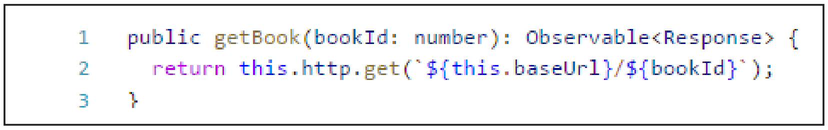
Parameterized Routes

Let's add reviews to the books. Every book may have a set of reviews. Each entry in the review will have name of the reviewer, rating (out of 5), and a comment. The table displaying a list of books will have a link that directs the user to a different page to display the list of reviews of that book. To do this, the reviews page needs the id of the book, and the books list page will pass the book id to the reviews page. The reviews page will read the book id from the UR and use it to call an API to obtain the reviews of that book.

If you need the list of books with their reviews and APIs to retrieve and add reviews, you may refer to the server.js and books.json files of the sample code provided with this chapter.

First, add a method to the *BooksService* to get details of a book when the id of the book is supplied. The book object returned from the API will contain the list of reviews in it. Listing 34.1 shows this method:

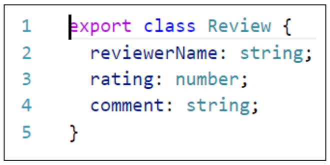
Listing 34.1: *Method to get details of a book*



```
public getBook(bookId: number): Observable<Response> {
  return this.http.get(`${this.baseUrl}/${bookId}`);
}
```

To represent the model of a review, the application needs a TypeScript model class. Add the following class to the file *book.model.ts:*

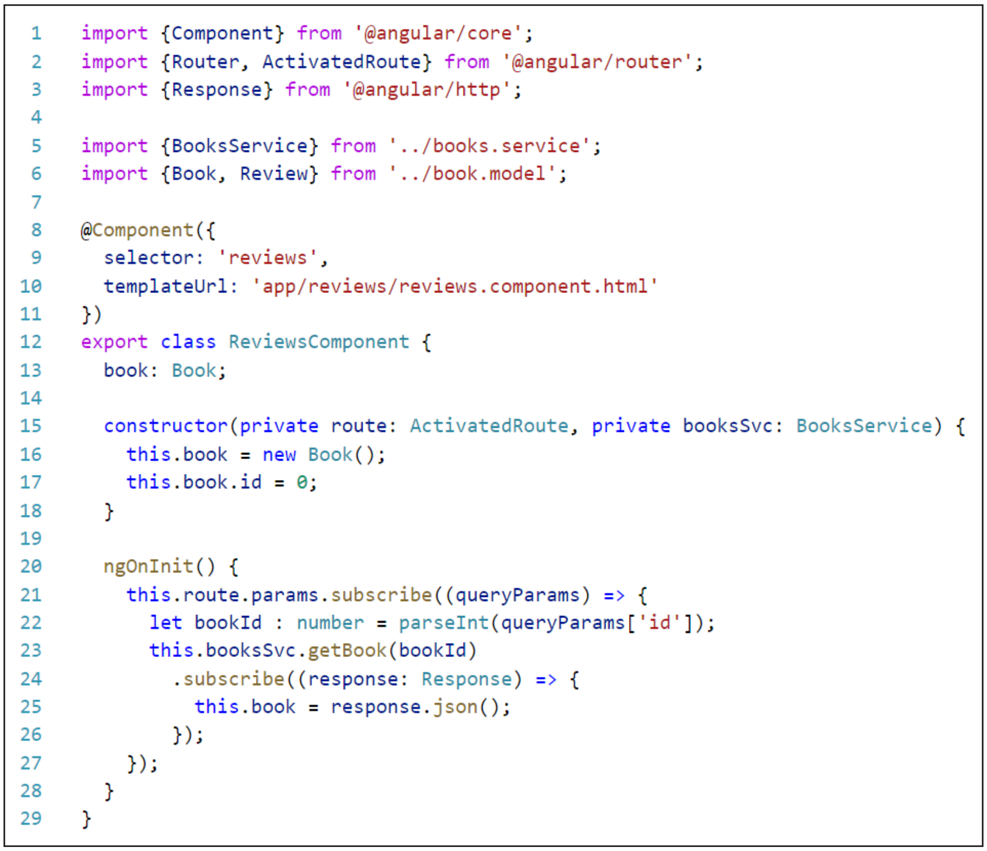
Listing 34.2: *Review class*



```
export class Review {
  reviewerName: string;
  rating: number;
  comment: string;
}
```

The application needs a component to display the list of reviews of a book. This component will read the id of the book from the route and use it to call the above method. Listing 34.3 shows the code for the *ReviewsComponent* class, add this code to a new file named *reviews.component.ts* in the *reviews* folder:

Listing 34.3: *Code of ReviewsComponent*

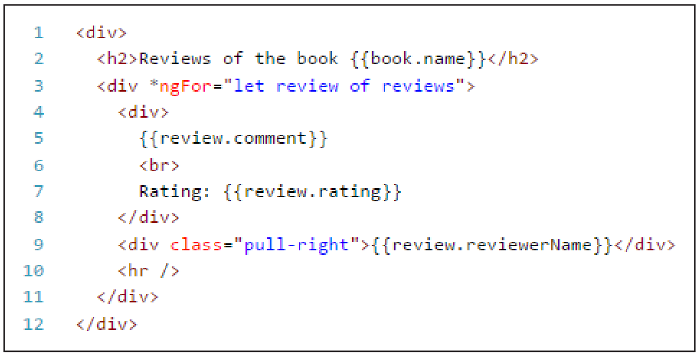


```
import {Component} from '@angular/core';
import {Router, ActivatedRoute} from '@angular/router';
import {Response} from '@angular/http';

import {BooksService} from '../books.service';
import {Book, Review} from '../book.model';

@Component({
  selector: 'reviews',
  templateUrl: 'app/reviews/reviews.component.html'
})
export class ReviewsComponent {
  book: Book;

  constructor(private route: ActivatedRoute, private booksSvc: BooksService) {
    this.book = new Book();
    this.book.id = 0;
  }

  ngOnInit() {
    this.route.params.subscribe((queryParams) => {
      let bookId : number = parseInt(queryParams['id']);
      this.booksSvc.getBook(bookId)
        .subscribe((response: Response) => {
          this.book = response.json();
        });
    });
  }
}
```

The *ActivatedRoute* service (used in the above class) provides access to the details of the current route. It provides access to the values passed as parameters to the current route through the *Observable* property *params*. The *ngOnInit* hook in listing 34.2 reads value of the parameter named id.

The template of this component will loop through the reviews and will display details of every review. Listing 34.4 shows the template:

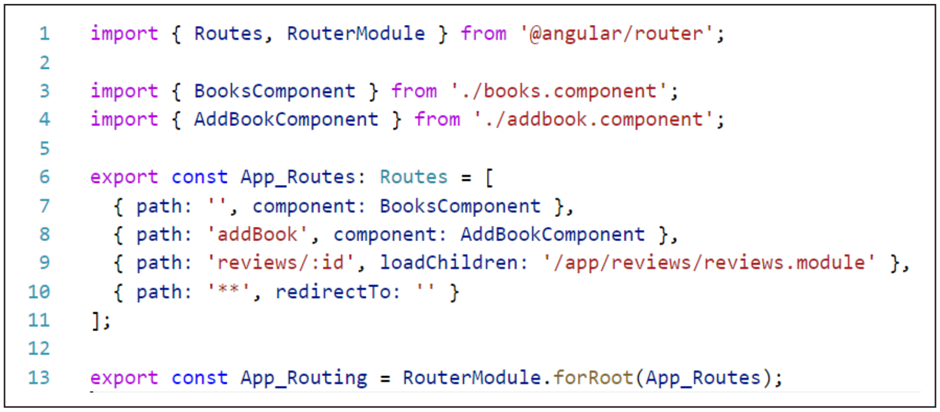
Listing 34.4: *Template of ReviewsComponent*



```
<div>
  <h2>Reviews of the book {{book.name}}</h2>
  <div *ngFor="let review of reviews">
    <div>
      {{review.comment}}
      <br>
      Rating: {{review.rating}}
    </div>
    <div class="pull-right">{{review.reviewerName}}</div>
    <hr />
  </div>
</div>
```

To reach this component, the application needs a route. Modify the routes list in the file *app.routes.ts* as following:

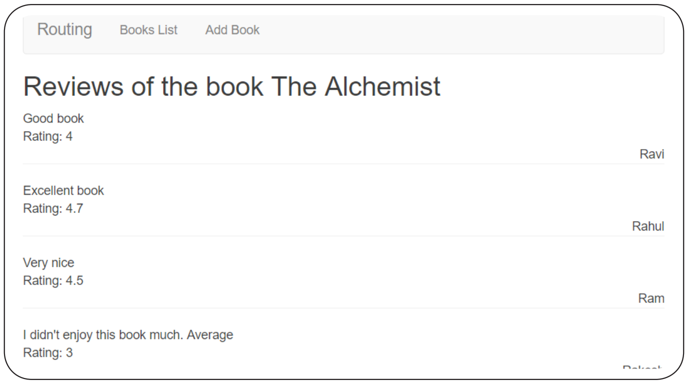
Listing 34.5: *Modified routes list*



```
import { Routes, RouterModule } from '@angular/router';

import { BooksComponent } from './books.component';
import { AddBookComponent } from './addbook.component';

export const App_Routes: Routes = [
  { path: '', component: BooksComponent },
  { path: 'addBook', component: AddBookComponent },
  { path: 'reviews/:id', loadChildren: '/app/reviews/reviews.module' },
  { path: '**', redirectTo: '' }
];

export const App_Routing = RouterModule.forRoot(App_Routes);
```

The third entry in the App_Routes array in listing 34.3 adds the parameterized reviews route. The syntax for passing a parameter to the route is by using a colon (:) in front of the name of the parameter. Any value passed after */reviews* in the URL is treated as value of this parameter.

Now you can change the URL in the browser to *http://localhost:3000/reviews/1* to see the list of reviews added for the first book and change the value of id to list reviews of other books.



Reviews of the book The Alchemist

Good book
Rating: 4
Ravi

Excellent book
Rating: 4.7
Rahul

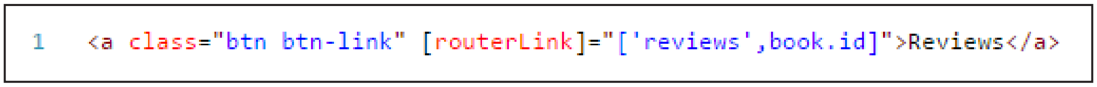
Very nice
Rating: 4.5
Ram

I didn't enjoy this book much. Average
Rating: 3

Figure 34.1: *Reviews page*

Finally, add a link to the books list table to navigate to the reviews page of the book. This doesn't require any changes to the component class. A link must be added in the books list table. Listing 34.6 shows the anchor element to be added to the row of the table:

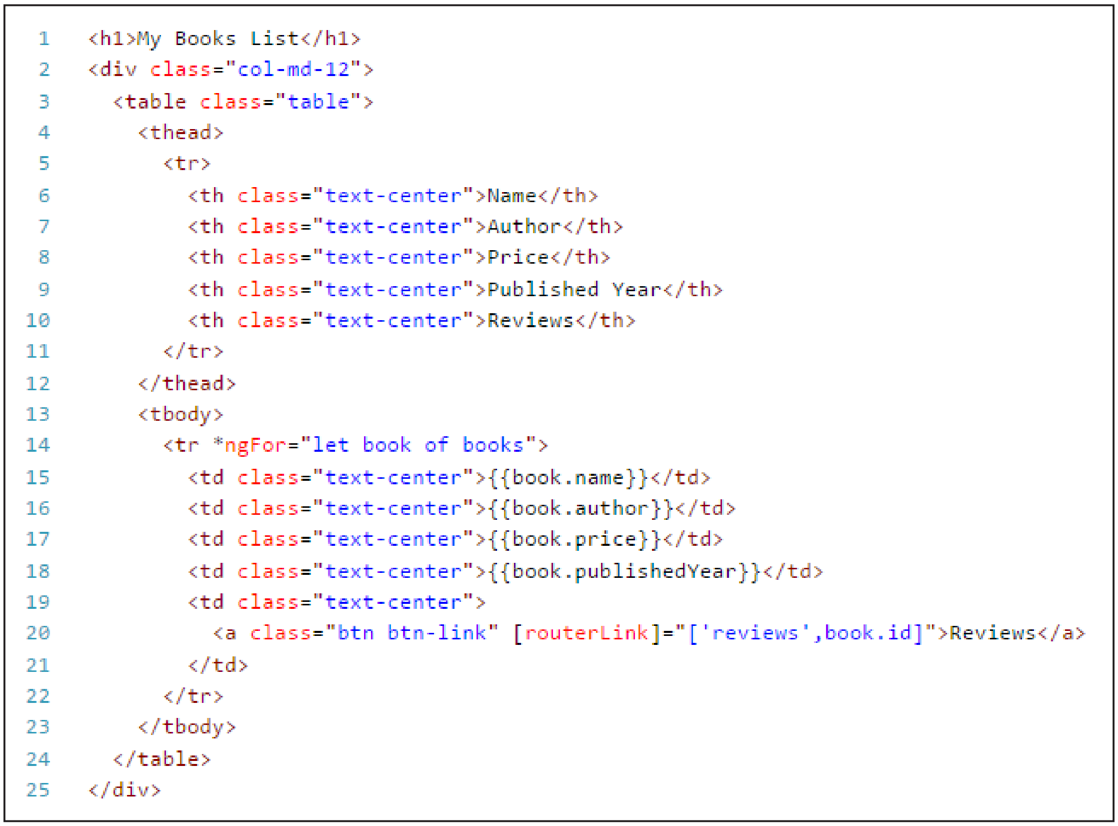
Listing 34.6: *Anchor tag with routerLink*



```
<a class="btn btn-link" [routerLink]="['reviews',book.id]">Reviews</a>
```

And listing 34.7 shows the modified template of the books.component.html file:

Listing 34.7: *Modified template of BooksComponent with a link to reviews page*



```
<h1>My Books List</h1>
<div class="col-md-12">
  <table class="table">
    <thead>
      <tr>
        <th class="text-center">Name</th>
        <th class="text-center">Author</th>
        <th class="text-center">Price</th>
        <th class="text-center">Published Year</th>
        <th class="text-center">Reviews</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let book of books">
        <td class="text-center">{{book.name}}</td>
        <td class="text-center">{{book.author}}</td>
        <td class="text-center">{{book.price}}</td>
        <td class="text-center">{{book.publishedYear}}</td>
        <td class="text-center">
          <a class="btn btn-link" [routerLink]="['reviews',book.id]">Reviews</a>
        </td>
      </tr>
    </tbody>
  </table>
</div>
```

Now you should be able to navigate to the reviews page using the link in the table.

Adding Nested Routes

Now that the application lists the reviews posted against a book, it would be optimal for it to provide a way to add reviews, too. As viewing the list of reviews and adding a new review to a book are two different pieces of functionality, it would be better to make them nested routes under reviews.

When a user clicks the reviews link on the books list table, the page will navigate to the reviews section, which will have two sub routes:

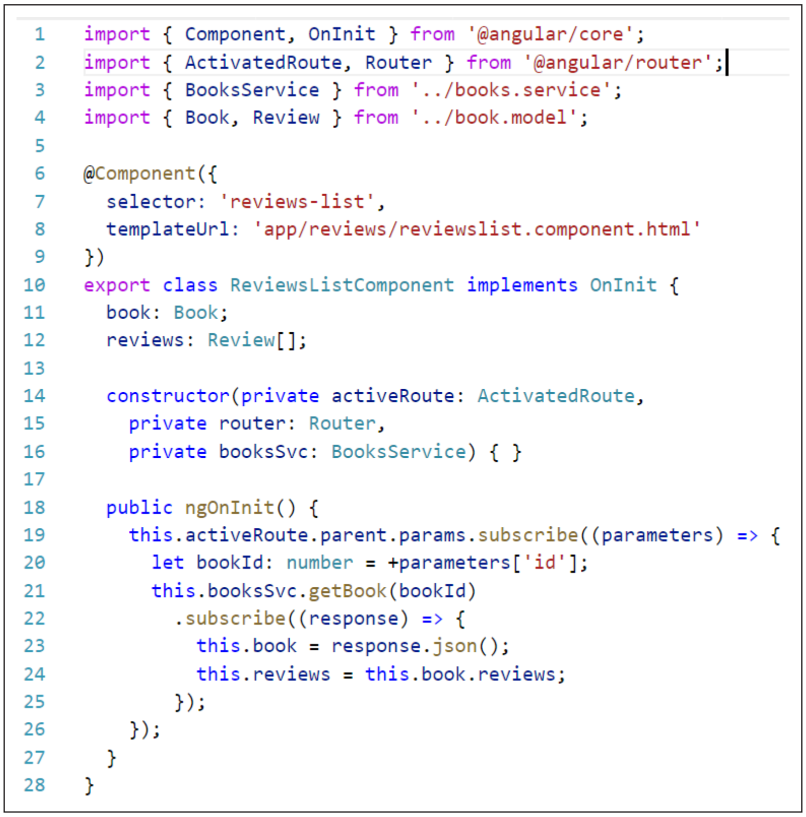
- Reviews list
- Add new review

The *ReviewersComponent* created in the previous section will become a host of routes and will have a placeholder to display one of these views in it. The nested routing in Angular 2 needs a submodule with the module-specific routes defined in the module. This module will then be loaded from the route config of the main module when the root route corresponding to the submodule is loaded.

Create a separate folder under the *app* folder and name it *reviews*. This module will hold functionality of the reviews section. Move the file containing code of *ReviewsComponent* to this folder.

Before defining routes and the sub module, let's build the required components and modify the reviews component.

Listing 34.8: *Code of ReviewsListComponent*

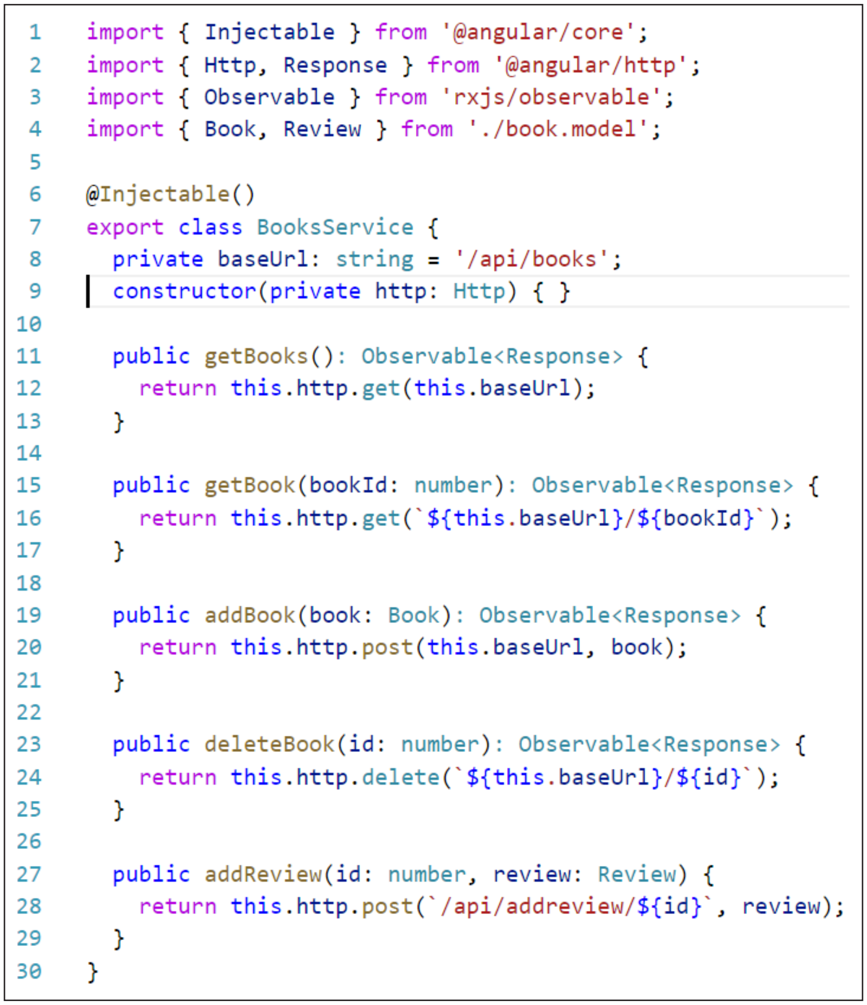


```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Router } from '@angular/router';
import { BooksService } from '../books.service';
import { Book, Review } from '../book.model';

@Component({
  selector: 'reviews-list',
  templateUrl: 'app/reviews/reviewslist.component.html'
})
export class ReviewsListComponent implements OnInit {
  book: Book;
  reviews: Review[];

  constructor(private activeRoute: ActivatedRoute,
    private router: Router,
    private booksSvc: BooksService) { }

  public ngOnInit() {
    this.activeRoute.parent.params.subscribe((parameters) => {
      let bookId: number = +parameters['id'];
      this.booksSvc.getBook(bookId)
        .subscribe((response) => {
          this.book = response.json();
          this.reviews = this.book.reviews;
        });
    });
  }
}
```

Most of the code of this component is the same as the *ReviewsComponent* written earlier. One significant difference is in the way the value of the book id from the URL is fetched. Note the first statement in the *ngOnInit* lifecycle hook. It reads the value of the book id from the parent route and converts it into a number using the plus sign. The template of this component has a div element iterating over the reviews and displaying different properties of the review object.

To add a new review, a method must be added in the *BooksService* to call the add review service. Listing 34.9 shows the method:

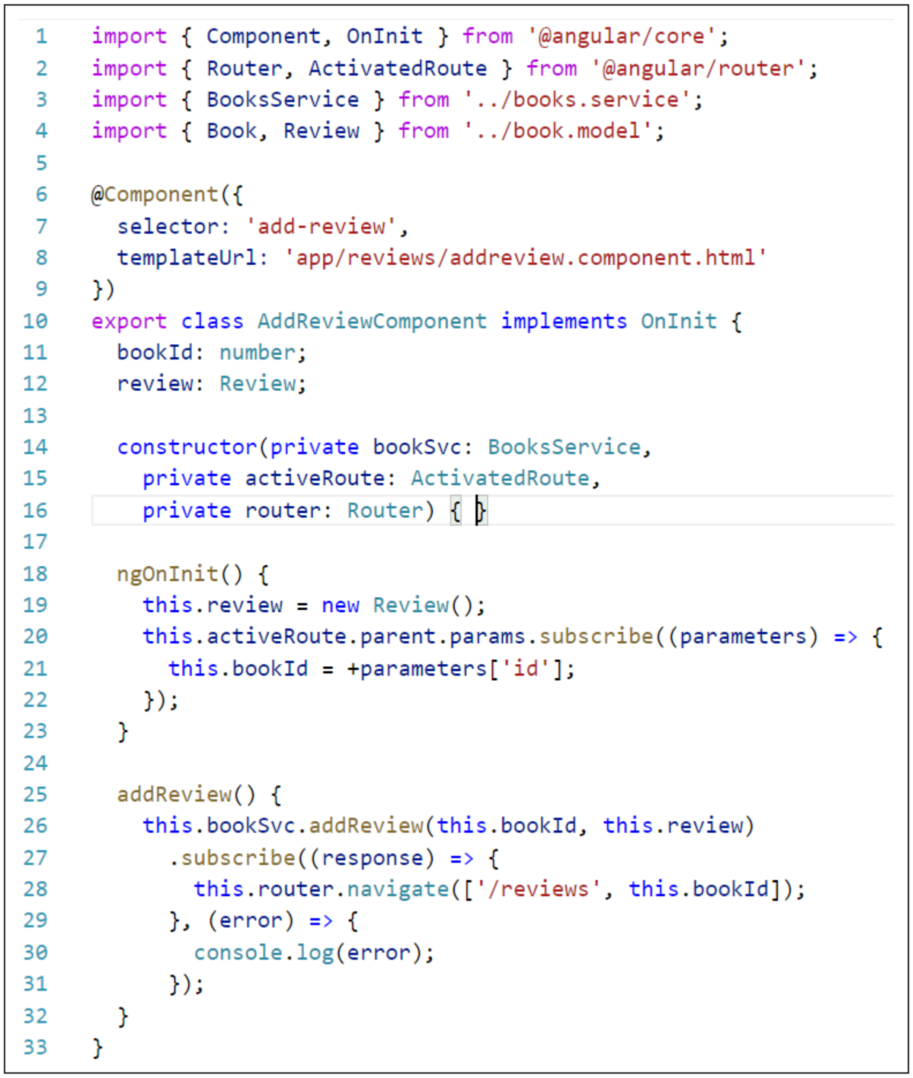
Listing 34.9: *BooksService updated with a method to add a review*



```
import { Injectable } from '@angular/core';
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/observable';
import { Book, Review } from './book.model';

@Injectable()
export class BooksService {
  private baseUrl: string = '/api/books';
  constructor(private http: Http) { }

  public getBooks(): Observable<Response> {
    return this.http.get(this.baseUrl);
  }

  public getBook(bookId: number): Observable<Response> {
    return this.http.get(`${this.baseUrl}/${bookId}`);
  }

  public addBook(book: Book): Observable<Response> {
    return this.http.post(this.baseUrl, book);
  }

  public deleteBook(id: number): Observable<Response> {
    return this.http.delete(`${this.baseUrl}/${id}`);
  }

  public addReview(id: number, review: Review) {
    return this.http.post(`/api/addreview/${id}`, review);
  }
}
```

The component to add a new review will have a form accepting the three fields of the review object. It will call the *addReview* method of the service to add the review to the book. Listing 34.10 shows the component class:

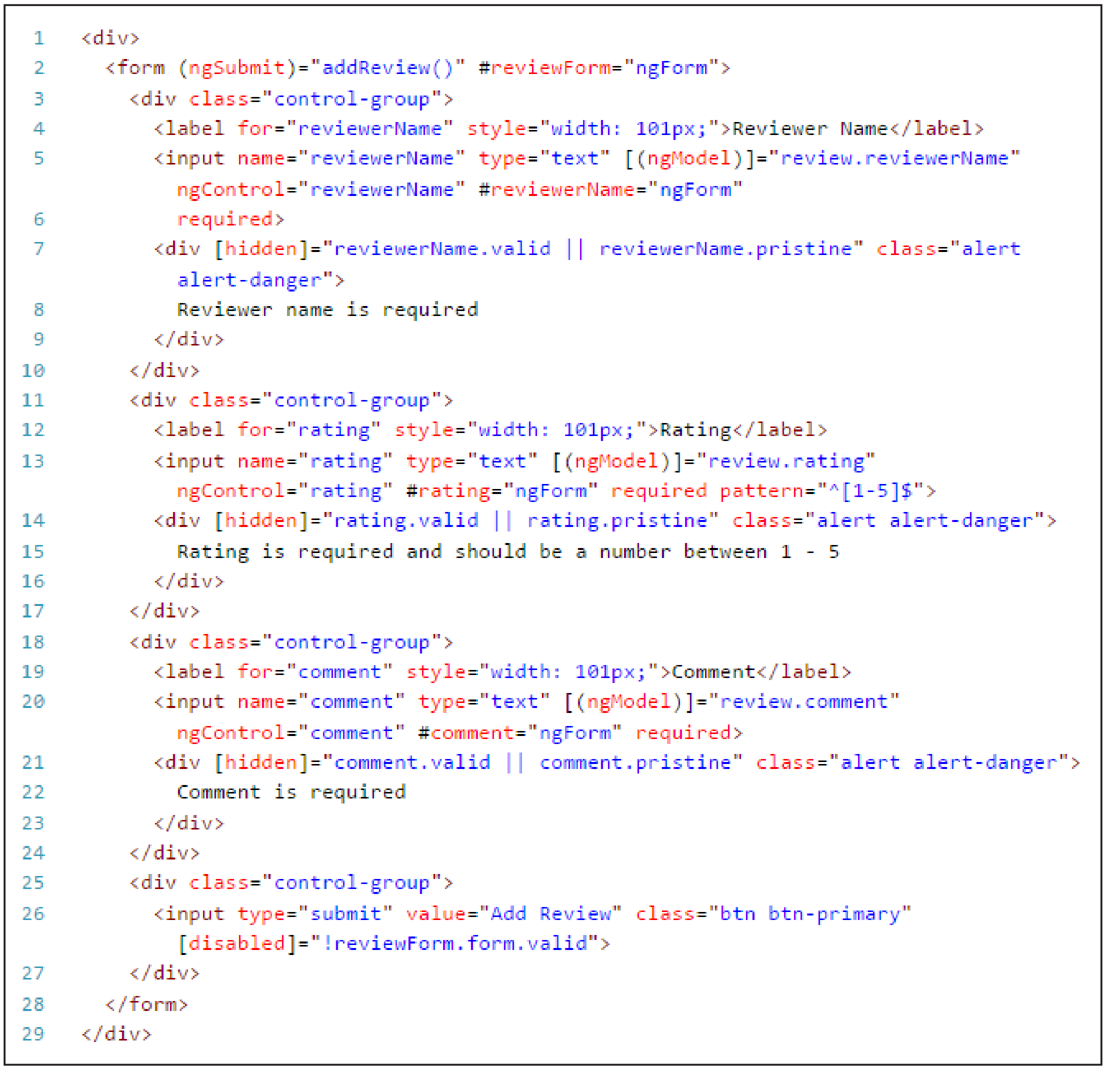
Listing 34.10: *Code of AddReviewComponent*



```
import { Component, OnInit } from '@angular/core';
import { Router, ActivatedRoute } from '@angular/router';
import { BooksService } from '../books.service';
import { Book, Review } from '../book.model';

@Component({
  selector: 'add-review',
  templateUrl: 'app/reviews/addreview.component.html'
})
export class AddReviewComponent implements OnInit {
  bookId: number;
  review: Review;

  constructor(private bookSvc: BooksService,
    private activeRoute: ActivatedRoute,
    private router: Router) { }

  ngOnInit() {
    this.review = new Review();
    this.activeRoute.parent.params.subscribe((parameters) => {
      this.bookId = +parameters['id'];
    });
  }

  addReview() {
    this.bookSvc.addReview(this.bookId, this.review)
      .subscribe((response) => {
        this.router.navigate(['/reviews', this.bookId]);
      }, (error) => {
        console.log(error);
      });
  }
}
```

Listing 34.11 shows the template of this component. It has a template-driven form validated using the validation directives in Angular 2.

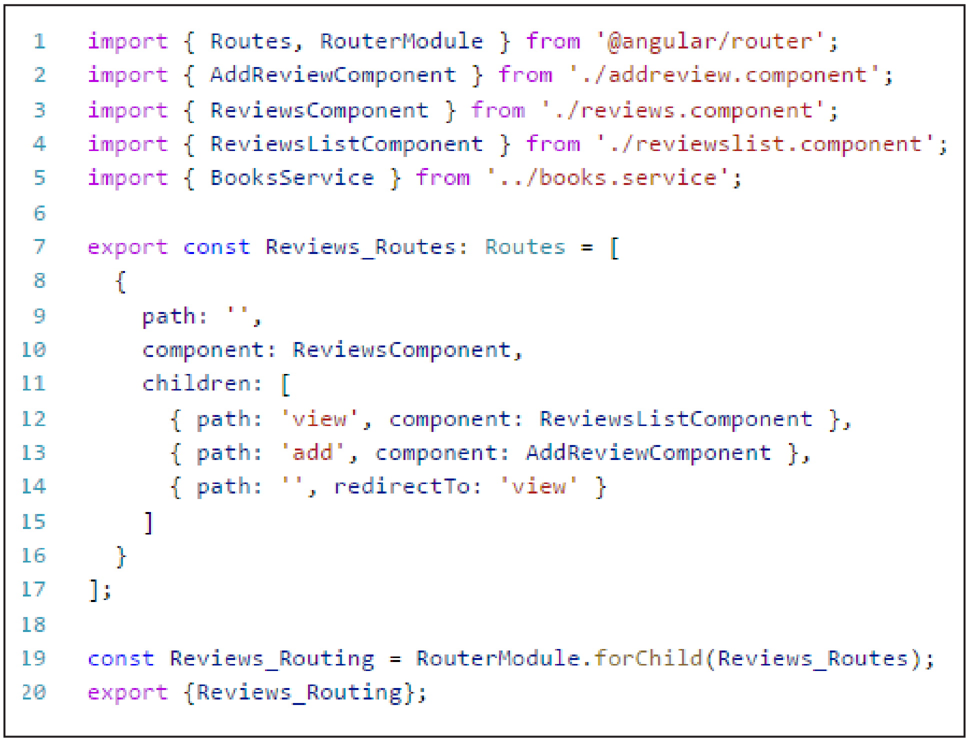
Listing 34.11: *Template of AddReviewComponent*



```
<div>
  <form (ngSubmit)="addReview()" #reviewForm="ngForm">
    <div class="control-group">
      <label for="reviewerName" style="width: 101px;">Reviewer Name</label>
      <input name="reviewerName" type="text" [(ngModel)]="review.reviewerName"
        ngControl="reviewerName" #reviewerName="ngForm"
        required>
      <div [hidden]="reviewerName.valid || reviewerName.pristine" class="alert
        alert-danger">
        Reviewer name is required
      </div>
    </div>
    <div class="control-group">
      <label for="rating" style="width: 101px;">Rating</label>
      <input name="rating" type="text" [(ngModel)]="review.rating"
        ngControl="rating" #rating="ngForm" required pattern="^[1-5]$">
      <div [hidden]="rating.valid || rating.pristine" class="alert alert-danger">
        Rating is required and should be a number between 1 - 5
      </div>
    </div>
    <div class="control-group">
      <label for="comment" style="width: 101px;">Comment</label>
      <input name="comment" type="text" [(ngModel)]="review.comment"
        ngControl="comment" #comment="ngForm" required>
      <div [hidden]="comment.valid || comment.pristine" class="alert alert-danger">
        Comment is required
      </div>
    </div>
    <div class="control-group">
      <input type="submit" value="Add Review" class="btn btn-primary"
        [disabled]="!reviewForm.form.valid">
    </div>
  </form>
</div>
```

After the components are ready, it is time to define the routes. As these routes will be children of the *ReviewsComponent*, the route configuration object must mention the *ReviewsComponent* and define the routes as such. To keep the routing files simpler, add a new file to the reviews folder named reviews.routes.ts and add the code shown in listing 34.12 to it:

Listing 34.12: *Routes of reviews*

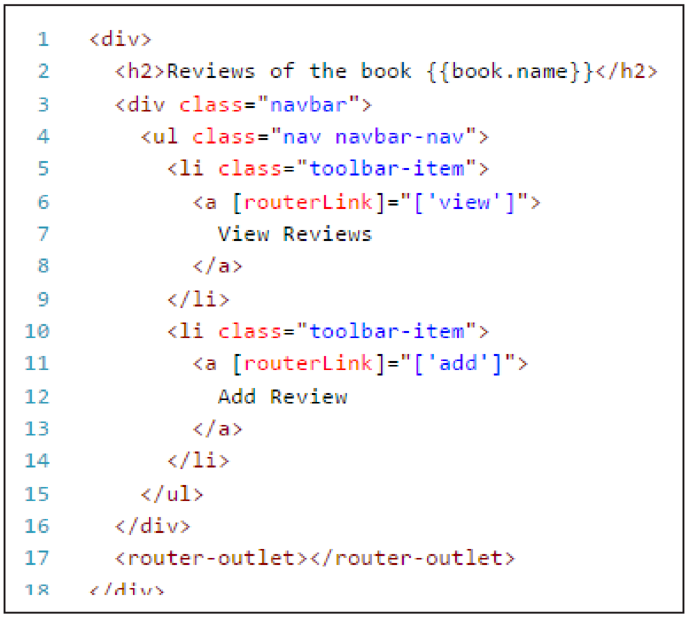


```
import { Routes, RouterModule } from '@angular/router';
import { AddReviewComponent } from './addreview.component';
import { ReviewsComponent } from './reviews.component';
import { ReviewsListComponent } from './reviewslist.component';
import { BooksService } from '../books.service';

export const Reviews_Routes: Routes = [
  {
    path: '',
    component: ReviewsComponent,
    children: [
      { path: 'view', component: ReviewsListComponent },
      { path: 'add', component: AddReviewComponent },
      { path: '', redirectTo: 'view' }
    ]
  }
];

const Reviews_Routing = RouterModule.forChild(Reviews_Routes);
export {Reviews_Routing};
```

Notice the way the routes are defined in the listing 34.10. The route configuration says the main component for the sub-route is *ReviewsComponent* and then it lists the nested routes in the *children* array. This means that *ReviewsComponent* will have placeholders for the children routes. Then a module is created for the child routes and is exported from the file. This module will be imported in the sub-module of reviews.

The *ReviewsComponent* needs to be updated to play the role of an entry point for these sub-routes. This component will not show the list of reviews, instead it will have the header of the page and will have navigation links to switch between the reviews list and add review views. It will also have a *router-outlet* component wherein the sub-routes will be rendered. Listing 34.13 shows the modified template of the *ReviewsComponent*:

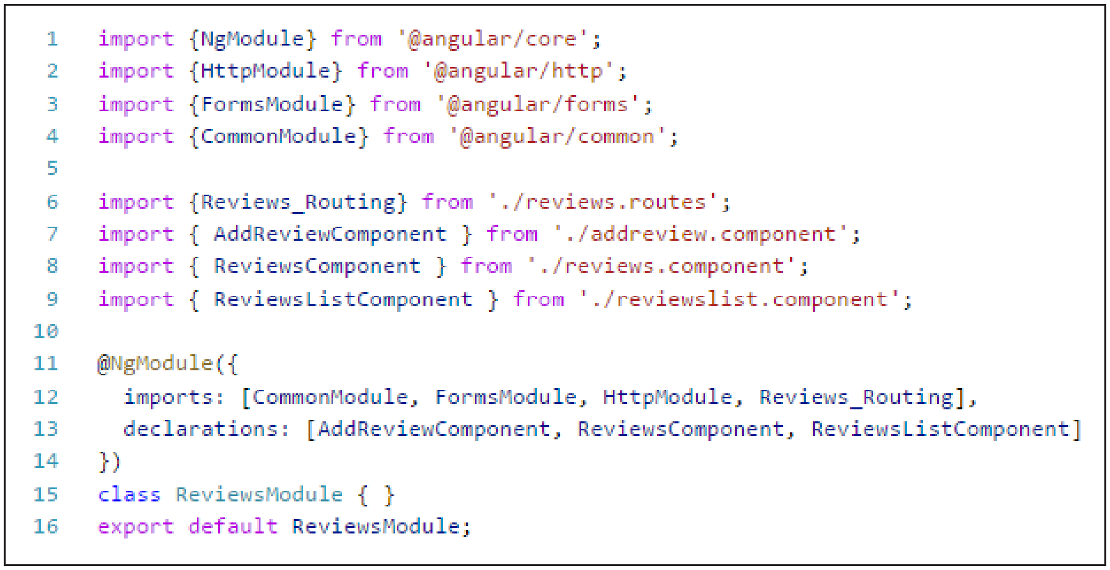
Listing 34.13: *Modified template of ReviewsComponent*



```
<div>
  <h2>Reviews of the book {{book.name}}</h2>
  <div class="navbar">
    <ul class="nav navbar-nav">
      <li class="toolbar-item">
        <a [routerLink]="['view']">
          View Reviews
        </a>
      </li>
      <li class="toolbar-item">
        <a [routerLink]="['add']">
          Add Review
        </a>
      </li>
    </ul>
  </div>
  <router-outlet></router-outlet>
</div>
```

The last thing to be done in the reviews folder is to add the sub module. This module will be responsible to register the components of reviews section and to import the route module created earlier. Add a new file to the *reviews* folder and name it *reviews.module.ts.* Listing 34.14 shows the code of the to be added to this file:

Listing 34.14: *Code of ReviewsModule*



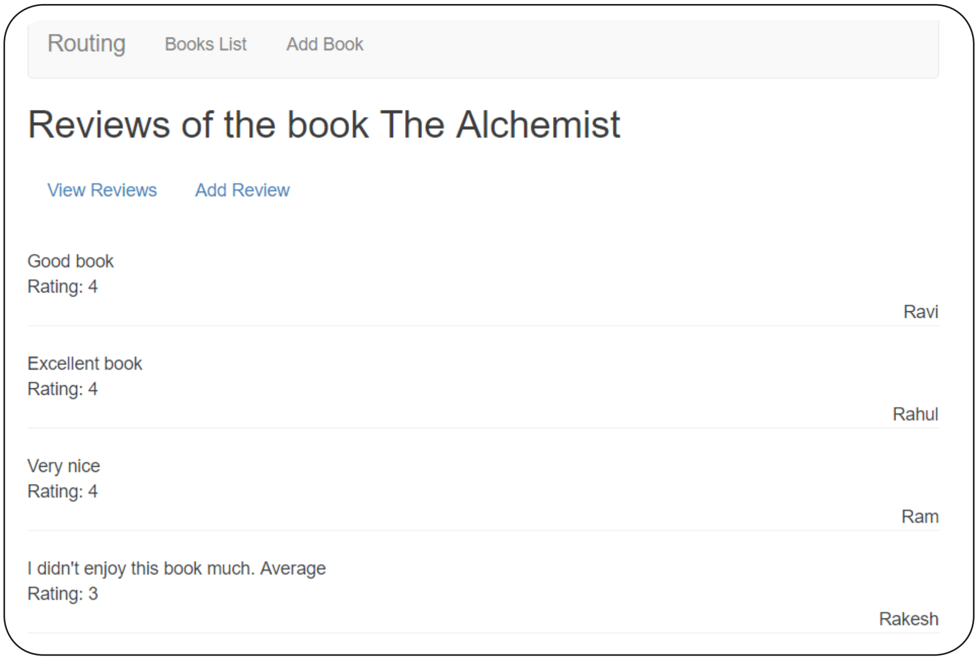
```
import {NgModule} from '@angular/core';
import {HttpModule} from '@angular/http';
import {FormsModule} from '@angular/forms';
import {CommonModule} from '@angular/common';

import {Reviews_Routing} from './reviews.routes';
import { AddReviewComponent } from './addreview.component';
import { ReviewsComponent } from './reviews.component';
import { ReviewsListComponent } from './reviewslist.component';

@NgModule({
  imports: [CommonModule, FormsModule, HttpModule, Reviews_Routing],
  declarations: [AddReviewComponent, ReviewsComponent, ReviewsListComponent]
})
class ReviewsModule { }
export default ReviewsModule;
```

Everything is ready in the reviews folder and it needs to be linked in the main application module. The reviews module must be linked in the reviews route of the application. Modify the route configuration in *app.routes.ts* as shown in the listing 34.15:

Listing 34.15: *Modified routes in app.routes.ts*

```
export const App_Routes: Routes = [
  { path: '', component: BooksComponent },
  { path: 'addBook', component: AddBookComponent },
  { path: 'reviews/:id', loadChildren: '/app/reviews/reviews.module' },
  { path: '**', redirectTo: '' }
];
```

If you run the application now and click on reviews of one of the rows in the table, you will see a view similar to the following figure:



Reviews of the book The Alchemist

Good book
Rating: 4
Ravi

Excellent book
Rating: 4
Rahul

Very nice
Rating: 4
Ram

I didn't enjoy this book much. Average
Rating: 3
Rakesh

Figure 34.2: *Reviews page*

Conclusion

A good routing system helps make the application richer and more useful. Angular 2 comes with a flexible and feature rich router to solve most of the problems with routing in SPA. Chapters 33 and 34 explained the process of creating and using simple, parameterized, and nested routes.

Chapter 35

Angular 2: Unit Testing Framework

TESTING YOUR CODE BEFORE it is moved to production is important. You should be certain that each unit of code is successfully executed as per the logic defined.

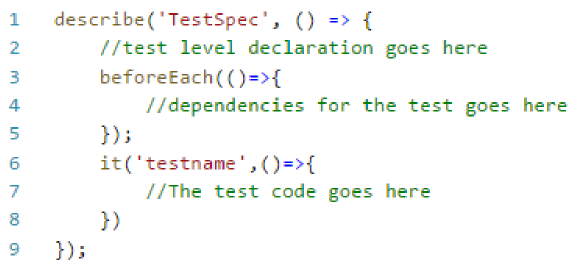
This chapter will explain the features offered by Angular 2 for implementing Unit Testing.

Similar to Angular 1, Angular 2 was designed with testability in mind, and it provides multiple options for unit testing. Jasmine is a behavior-driven JavaScript framework for testing JavaScript code and does not rely on browsers, DOM, or any JavaScript framework. It provides an easy syntax for writing scripts. In Jasmine, the test starts from the global function **describe.** This function accepts two parameters: a string (which is the name of the spec), and the function (which is a block of the spec suite). This function contains the code for the test initialization, setting up dependencies (beforeEach) and the tests (it).

beforeEach() accepts a function which can be used to setup the objects required for a group of tests. For example, if an Angular service needs to be tested then the beforeEach block creates an instance of the service.

The **it()** function is used to write the code for testing. This function accepts two parameters: a string (which is the name of spec), and the function (which contains the logic for the test). Based on the unit testing requirements, this code may call the actual object of the function to be tested, or may use a mocking framework.

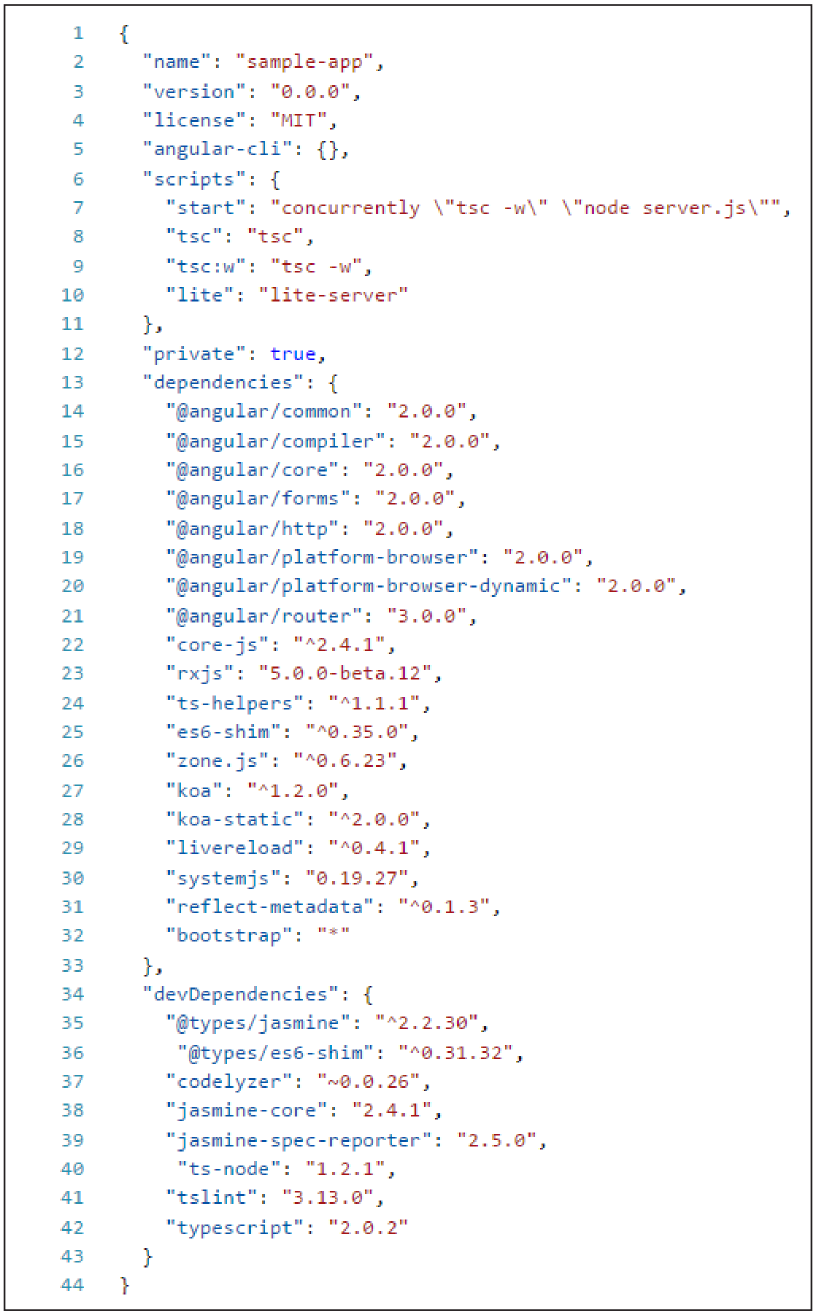
The outline of the code is as following:



```
describe('TestSpec', () => {
    //test level declaration goes here
    beforeEach(()=>{
        //dependencies for the test goes here
    });
    it('testname',()=>{
        //The test code goes here
    })
});
```

Angular 2 provides an out-of-box support for testing by providing Angular 2 testing libraries. These libraries are provided in separate packages and can be used for testing Angular 2 Component, Services, etc. Package.json needs to be modified for testing as shown in listing 35.1:

Listing 35.1: *package.config for specifying packages needed for testing*



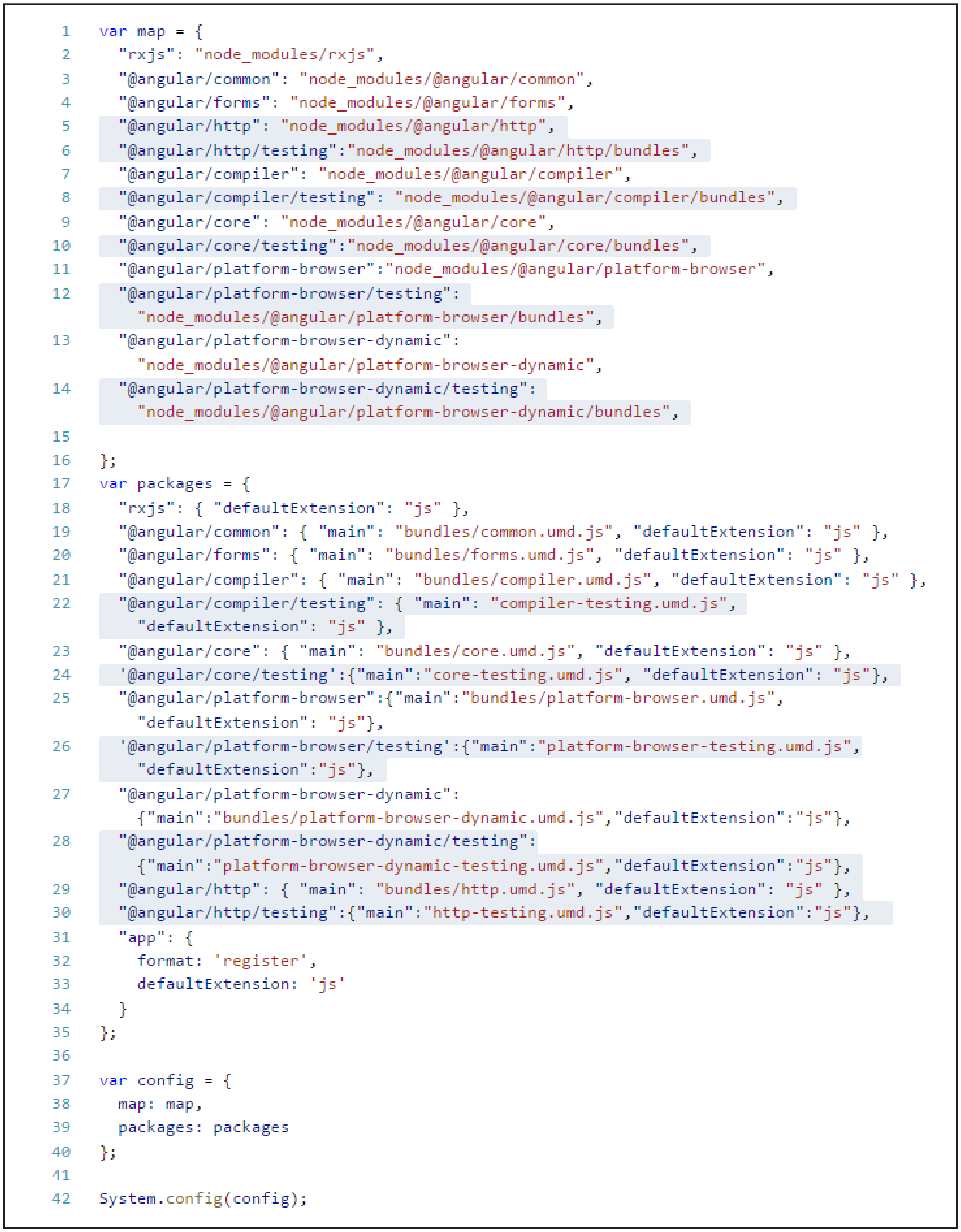
```
{
  "name": "sample-app",
  "version": "0.0.0",
  "license": "MIT",
  "angular-cli": {},
  "scripts": {
    "start": "concurrently \"tsc -w\" \"node server.js\"",
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "lite": "lite-server"
  },
  "private": true,
  "dependencies": {
    "@angular/common": "2.0.0",
    "@angular/compiler": "2.0.0",
    "@angular/core": "2.0.0",
    "@angular/forms": "2.0.0",
    "@angular/http": "2.0.0",
    "@angular/platform-browser": "2.0.0",
    "@angular/platform-browser-dynamic": "2.0.0",
    "@angular/router": "3.0.0",
    "core-js": "^2.4.1",
    "rxjs": "5.0.0-beta.12",
    "ts-helpers": "^1.1.1",
    "es6-shim": "^0.35.0",
    "zone.js": "^0.6.23",
    "koa": "^1.2.0",
    "koa-static": "^2.0.0",
    "livereload": "^0.4.1",
    "systemjs": "0.19.27",
    "reflect-metadata": "^0.1.3",
    "bootstrap": "*"
  },
  "devDependencies": {
    "@types/jasmine": "^2.2.30",
     "@types/es6-shim": "^0.31.32",
    "codelyzer": "~0.0.26",
    "jasmine-core": "2.4.1",
    "jasmine-spec-reporter": "2.5.0",
     "ts-node": "1.2.1",
    "tslint": "3.13.0",
    "typescript": "2.0.2"
  }
}
```

The **jasmine-core** and **jasmine-spec-reporter** will install required packages for Jasmine framework used for testing. These packages can be installed for the current project using the following command:

```
npm install
```

Once these packages are installed, the systemjs.config.js file must be modified for defining the library paths for testing. The following snippet provides a listing of all testing libraries from the systemjs.config.js file.

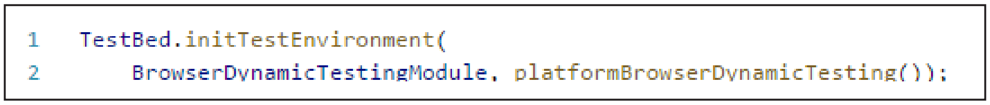
Listing 35.2: *The Angular 2 testing libraries path (highlighted)*



```
var map = {
  "rxjs": "node_modules/rxjs",
  "@angular/common": "node_modules/@angular/common",
  "@angular/forms": "node_modules/@angular/forms",
  "@angular/http": "node_modules/@angular/http",
  "@angular/http/testing":"node_modules/@angular/http/bundles",
  "@angular/compiler": "node_modules/@angular/compiler",
  "@angular/compiler/testing": "node_modules/@angular/compiler/bundles",
  "@angular/core": "node_modules/@angular/core",
  "@angular/core/testing":"node_modules/@angular/core/bundles",
  "@angular/platform-browser":"node_modules/@angular/platform-browser",
  "@angular/platform-browser/testing":
    "node_modules/@angular/platform-browser/bundles",
  "@angular/platform-browser-dynamic":
    "node_modules/@angular/platform-browser-dynamic",
  "@angular/platform-browser-dynamic/testing":
    "node_modules/@angular/platform-browser-dynamic/bundles",

};
var packages = {
  "rxjs": { "defaultExtension": "js" },
  "@angular/common": { "main": "bundles/common.umd.js", "defaultExtension": "js" },
  "@angular/forms": { "main": "bundles/forms.umd.js", "defaultExtension": "js" },
  "@angular/compiler": { "main": "bundles/compiler.umd.js", "defaultExtension": "js" },
  "@angular/compiler/testing": { "main": "compiler-testing.umd.js",
    "defaultExtension": "js" },
  "@angular/core": { "main": "bundles/core.umd.js", "defaultExtension": "js" },
  '@angular/core/testing':{"main":"core-testing.umd.js", "defaultExtension": "js"},
  "@angular/platform-browser":{"main":"bundles/platform-browser.umd.js",
    "defaultExtension": "js"},
  '@angular/platform-browser/testing':{"main":"platform-browser-testing.umd.js",
    "defaultExtension":"js"},
  "@angular/platform-browser-dynamic":
    {"main":"bundles/platform-browser-dynamic.umd.js","defaultExtension":"js"},
  "@angular/platform-browser-dynamic/testing":
    {"main":"platform-browser-dynamic-testing.umd.js","defaultExtension":"js"},
  "@angular/http": { "main": "bundles/http.umd.js", "defaultExtension": "js" },
  "@angular/http/testing":{"main":"http-testing.umd.js","defaultExtension":"js"},
  "app": {
    format: 'register',
    defaultExtension: 'js'
  }
};

var config = {
  map: map,
  packages: packages
};

System.config(config);
```

The application needs these libraries for writing tests. To test Angular 2 code, the **TestBed** class must be used. This class is provided in @angular/core/testing package and uses **BrowserTestingModule** class which is available in @angular/platform-browser-dynamic/testing package. The TestBed class is used to initialize the testing environment. A snippet of it is shown in Listing 35.3.

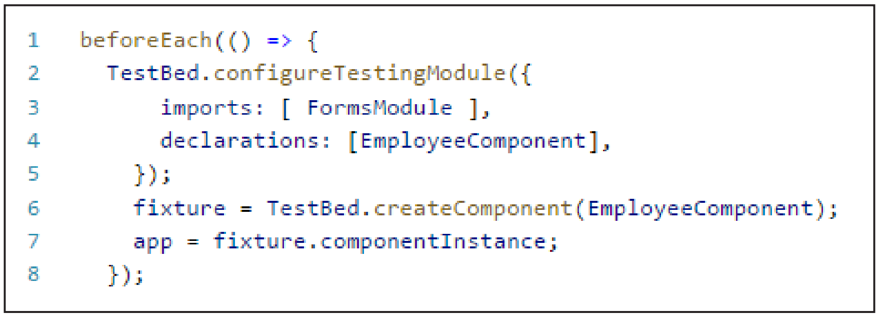
Listing 35.3: *TestBed class to initialize the testing environment*



```
TestBed.initTestEnvironment(
    BrowserDynamicTestingModule, platformBrowserDynamicTesting());
```

While executing Component tests in Angular 2, the Component must be compiled using the **ComponentFixture** class. This class is also provided in @angular/core/testing package. The TestBed must configure testing module before compiling component, which can be done using the static method **configureTestingModule()** of the TestBed class. This method must be passed to the component declaration that's being tested and to dependencies of the modules which are used by the component. For instance, if the component being tested is EmployeeComponent and it is using the directive ngModel, then it requires FormsModule. In this case, the TestBed will initialize modules as shown in the following listing:

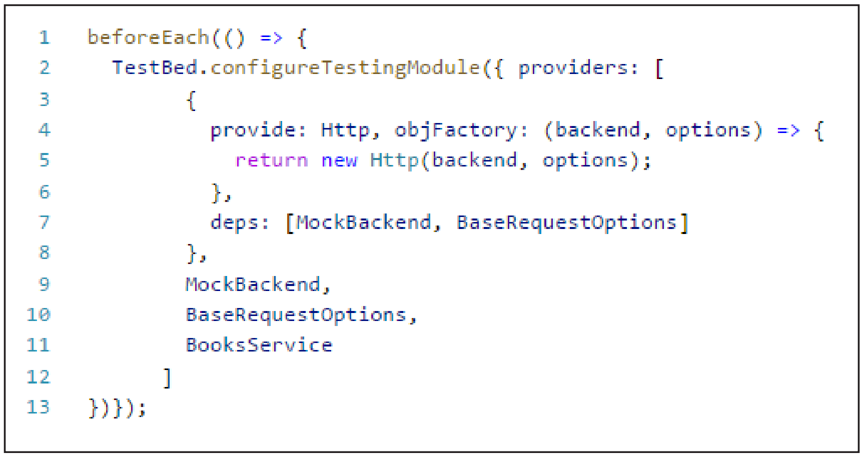
Listing 35.4: *The Initialization of Testing Module*



```
beforeEach(() => {
  TestBed.configureTestingModule({
      imports: [ FormsModule ],
      declarations: [EmployeeComponent],
    });
    fixture = TestBed.createComponent(EmployeeComponent);
    app = fixture.componentInstance;
  });
```

In case of testing an Angular 2 Service that performs an AJAX call using the *Http* service, the module configuration must include the service providers of Http and MockBackend as shown in Listing 35.5.

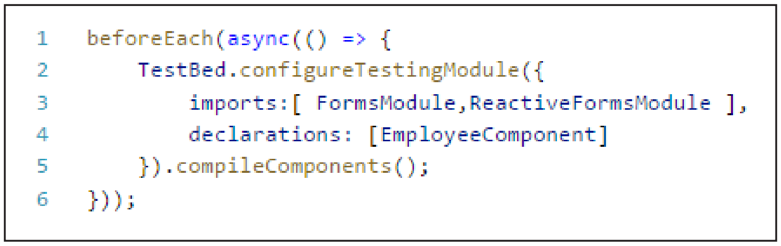
Listing 35.5: *module initialization in case of Http service testing*



```
beforeEach(() => {
  TestBed.configureTestingModule({ providers: [
        {
          provide: Http, objFactory: (backend, options) => {
            return new Http(backend, options);
          },
          deps: [MockBackend, BaseRequestOptions]
        },
        MockBackend,
        BaseRequestOptions,
        BooksService
      ]
})});
```

If the test is written for the Component using an external Html containing a form, then the external template must be first compiled using the TestBed.compileComponents() method. In this case, the module initialization will be as per listing 36.6.

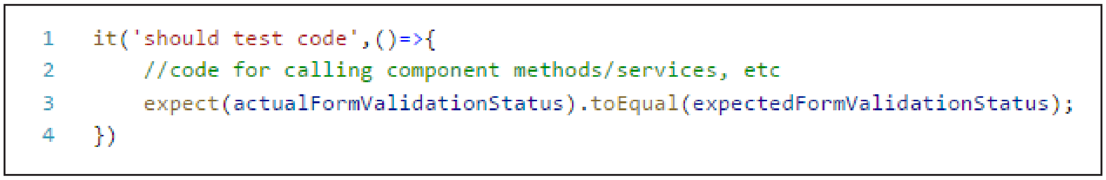
Listing 36.6: *Module initialization in case of external html template in component*



```
beforeEach(async(() => {
    TestBed.configureTestingModule({
        imports:[ FormsModule,ReactiveFormsModule ],
        declarations: [EmployeeComponent]
    }).compileComponents();
}));
```

Once the module initialization is completed, the spec may be implemented using **it()** function which sets the expectation for testing using **expect()** function. See Listing 36.7.

Listing 36.7: *The expect function in 'it()'*



```
it('should test code',()=>{
    //code for calling component methods/services, etc
    expect(actualFormValidationStatus).toEqual(expectedFormValidationStatus);
})
```

A complete implementation of Angular 2 testing with Component Testing, Service Testing, Http Service Testing, and Form Testing is forthcoming in Chapters 36 through 39.

Conclusion

This article explained the features provides by Angular 2 for unit-testing your application using the Jasmine framework. Chapter 36 explains the steps of writing a unit test for an Angular 2 component using Jasmine.

Chapter

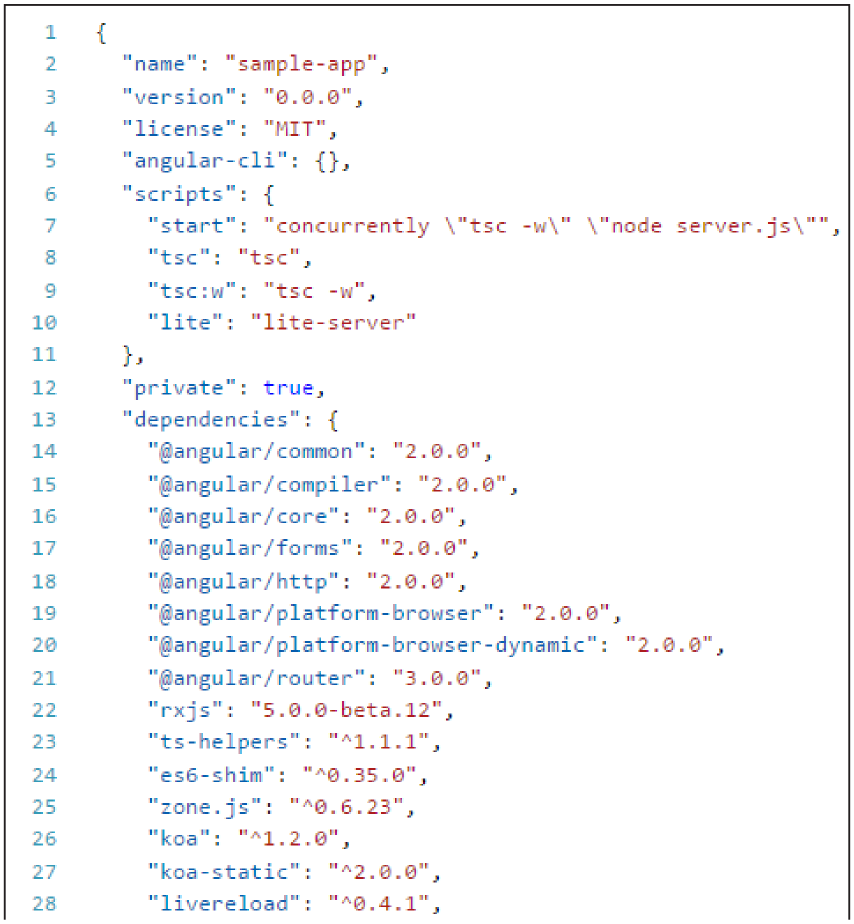
36

Angular 2: Simple Component Testing

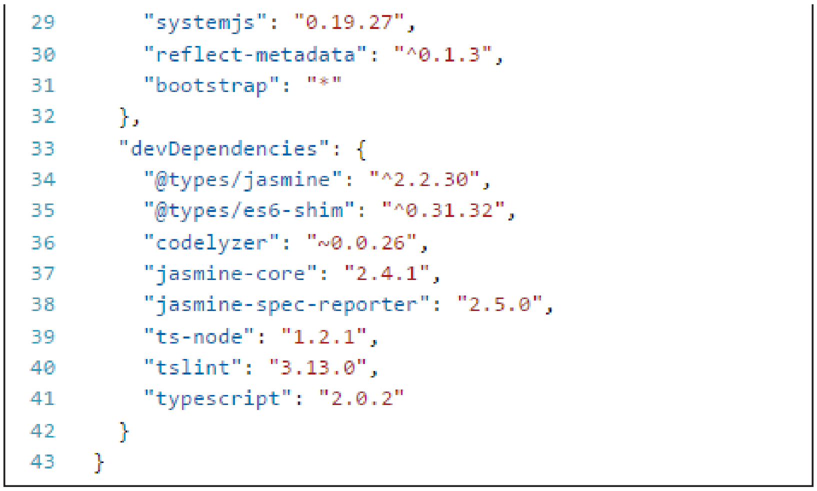
THIS CHAPTER'S FOCUS IS writing a unit test for an Angular 2 component using Jasmine. The chapter is implemented from scratch, for simplicity, using Visual Studio Code. (Note: The code for implementing server using koa is used as it is. See Chapter 5 for details.)

Our application needs package.json as shown in the following listing:

Listing 36.1: *package.json with all required packages for the application*



```
{
  "name": "sample-app",
  "version": "0.0.0",
  "license": "MIT",
  "angular-cli": {},
  "scripts": {
    "start": "concurrently \"tsc -w\" \"node server.js\"",
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "lite": "lite-server"
  },
  "private": true,
  "dependencies": {
    "@angular/common": "2.0.0",
    "@angular/compiler": "2.0.0",
    "@angular/core": "2.0.0",
    "@angular/forms": "2.0.0",
    "@angular/http": "2.0.0",
    "@angular/platform-browser": "2.0.0",
    "@angular/platform-browser-dynamic": "2.0.0",
    "@angular/router": "3.0.0",
    "rxjs": "5.0.0-beta.12",
    "ts-helpers": "^1.1.1",
    "es6-shim": "^0.35.0",
    "zone.js": "^0.6.23",
    "koa": "^1.2.0",
    "koa-static": "^2.0.0",
    "livereload": "^0.4.1",
```



```
    "systemjs": "0.19.27",
    "reflect-metadata": "^0.1.3",
    "bootstrap": "*"
  },
  "devDependencies": {
    "@types/jasmine": "^2.2.30",
    "@types/es6-shim": "^0.31.32",
    "codelyzer": "~0.0.26",
    "jasmine-core": "2.4.1",
    "jasmine-spec-reporter": "2.5.0",
    "ts-node": "1.2.1",
    "tslint": "3.13.0",
    "typescript": "2.0.2"
  }
}
```

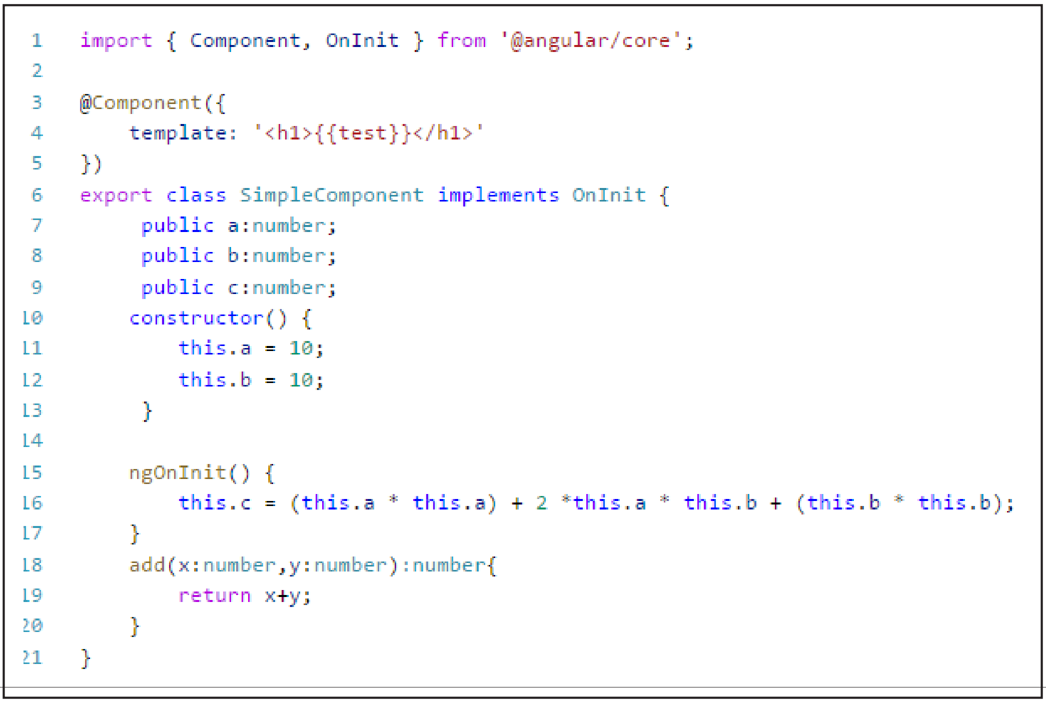
As shown above, the application requires the **Jasmine** Framework (2.4.1 as of this writing) for writing unit tests for the application. Jasmine is a behavior-driven framework for testing JavaScript code in web applications. This provides an easy way to write unit tests on the code without requiring any other third party framework/library and DOM.

All of the above packages must be installed using following command:

```
npm install
```

In the app.component.ts file of the **app** folder of the application, the component code is written as shown in Listing 36.2:

Listing 36.2: *The component code*



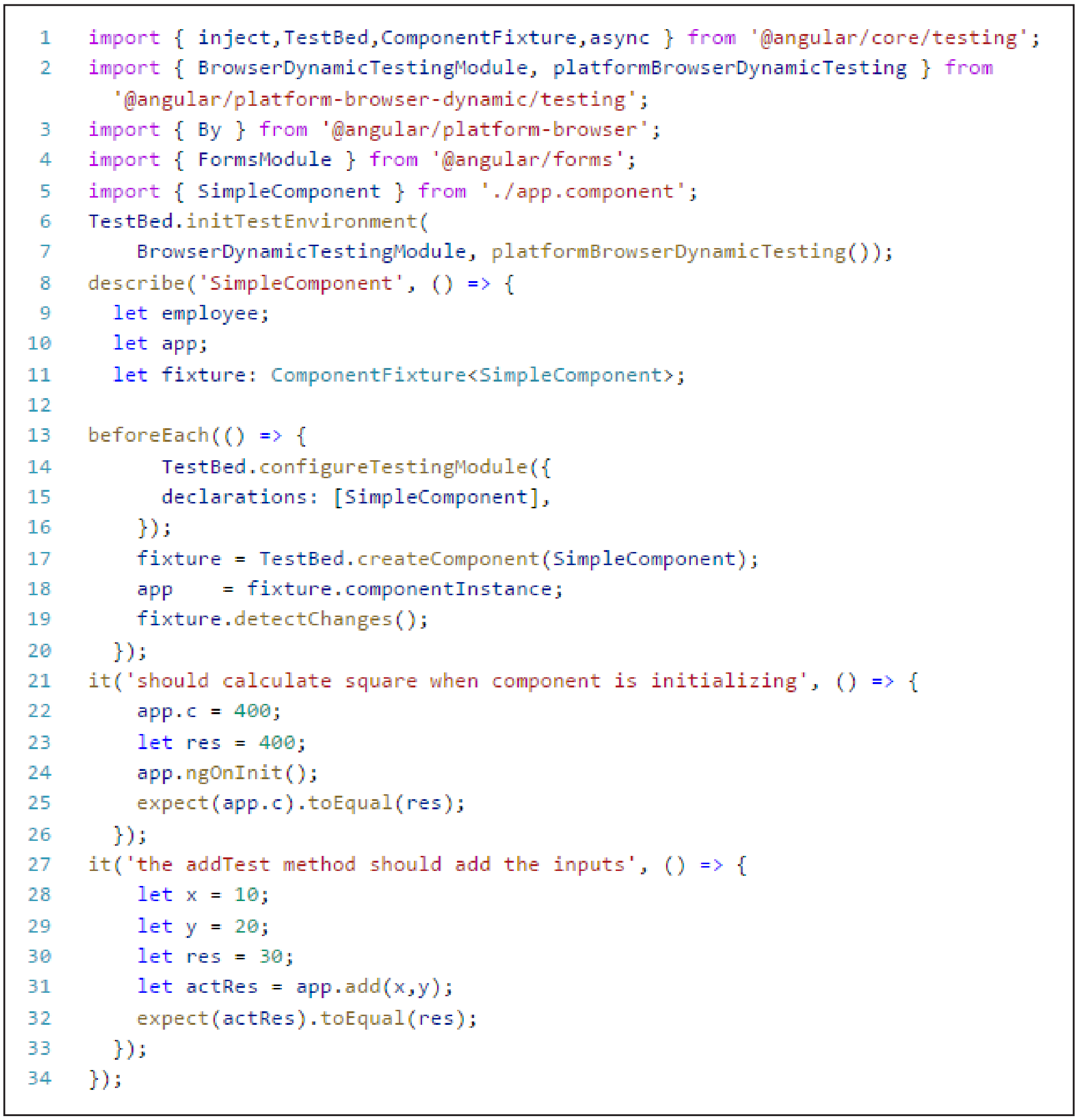
```
import { Component, OnInit } from '@angular/core';

@Component({
    template: '<h1>{{test}}</h1>'
})
export class SimpleComponent implements OnInit {
     public a:number;
     public b:number;
     public c:number;
    constructor() {
        this.a = 10;
        this.b = 10;
     }

    ngOnInit() {
        this.c = (this.a * this.a) + 2 *this.a * this.b + (this.b * this.b);
    }
    add(x:number,y:number):number{
        return x+y;
    }
}
```

The component seen in Listing 36.2 implements the OnInit interface and overrides the ngOnInit() method. The component defines public members a,b, and c (the values for a and b are set in the constructor). The ngOnInit() method contains logic for (a+b) square. The add() method performs simple addition of two numbers. The component defines template with inline HTML markup in it. This is an important step; if the component does not define a template, then the test will be unable to create an instance of the component.

The test must load required packages provided in the module @angular/core. In Angular 2, every module has its corresponding testing module. Our unit test implementation must refer these modules. Refer to Chapter 35, which lists the required modules to be loaded for testing.

The unit test logic is written in the app.component.spec.ts file as shown in Listing 36.3.

Listing 36.3: *The code for testing*



```
import { inject,TestBed,ComponentFixture,async } from '@angular/core/testing';
import { BrowserDynamicTestingModule, platformBrowserDynamicTesting } from
  '@angular/platform-browser-dynamic/testing';
import { By } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { SimpleComponent } from './app.component';
TestBed.initTestEnvironment(
    BrowserDynamicTestingModule, platformBrowserDynamicTesting());
describe('SimpleComponent', () => {
  let employee;
  let app;
  let fixture: ComponentFixture<SimpleComponent>;

beforeEach(() => {
      TestBed.configureTestingModule({
      declarations: [SimpleComponent],
    });
    fixture = TestBed.createComponent(SimpleComponent);
    app    = fixture.componentInstance;
    fixture.detectChanges();
  });
it('should calculate square when component is initializing', () => {
    app.c = 400;
    let res = 400;
    app.ngOnInit();
    expect(app.c).toEqual(res);
  });
it('the addTest method should add the inputs', () => {
    let x = 10;
    let y = 20;
    let res = 30;
    let actRes = app.add(x,y);
    expect(actRes).toEqual(res);
  });
});
```

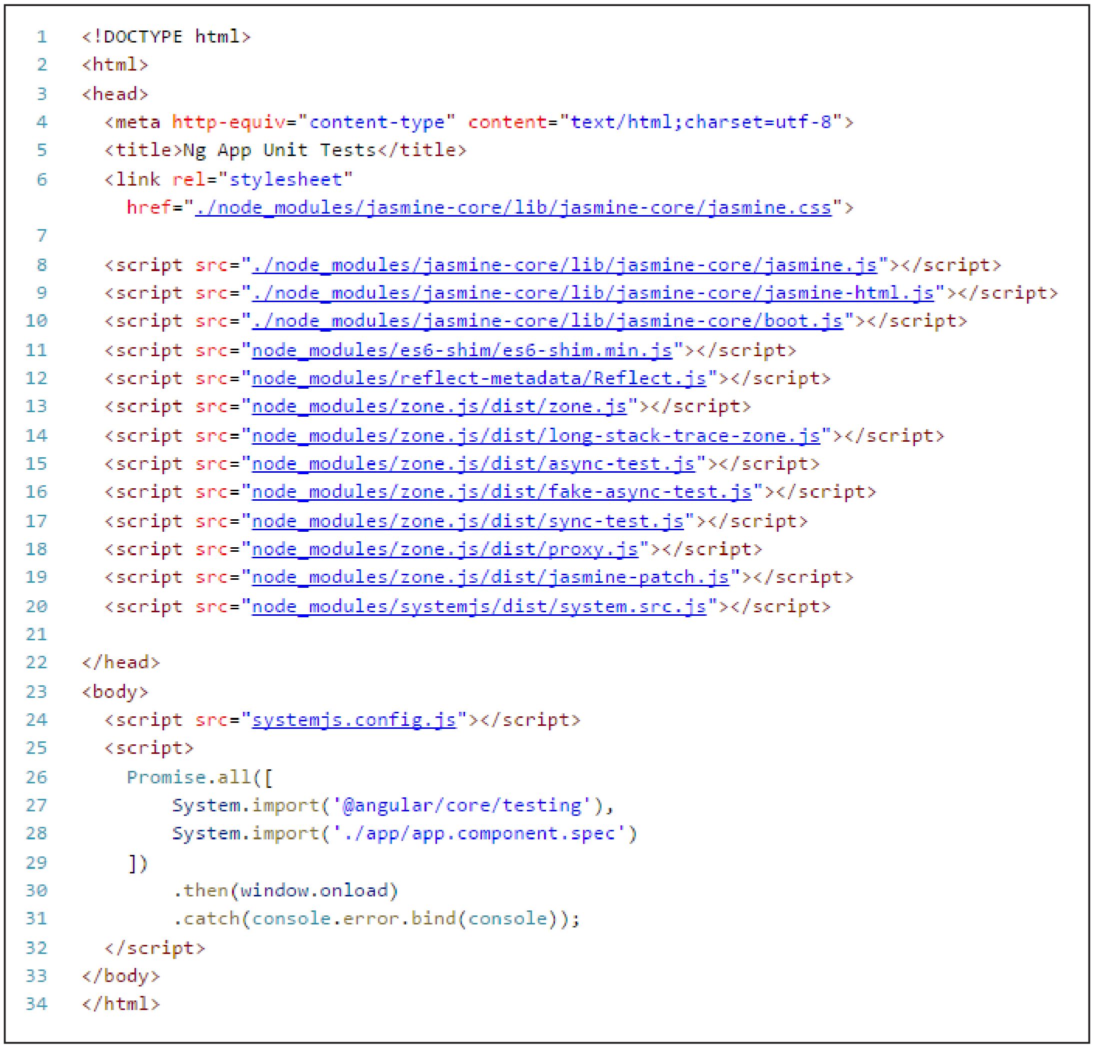
The code imports **TestBed** class from @angular/core/testing. This class is used for initializing the testing environment and providing a mechanism to initialize test environment using **initTestEnvironment()** function. This method uses **BrowserDynamicTestingModule** class and **platformBrowserDynamicTesting(),** which are used to load an instance of @NgModule created using **PlatformRef** class. The **configureTestingModule()** method of TestBed class is used to create a module for unit testing; it imports all providers, components, directives, and pipes required for testing. The **createComponent()** method of TestBed class is used to compile a component. It returns an object of generic ComponentFixture<ComponentType> type in which the component's instance is stored in the **componentInstance** property. The createComponent() method of the TestBed class returns ComponentFixture object.

Jasmine provides describe(), beforeEach(), and it() functions to implement the testing logic. The describe() method acts as an entry-point for testing where all declarations are placed. The beforeEach() method is used to configure testing modules and for importing all required dependencies. Once these dependencies are imported, the component will be complied. This will return ComponentFixture instance from which an actual instance of the component to be tested is retrieved.

Listing 36.3 contains two tests declared using the it() function: ngOnInitTest tests the ngOnInit() method of the SimpleComponent, whereas addTest tests the add() method of the same component.

To view the test results in the browser, the application contains simplecomponenttest.html file with the code and references as shown in Listing 36.4.

Listing 36.4: *The html code for test*



```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="content-type" content="text/html;charset=utf-8">
  <title>Ng App Unit Tests</title>
  <link rel="stylesheet"
    href="./node_modules/jasmine-core/lib/jasmine-core/jasmine.css">

  <script src="./node_modules/jasmine-core/lib/jasmine-core/jasmine.js"></script>
  <script src="./node_modules/jasmine-core/lib/jasmine-core/jasmine-html.js"></script>
  <script src="./node_modules/jasmine-core/lib/jasmine-core/boot.js"></script>
  <script src="node_modules/es6-shim/es6-shim.min.js"></script>
  <script src="node_modules/reflect-metadata/Reflect.js"></script>
  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/zone.js/dist/long-stack-trace-zone.js"></script>
  <script src="node_modules/zone.js/dist/async-test.js"></script>
  <script src="node_modules/zone.js/dist/fake-async-test.js"></script>
  <script src="node_modules/zone.js/dist/sync-test.js"></script>
  <script src="node_modules/zone.js/dist/proxy.js"></script>
  <script src="node_modules/zone.js/dist/jasmine-patch.js"></script>
  <script src="node_modules/systemjs/dist/system.src.js"></script>

</head>
<body>
  <script src="systemjs.config.js"></script>
  <script>
    Promise.all([
        System.import('@angular/core/testing'),
        System.import('./app/app.component.spec')
    ])
        .then(window.onload)
        .catch(console.error.bind(console));
  </script>
</body>
</html>
```

Listing 36.4 shows all required references for Html testing of the component. This imports @angular/core/testing and app.component.spec modules for execution. The main.ts file contains the application module which is shown in listing 36.5.

Listing 36.5: *The main.ts code*

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';
import{SimpleComponent} from './app.component';
 @NgModule({
   imports: [BrowserModule,FormsModule],
  declarations: [SimpleComponent],
  bootstrap: [SimpleComponent]
})
export class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

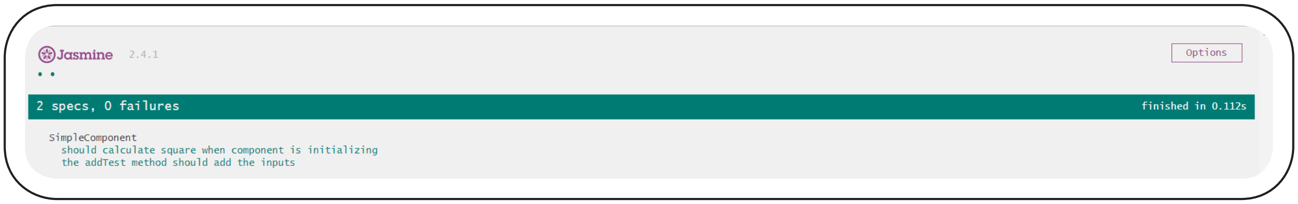
Use the following command to run the server

```
npm run start
```

The test result can be viewed by entering following URL in the browser

http://localhost:3000/simplecomponenttest.html

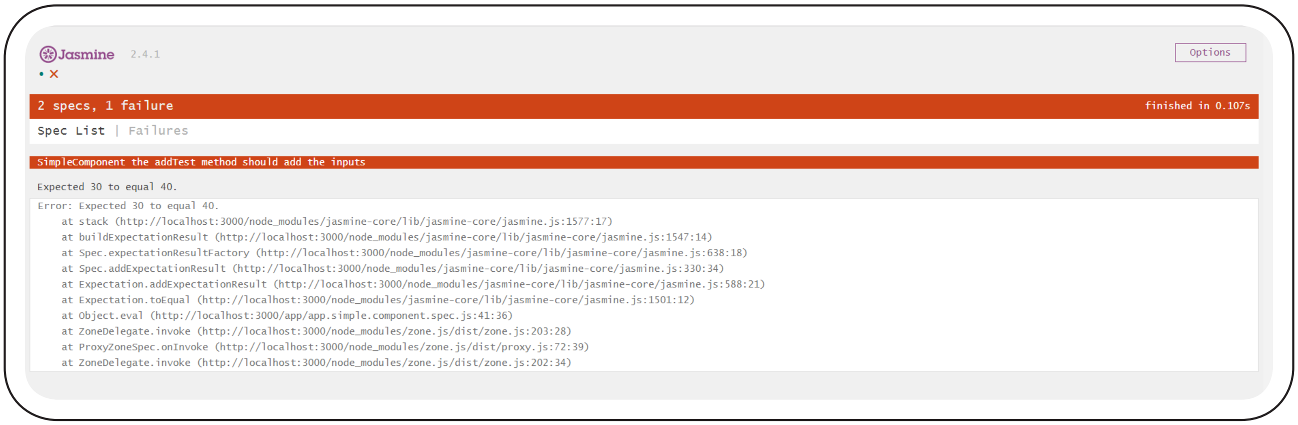
The test result will be shown as displayed in Figure 36.1



```
2 specs, 0 failures                                    finished in 0.112s

SimpleComponent
  should calculate square when component is initializing
  the addTest method should add the inputs
```

Figure 36.1: *Test Result*

The result shows both tests are successful. In **addTest** if the res variable value is changed from 30 to 40, then this test will fail as shown in Figure 36.2.



```
2 specs, 1 failure                                     finished in 0.107s

SimpleComponent the addTest method should add the inputs

Expected 30 to equal 40.
Error: Expected 30 to equal 40.
    at stack (http://localhost:3000/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:1577:17)
    at buildExpectationResult (http://localhost:3000/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:1547:14)
    at Spec.expectationResultFactory (http://localhost:3000/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:638:18)
    at Spec.addExpectationResult (http://localhost:3000/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:330:34)
    at Expectation.addExpectationResult (http://localhost:3000/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:588:21)
    at Expectation.toEqual (http://localhost:3000/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:1501:12)
    at Object.eval (http://localhost:3000/app/app.simple.component.spec.js:41:36)
    at ZoneDelegate.invoke (http://localhost:3000/node_modules/zone.js/dist/zone.js:203:28)
    at ProxyZoneSpec.onInvoke (http://localhost:3000/node_modules/zone.js/dist/proxy.js:72:39)
    at ZoneDelegate.invoke (http://localhost:3000/node_modules/zone.js/dist/zone.js:202:34)
```

Figure 36.2: *The failed Test*

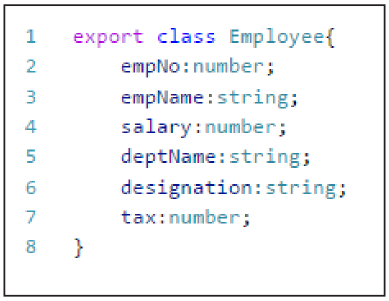
TESTING COMPONENT HAVING HTML TEMPLATE WITH DATABINDING

After completing the process of testing a simple component, what happens if the component has an Html template with two-way databinding? In that case, the test should be capable of parsing the Html template having databinding expressions in it. Chapters from 16 to 20 discussed Databinding and the use of **FormsModule** to execute **ngModel** property. If the component contains HTML template that uses the directive ngModel, the testing module must import the FormsModule as well.

To implement Component Testing with HTML template, create a new application with all the steps as in in listings 36-1, 36-3, and 36-5.

In the application, the Employee model class is created in employee.model.ts file as shown in Listing 36.6.

Listing 36.6: *Employee model class*

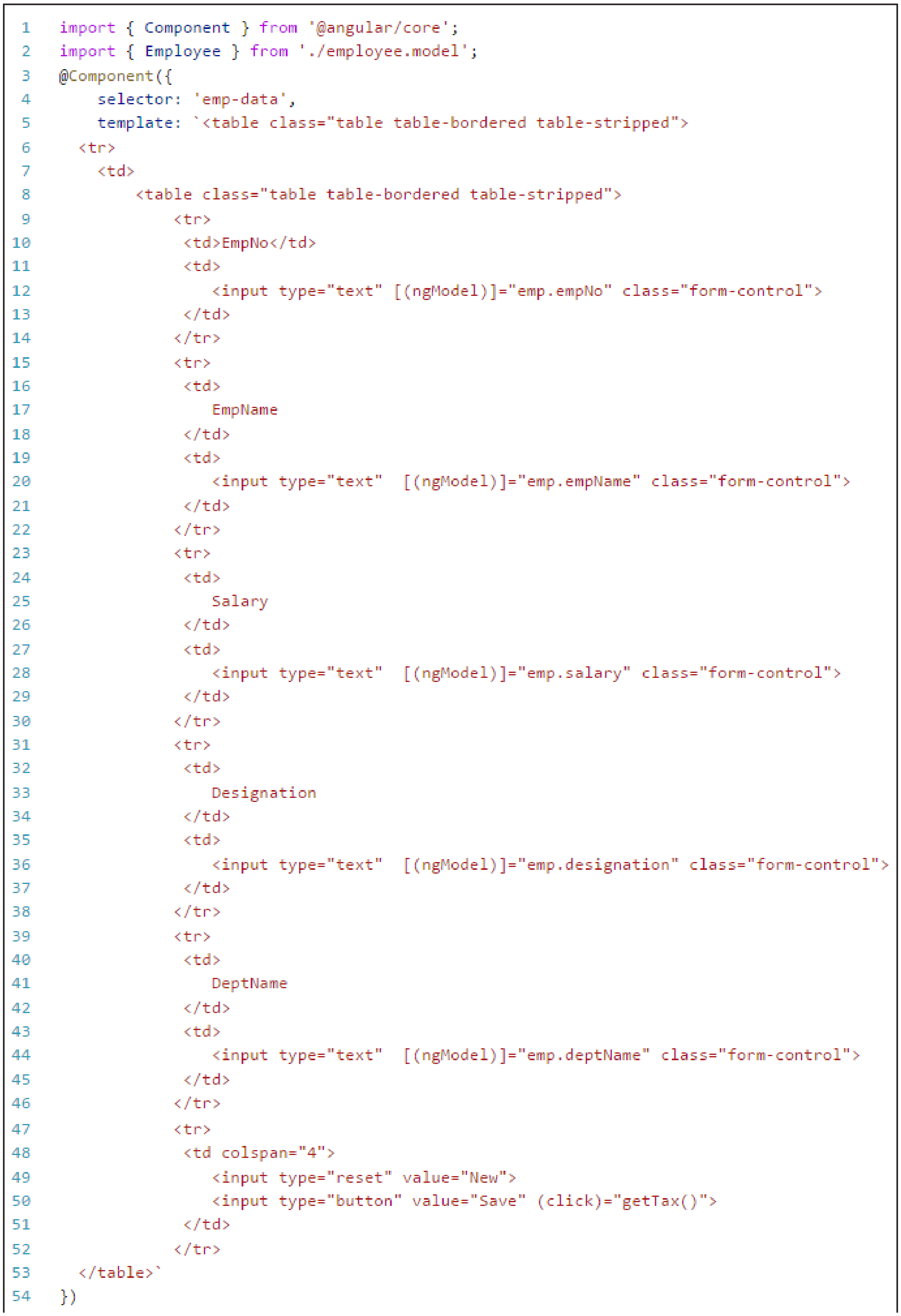


```
export class Employee{
    empNo:number;
    empName:string;
    salary:number;
    deptName:string;
    designation:string;
    tax:number;
}
```

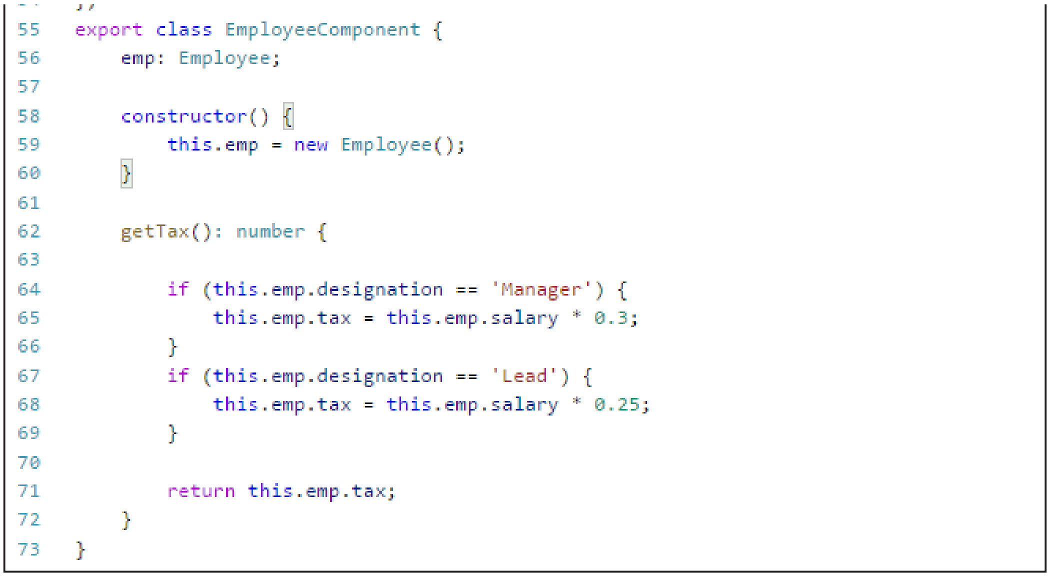
The Employee model class will be used to accept Employee details.

The app.component.ts file contains the component as shown in Listing 36.7.

Listing 36.7: *Component with Html template having Databinding*



```
import { Component } from '@angular/core';
import { Employee } from './employee.model';
@Component({
    selector: 'emp-data',
    template: `<table class="table table-bordered table-stripped">
  <tr>
    <td>
        <table class="table table-bordered table-stripped">
            <tr>
             <td>EmpNo</td>
             <td>
                <input type="text" [(ngModel)]="emp.empNo" class="form-control">
             </td>
            </tr>
            <tr>
             <td>
                EmpName
             </td>
             <td>
                <input type="text"  [(ngModel)]="emp.empName" class="form-control">
             </td>
            </tr>
            <tr>
             <td>
                Salary
             </td>
             <td>
                <input type="text"  [(ngModel)]="emp.salary" class="form-control">
             </td>
            </tr>
            <tr>
             <td>
                Designation
             </td>
             <td>
                <input type="text"  [(ngModel)]="emp.designation" class="form-control">
             </td>
            </tr>
            <tr>
             <td>
                DeptName
             </td>
             <td>
                <input type="text"  [(ngModel)]="emp.deptName" class="form-control">
             </td>
            </tr>
            <tr>
             <td colspan="4">
                <input type="reset" value="New">
                <input type="button" value="Save" (click)="getTax()">
             </td>
            </tr>
  </table>`
})
```

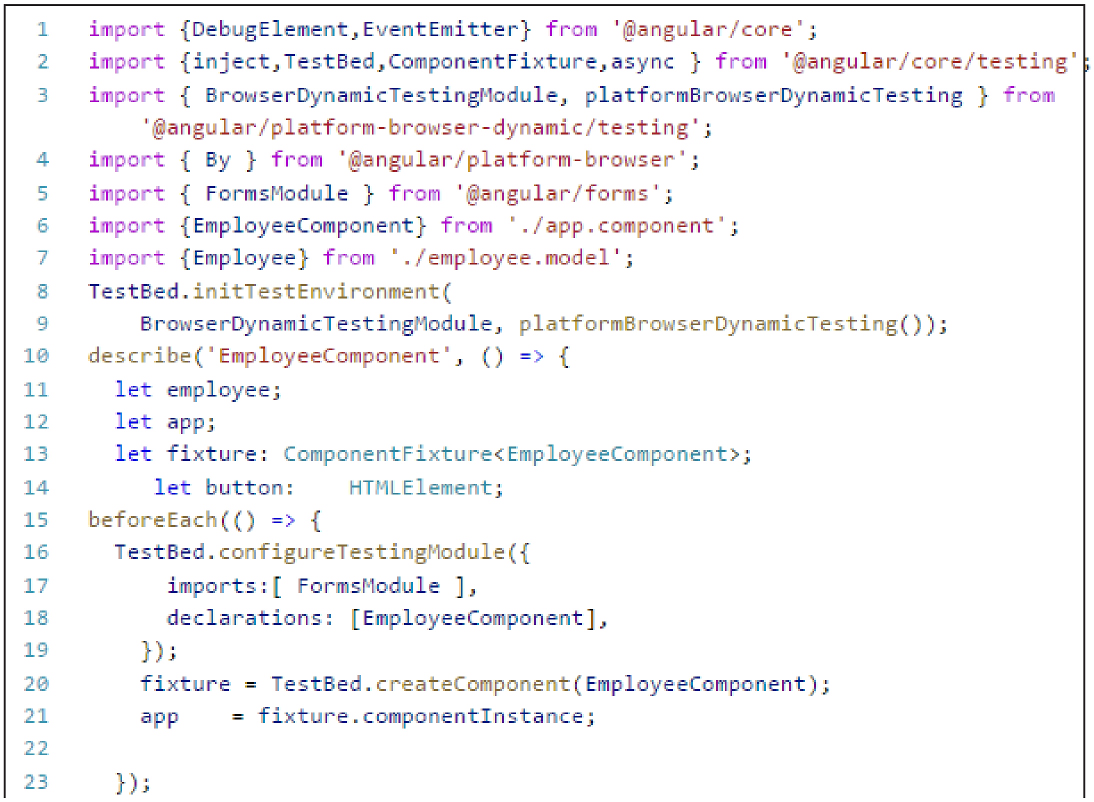


```
export class EmployeeComponent {
    emp: Employee;

    constructor() {
        this.emp = new Employee();
    }

    getTax(): number {

        if (this.emp.designation == 'Manager') {
            this.emp.tax = this.emp.salary * 0.3;
        }
        if (this.emp.designation == 'Lead') {
            this.emp.tax = this.emp.salary * 0.25;
        }

        return this.emp.tax;
    }
}
```

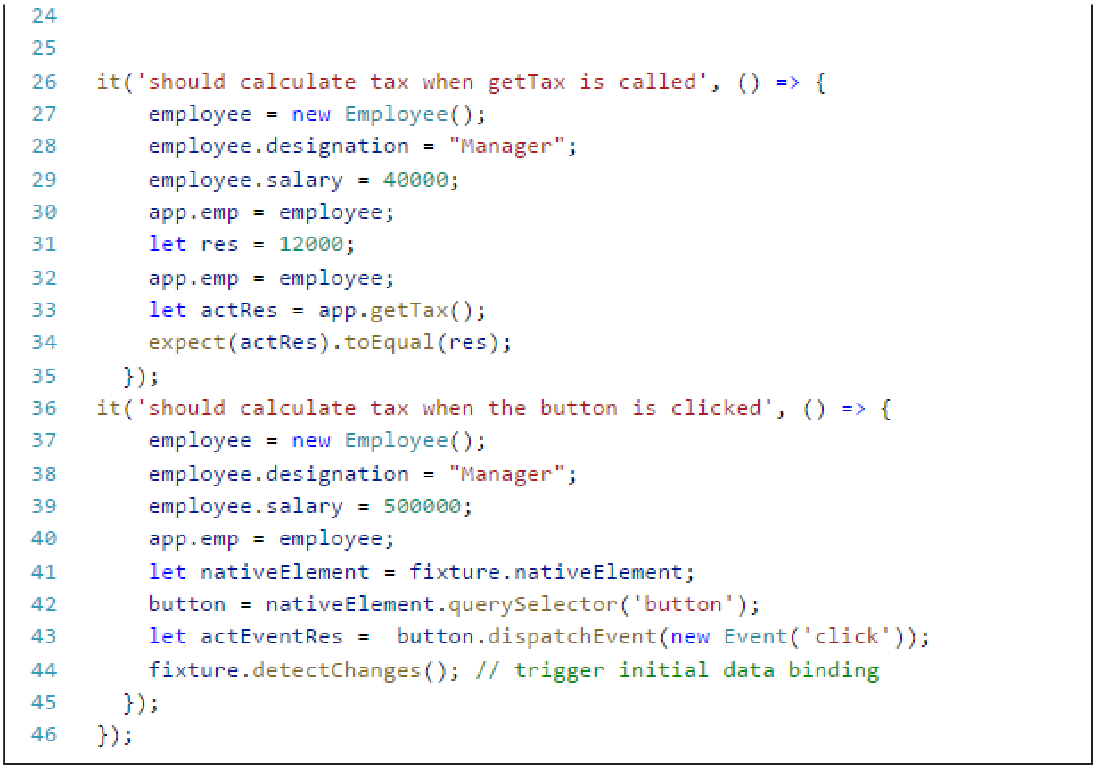
This listing defines EmployeeComponent with an inline Html template. The template contains <input> elements which are bound with the properties of Employee model class using **ngModel.** The **getTax()** method calculates tax of the employee based on the designation.

The app.component.spec.ts file contains the logic for testing as shown in Listing 36.8:

Listing 36.8: *The component spec code for testing environment*



```
import {DebugElement,EventEmitter} from '@angular/core';
import {inject,TestBed,ComponentFixture,async } from '@angular/core/testing';
import { BrowserDynamicTestingModule, platformBrowserDynamicTesting } from
    '@angular/platform-browser-dynamic/testing';
import { By } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import {EmployeeComponent} from './app.component';
import {Employee} from './employee.model';
TestBed.initTestEnvironment(
    BrowserDynamicTestingModule, platformBrowserDynamicTesting());
describe('EmployeeComponent', () => {
  let employee;
  let app;
  let fixture: ComponentFixture<EmployeeComponent>;
     let button:    HTMLElement;
beforeEach(() => {
  TestBed.configureTestingModule({
      imports:[ FormsModule ],
      declarations: [EmployeeComponent],
    });
    fixture = TestBed.createComponent(EmployeeComponent);
    app    = fixture.componentInstance;

  });
```



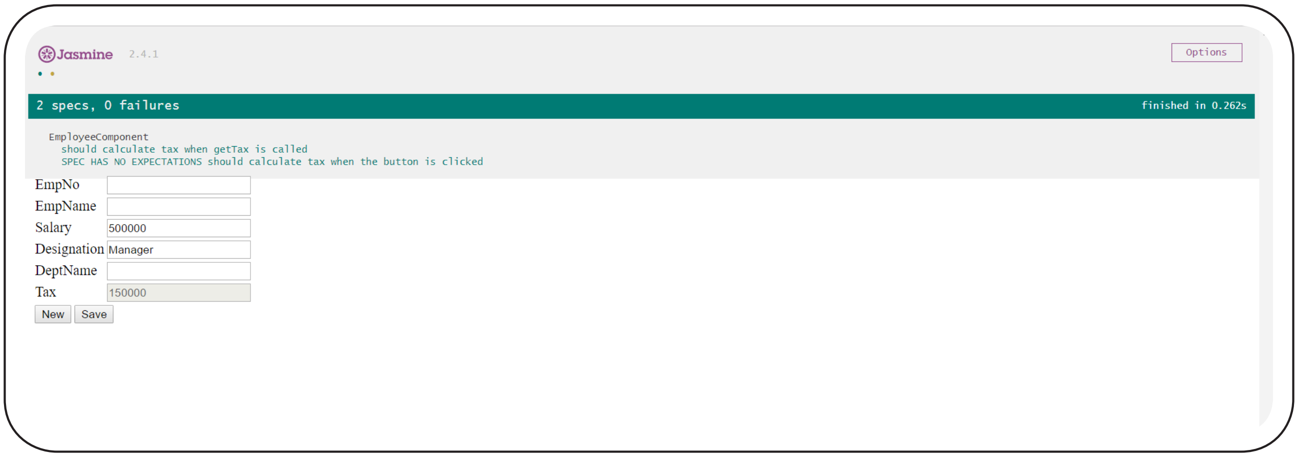
```


  it('should calculate tax when getTax is called', () => {
      employee = new Employee();
      employee.designation = "Manager";
      employee.salary = 40000;
      app.emp = employee;
      let res = 12000;
      app.emp = employee;
      let actRes = app.getTax();
      expect(actRes).toEqual(res);
    });
  it('should calculate tax when the button is clicked', () => {
      employee = new Employee();
      employee.designation = "Manager";
      employee.salary = 500000;
      app.emp = employee;
      let nativeElement = fixture.nativeElement;
      button = nativeElement.querySelector('button');
      let actEventRes =  button.dispatchEvent(new Event('click'));
      fixture.detectChanges(); // trigger initial data binding
    });
  });
```

As seen in Listing 36-4, the code for Testing environment using TestBed is similar except that the **configureTestingModule()** method has an additional line for importing **FormModule.** The test getTax implements code for testing.

The test **getTaxClickEvent** sets salary and designation values for Employee object to calculate tax. This test uses **nativeElement** property of the ComponentFixture class to read the Button element from the HTML template using the querySelector() method. querySelector returns an instance of the DOM element (in this case button). Using the **dispatchEvent()** method, the **click** event of the button can be raised using code in the test. Since the click event is bound with the **getTax()** function, it can be invoked and executed using the detectChanges() function of the ComponentFixture class.

This is how the HTML template from the Component with its event can be tested.

The componentTest.html file is used for Html testing; this file will have a similar code to in listing 36-5. When this file is viewed in browser using the URL *http://localhost:3000/componentTest.html,* the following result will be displayed.

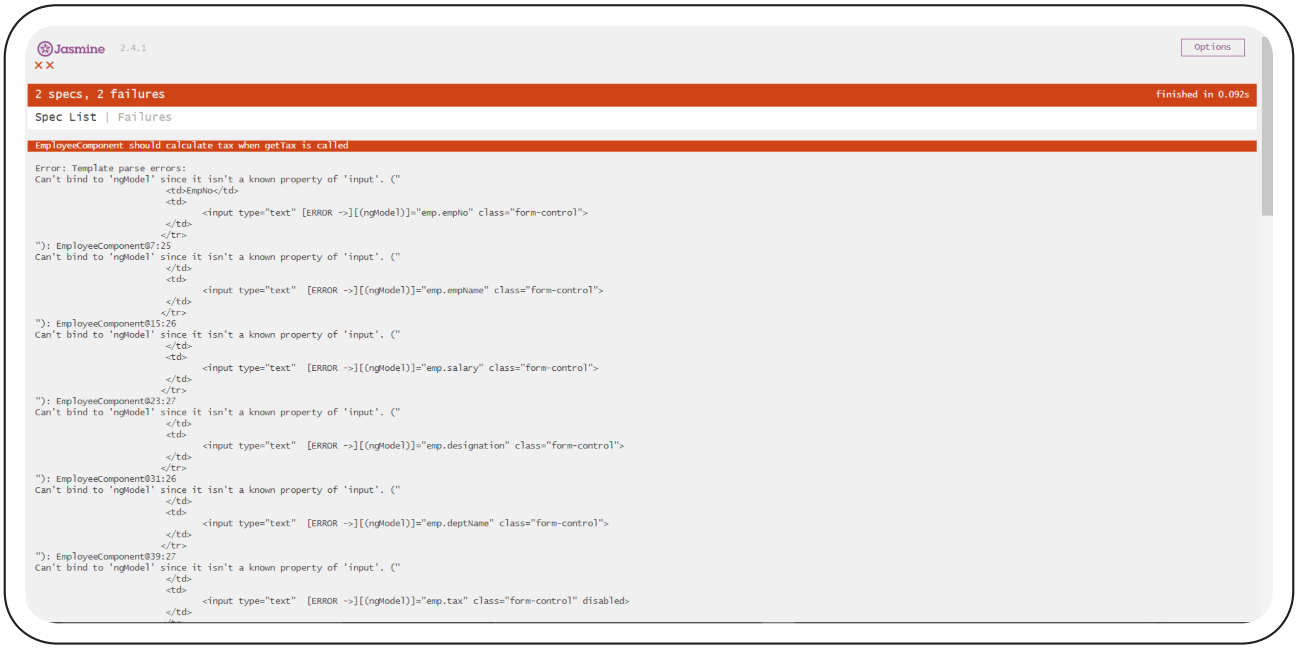


```
2 specs, 0 failures                                                     finished in 0.262s

  EmployeeComponent
    should calculate tax when getTax is called
    SPEC HAS NO EXPECTATIONS should calculate tax when the button is clicked
```

EmpNo
EmpName
Salary 500000
Designation Manager
DeptName
Tax 150000

***Figure 36.3:** The success Test for Component with Html template*

If the configureTestingModule method does not import **FormsModule,** then the following result will be displayed:



```
2 specs, 2 failures                                                     finished in 0.092s

EmployeeComponent should calculate tax when getTax is called

Error: Template parse errors:
Can't bind to 'ngModel' since it isn't a known property of 'input'. ("
                        <td>EmpNo</td>
                        <td>
                              <input type="text" [ERROR ->][(ngModel)]="emp.empNo" class="form-control">
                        </td>
                       </tr>
"): EmployeeComponent@7:25
Can't bind to 'ngModel' since it isn't a known property of 'input'. ("
                        </td>
                        <td>
                              <input type="text"  [ERROR ->][(ngModel)]="emp.empName" class="form-control">
                        </td>
                       </tr>
"): EmployeeComponent@15:26
Can't bind to 'ngModel' since it isn't a known property of 'input'. ("
                        </td>
                        <td>
                              <input type="text"  [ERROR ->][(ngModel)]="emp.salary" class="form-control">
                        </td>
                       </tr>
"): EmployeeComponent@23:27
Can't bind to 'ngModel' since it isn't a known property of 'input'. ("
                        </td>
                        <td>
                              <input type="text"  [ERROR ->][(ngModel)]="emp.designation" class="form-control">
                        </td>
                       </tr>
"): EmployeeComponent@31:26
Can't bind to 'ngModel' since it isn't a known property of 'input'. ("
                        </td>
                        <td>
                              <input type="text"  [ERROR ->][(ngModel)]="emp.deptName" class="form-control">
                        </td>
                       </tr>
"): EmployeeComponent@39:27
Can't bind to 'ngModel' since it isn't a known property of 'input'. ("
                        </td>
                        <td>
                              <input type="text"  [ERROR ->][(ngModel)]="emp.tax" class="form-control" disabled>
                        </td>
```

***Figure 36.4:** The failed test as a result of no import of FormsModule*

The Test result clearly specifies that there are **Template parse errors,** and it is not able to parse **ngModel** property**.**

Conclusion

The components are the basic building blocks of any Angular application and they contain the logic that controls the way an application looks. So, it is important to make sure that the component is tested for its logic. This chapter gave a basic overview of getting started with testing the components.

Chapter 37

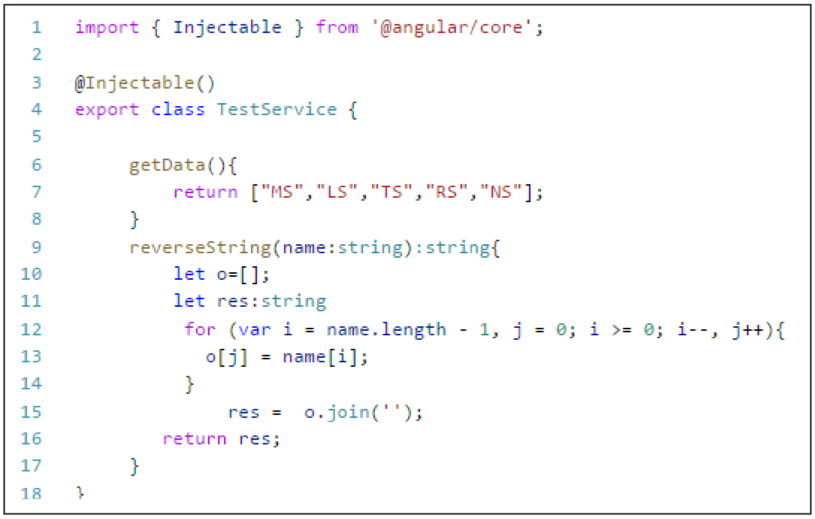
Testing Angular 2 Service

CHAPTER 36 FOCUSED ON the use of Jasmine for testing Angular 2 components and HTML template with an event. This chapter will examine the implementation of Service Testing in isolation, and Component Testing having service dependency.

A component forms a major building block of Angular 2 applications. It is the component which makes call to all external Web API/REST Services using an Angular 2 Service. In actual practice, the Angular 2 Service is not limited to making REST calls. It can also contain the utility logic, which is not preferred to be written in an Angular 2 Component. The Service is injected into the Component using Dependency Injection.

Chapter 25 showed how to create an Angular 2 Service. This chapter will use a new service. Listing 37.1 shows the code of TestService in the testservice.ts file.

Listing 37.1: *The TestService code*



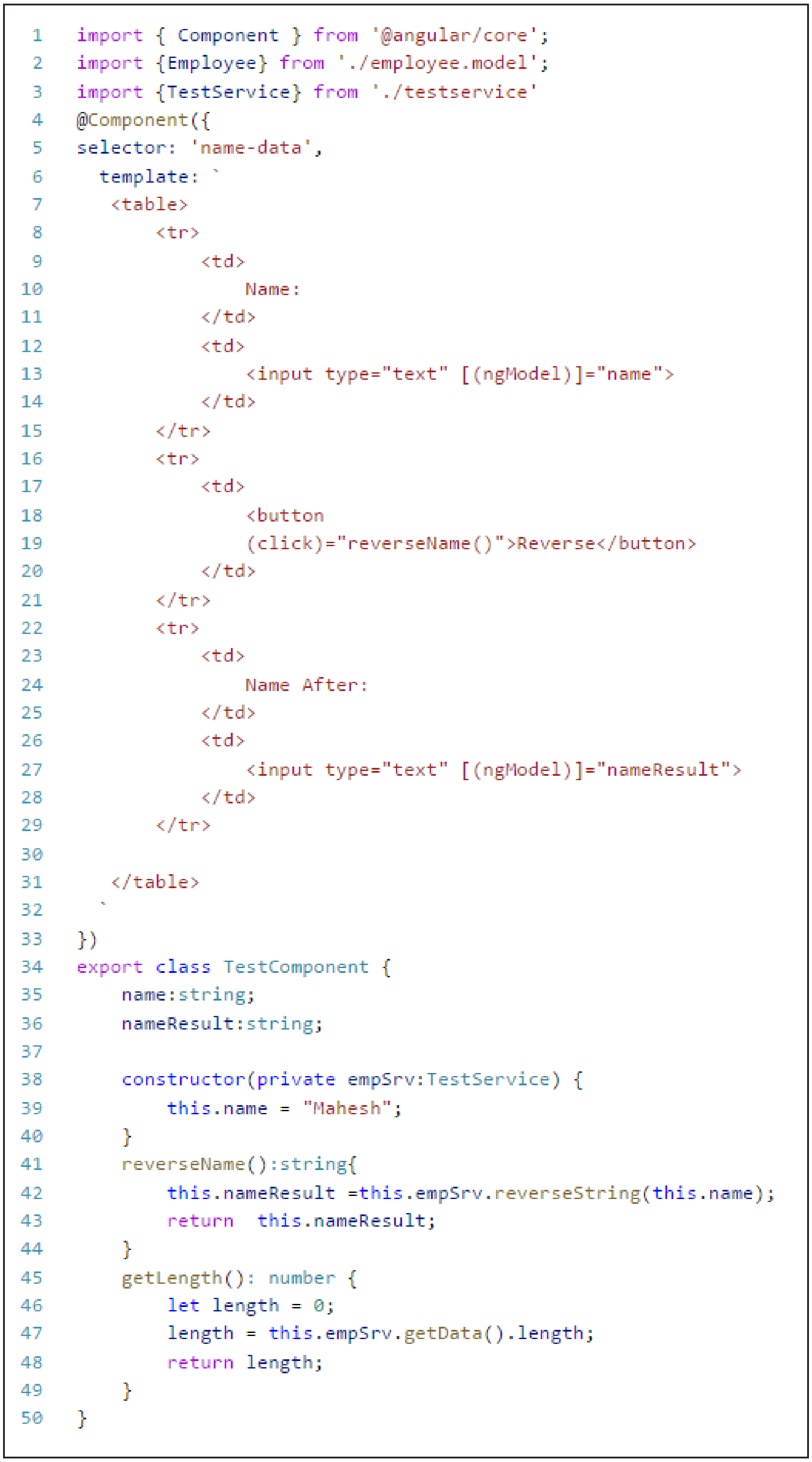
```
import { Injectable } from '@angular/core';

@Injectable()
export class TestService {

     getData(){
         return ["MS","LS","TS","RS","NS"];
     }
     reverseString(name:string):string{
         let o=[];
         let res:string
          for (var i = name.length - 1, j = 0; i >= 0; i--, j++){
            o[j] = name[i];
          }
              res =  o.join('');
        return res;
     }
}
```

This code contains two methods:

- getData() returns an array of strings.
- reverseString() accepts a string input argument and reverses it.

The TestService is injected in an Angular 2 component of the name TestComponent of the app.component.ts file.

Listing 37.2: *The Component class*

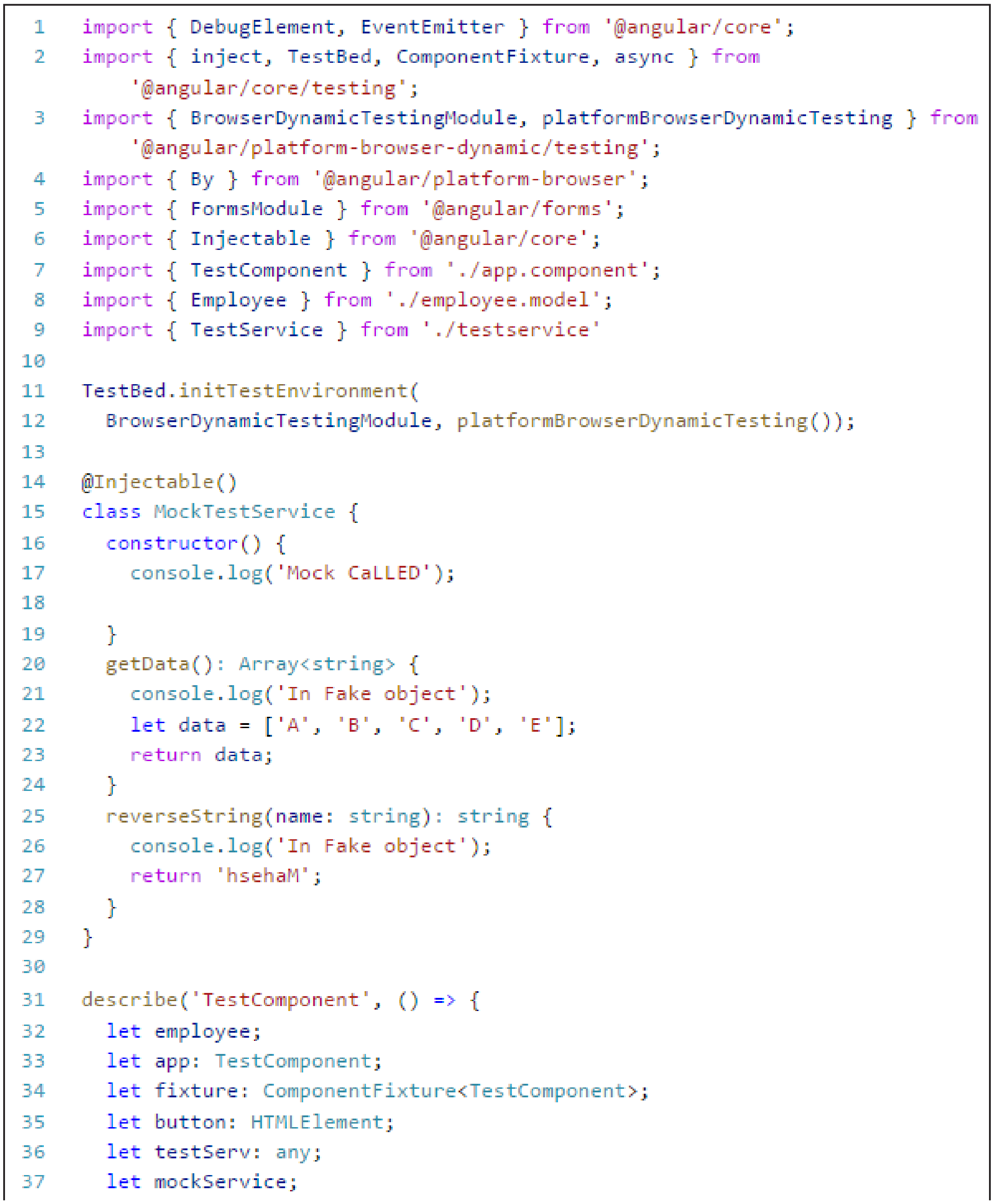


```
import { Component } from '@angular/core';
import {Employee} from './employee.model';
import {TestService} from './testservice'
@Component({
selector: 'name-data',
  template: `
   <table>
       <tr>
           <td>
               Name:
           </td>
           <td>
               <input type="text" [(ngModel)]="name">
           </td>
       </tr>
       <tr>
           <td>
               <button
               (click)="reverseName()">Reverse</button>
           </td>
       </tr>
       <tr>
           <td>
               Name After:
           </td>
           <td>
               <input type="text" [(ngModel)]="nameResult">
           </td>
       </tr>

   </table>
  `
})
export class TestComponent {
    name:string;
    nameResult:string;

    constructor(private empSrv:TestService) {
        this.name = "Mahesh";
    }
    reverseName():string{
        this.nameResult =this.empSrv.reverseString(this.name);
        return  this.nameResult;
    }
    getLength(): number {
        let length = 0;
        length = this.empSrv.getData().length;
        return length;
    }
}
```

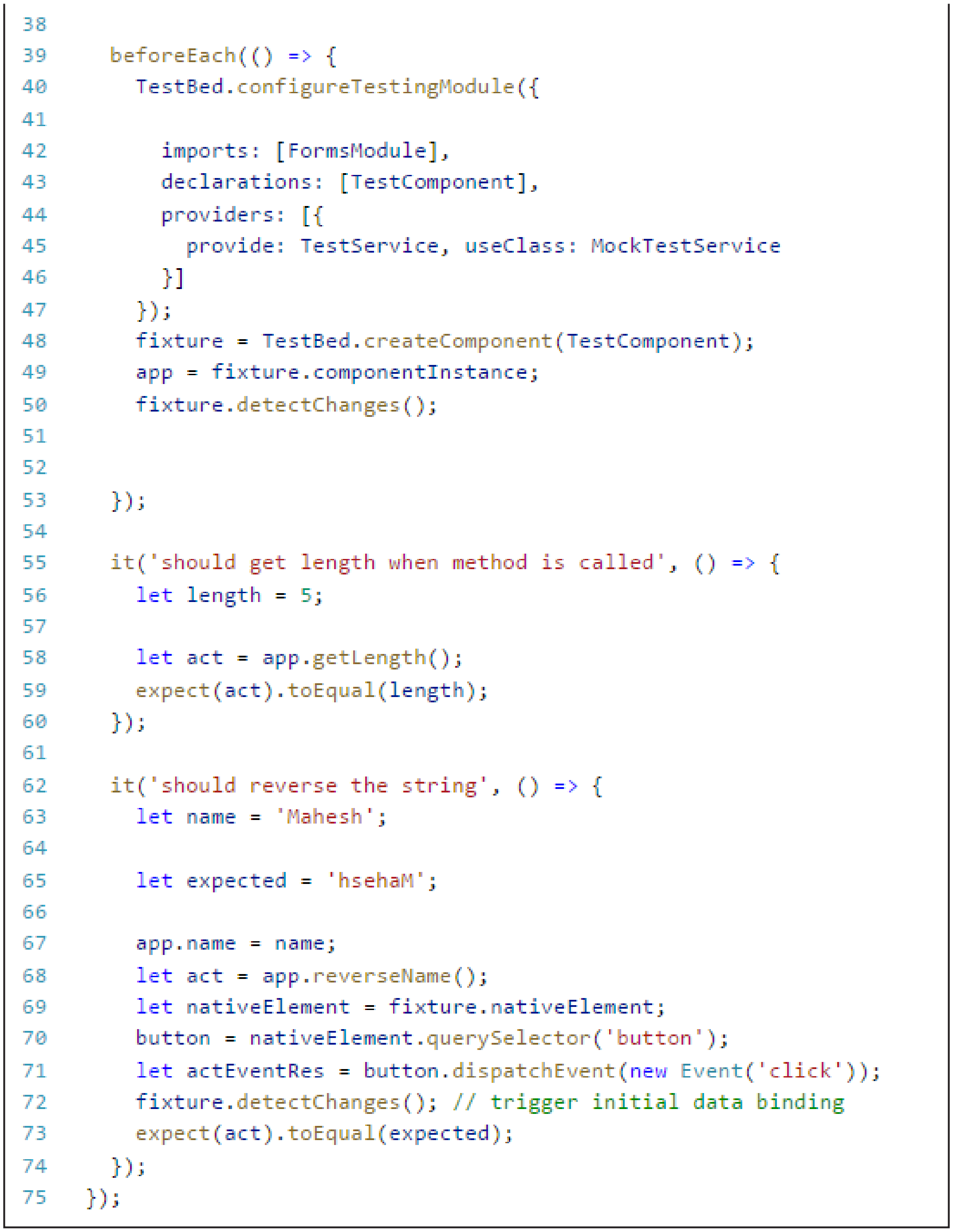
The component is injected with TestService using constructor injection.

The Test for this component is implemented using app.component.spec.ts file as shown in Listing 37.3.

Listing 37.3: *The component testing code*



```
import { DebugElement, EventEmitter } from '@angular/core';
import { inject, TestBed, ComponentFixture, async } from
    '@angular/core/testing';
import { BrowserDynamicTestingModule, platformBrowserDynamicTesting } from
    '@angular/platform-browser-dynamic/testing';
import { By } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { Injectable } from '@angular/core';
import { TestComponent } from './app.component';
import { Employee } from './employee.model';
import { TestService } from './testservice'

TestBed.initTestEnvironment(
  BrowserDynamicTestingModule, platformBrowserDynamicTesting());

@Injectable()
class MockTestService {
  constructor() {
    console.log('Mock CaLLED');

  }
  getData(): Array<string> {
    console.log('In Fake object');
    let data = ['A', 'B', 'C', 'D', 'E'];
    return data;
  }
  reverseString(name: string): string {
    console.log('In Fake object');
    return 'hsehaM';
  }
}

describe('TestComponent', () => {
  let employee;
  let app: TestComponent;
  let fixture: ComponentFixture<TestComponent>;
  let button: HTMLElement;
  let testServ: any;
  let mockService;
```



```

  beforeEach(() => {
    TestBed.configureTestingModule({

      imports: [FormsModule],
      declarations: [TestComponent],
      providers: [{
        provide: TestService, useClass: MockTestService
      }]
    });
    fixture = TestBed.createComponent(TestComponent);
    app = fixture.componentInstance;
    fixture.detectChanges();


  });

  it('should get length when method is called', () => {
    let length = 5;

    let act = app.getLength();
    expect(act).toEqual(length);
  });

  it('should reverse the string', () => {
    let name = 'Mahesh';

    let expected = 'hsehaM';

    app.name = name;
    let act = app.reverseName();
    let nativeElement = fixture.nativeElement;
    button = nativeElement.querySelector('button');
    let actEventRes = button.dispatchEvent(new Event('click'));
    fixture.detectChanges(); // trigger initial data binding
    expect(act).toEqual(expected);
  });
});
```

Since the component is dependent on the TestService, the MockTestService class is created using the same signature as TestService. This mock class is used to mock calls from component to the service. The configureTestingModule() method of the TestBed class accepts the provider's parameter, which is set to the TestService and the useClass as MockTestService. Then, the mock object will be used while methods from the component are called. Listing 37.4 shows the TestBed configuration.

Listing 37.4: *TestService passed to TestBed.configureTestingModule and Mock object(highlighted)*

```
TestBed.configureTestingModule({
      imports:[ FormsModule ],
      declarations: [TestComponent],
      providers:    [ {provide: TestService, useClass: MockTestService} ]
});
```

Observe the tests for getLength() and reverseName() method of the component class.

To execute the test, the project is added with servecomponent.html. This has code similar to the one in Chapter 36 for test html page.

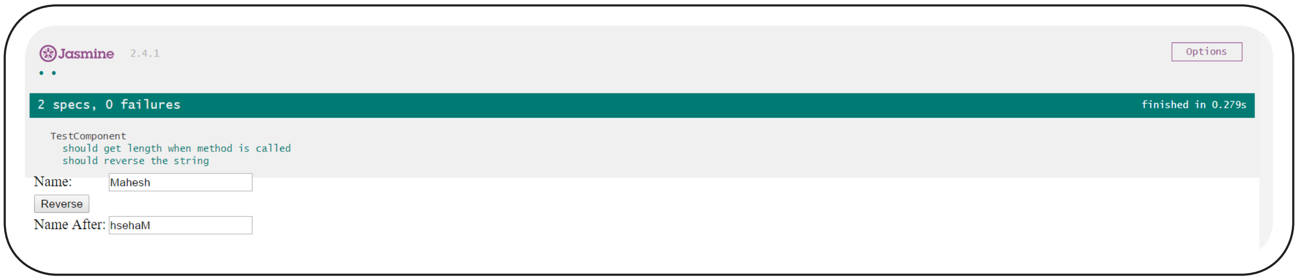
Run the application using following command:

npm run start

The html test can be seen using the following URL:

http://localhost:3000/ServComponent.html

The result will be as follows:



```
2 specs, 0 failures                                        finished in 0.279s

  TestComponent
    should get length when method is called
    should reverse the string
```

Name: Mahesh

Name After: hsehaM

Figure 37.1: *The test result*

It's important to note here that if the component is dependent on the service, then the test must be provided with the service.

In Angular 2, it is important to write tests for isolated blocks of the application. It's better that the Angular 2 service is directly tested without writing tests for the component. The advantage of this approach are: the test has more focus on each isolated component, it is more productive, and chances of logical errors are reduced.

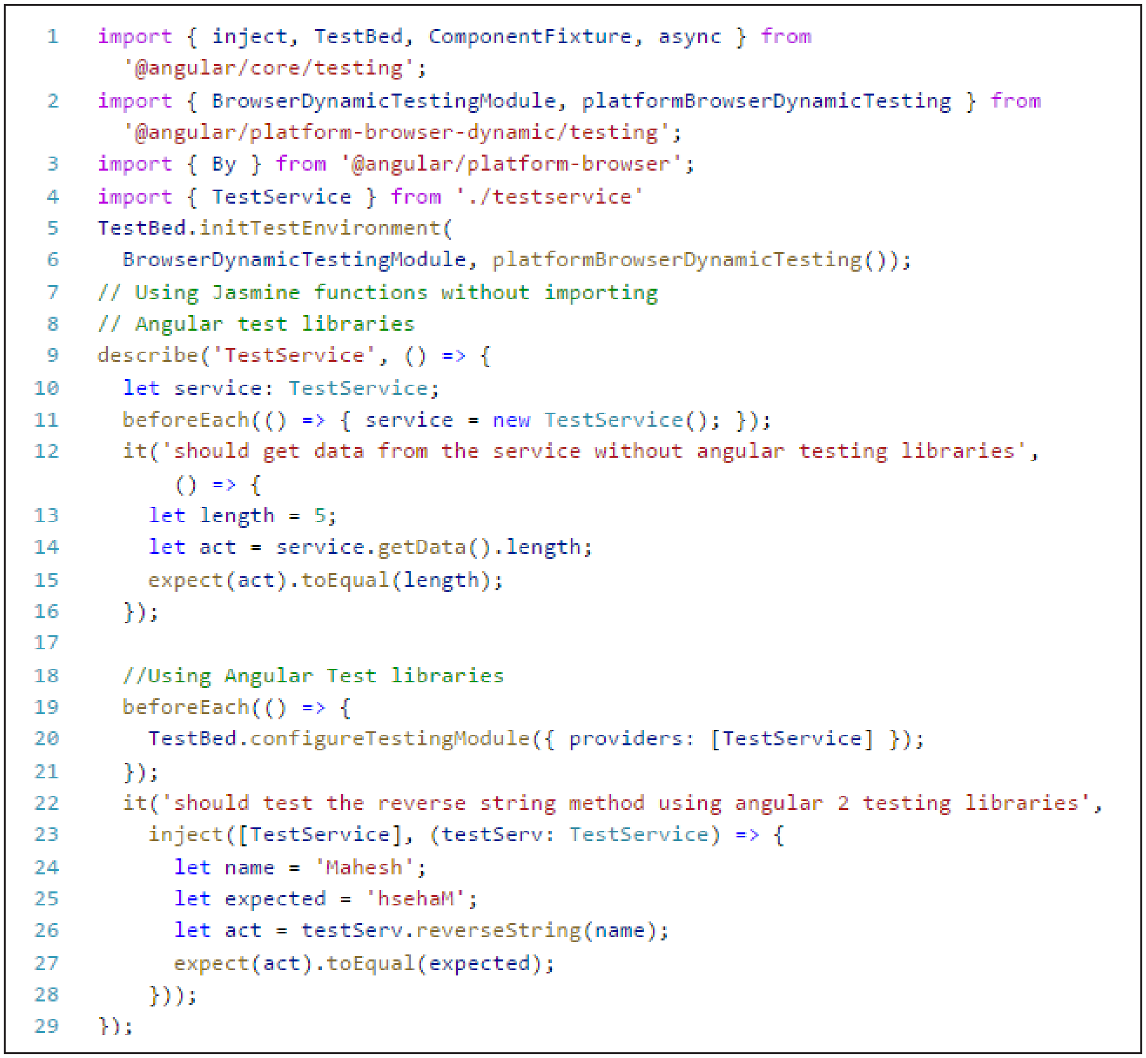
Isolated Service testing

Angular 2 Service can be used to contain utilities methods, as well as external calls using Http dependencies. In such cases, separate tests can be written on the service.

This chapter discusses isolated service testing without having any dependency in it. Service testing with Http dependency will be discussed in Chapter 38.

A service can be directly tested using Jasmine framework and without using an Angular 2 testing library. It is an individual's choice to decide whether to use Angular 2 testing library or not. The code in the following listing implements test for the TestService methods—with and without Angular 2 testing library.

Listing 37.5: *The testservice.spec.ts file with the service testing code*



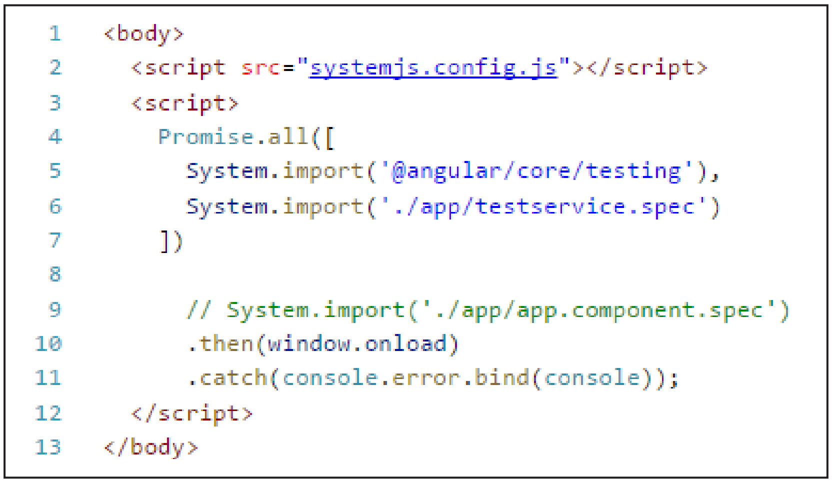
```
import { inject, TestBed, ComponentFixture, async } from
  '@angular/core/testing';
import { BrowserDynamicTestingModule, platformBrowserDynamicTesting } from
  '@angular/platform-browser-dynamic/testing';
import { By } from '@angular/platform-browser';
import { TestService } from './testservice'
TestBed.initTestEnvironment(
  BrowserDynamicTestingModule, platformBrowserDynamicTesting());
// Using Jasmine functions without importing
// Angular test libraries
describe('TestService', () => {
  let service: TestService;
  beforeEach(() => { service = new TestService(); });
  it('should get data from the service without angular testing libraries',
      () => {
    let length = 5;
    let act = service.getData().length;
    expect(act).toEqual(length);
  });

  //Using Angular Test libraries
  beforeEach(() => {
    TestBed.configureTestingModule({ providers: [TestService] });
  });
  it('should test the reverse string method using angular 2 testing libraries',
    inject([TestService], (testServ: TestService) => {
      let name = 'Mahesh';
      let expected = 'hsehaM';
      let act = testServ.reverseString(name);
      expect(act).toEqual(expected);
    }));
});
```

The code shown in Listing 37.5 has a test case (getDataTest) which does not use the Angular 2 Test library. This test uses Jasmine to creates an instance of the TestService class and call its getData() method.

The testwithangular2lib test uses Angular 2 library. It uses the TestBed class to provide the TestService to it. inject() is used to call the method from the TestService asynchronously. The reverseSting() method is called and executed asynchronously.

To run the test, use servicetest.html, which is similar to servecomponent.html. The only change in this html file is shown in the following listing:

Listing 37.6: *The html file Isolated service testing*



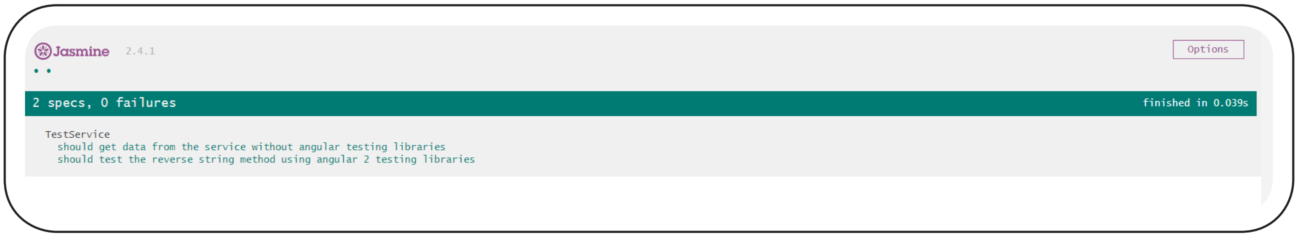
```
<body>
  <script src="systemjs.config.js"></script>
  <script>
    Promise.all([
      System.import('@angular/core/testing'),
      System.import('./app/testservice.spec')
    ])

      // System.import('./app/app.component.spec')
      .then(window.onload)
      .catch(console.error.bind(console));
  </script>
</body>
```

Run the server using following command:

```
npm run start
```

The test can be viewed using the following URL:

http://localhost:3000/servicetest.html



```
• •
2 specs, 0 failures                                          finished in 0.039s

TestService
  should get data from the service without angular testing libraries
  should test the reverse string method using angular 2 testing libraries
```

While creating Angular 2 apps, Service testing is important. It is best to implement a test for the component having dependency on a service, and to perform isolated service testing so that logical errors can be reduced.

Conclusion

Services hold most of the business logic and data operations in an Angular application. It is very important to test the logic in the services to avoid any kind of business and data related bugs in the application. This chapter gives you a good start with testing services. You can build on top of these ideas to test the services in your applications.

Chapter 38

Testing Http Request

SERVICES ARE USED TO contain utility methods and to make external Http calls. Chapter 37 explained the testing of a component using service dependency, as well an isolated test for Angular 2 service with and without the Angular 2 Testing library.

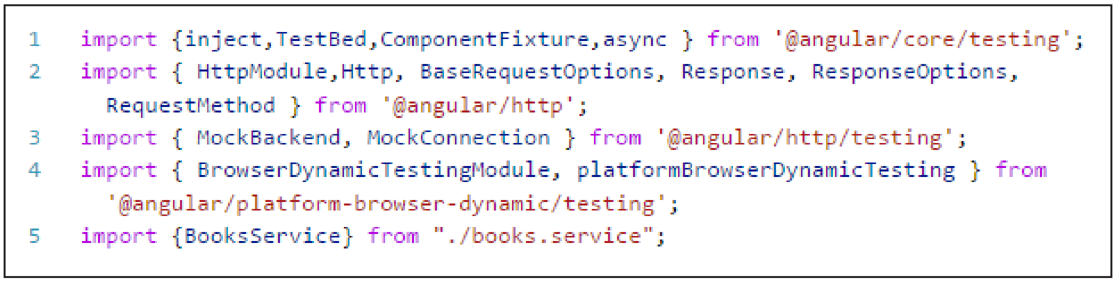
This chapter looks at another use case of Angular 2 Service testing, which makes a Http call to REST API or Web API.

Angular 2 uses the Http module for creating an Injectable service. While writing tests for the service, it is important that the test code should not make an XHR call. Instead, it should respond with a set of mock data. The Angular 2 team has written the **MockBackend** module, which can be used to mock the HTTP calls and provide a static response to the requests.

In Angular 2, the MockBackend and MockConnection modules are provided in @ angular/http/testing package.

This chapter uses the code sample of Chapter 29 to test Http calls. The testing code is written in bookservice.spec.ts. The test code must import modules as shown in the following listing:

Listing 38.1: *The list of modules needed for test*

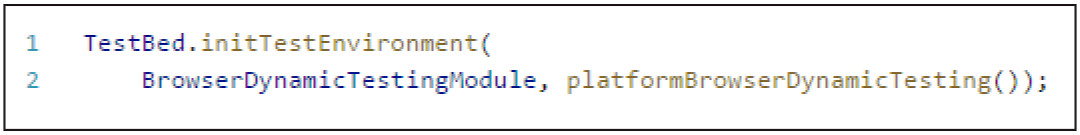


```
import {inject,TestBed,ComponentFixture,async } from '@angular/core/testing';
import { HttpModule,Http, BaseRequestOptions, Response, ResponseOptions,
  RequestMethod } from '@angular/http';
import { MockBackend, MockConnection } from '@angular/http/testing';
import { BrowserDynamicTestingModule, platformBrowserDynamicTesting } from
  '@angular/platform-browser-dynamic/testing';
import {BooksService} from "./books.service";
```

The **Http** module is used to represent the Http call made from the service, to external services. The **MockBackend** module is used to mock the backend. The **MockConnection** represents the state of the XMLHttpRequest.readyState value. The **ResponseOptions** module represents the body of the response, which may be a String, Object, ArrayBuffer or Blob.

The TestBed environment for the test is set using the code shown in listing 38.2.

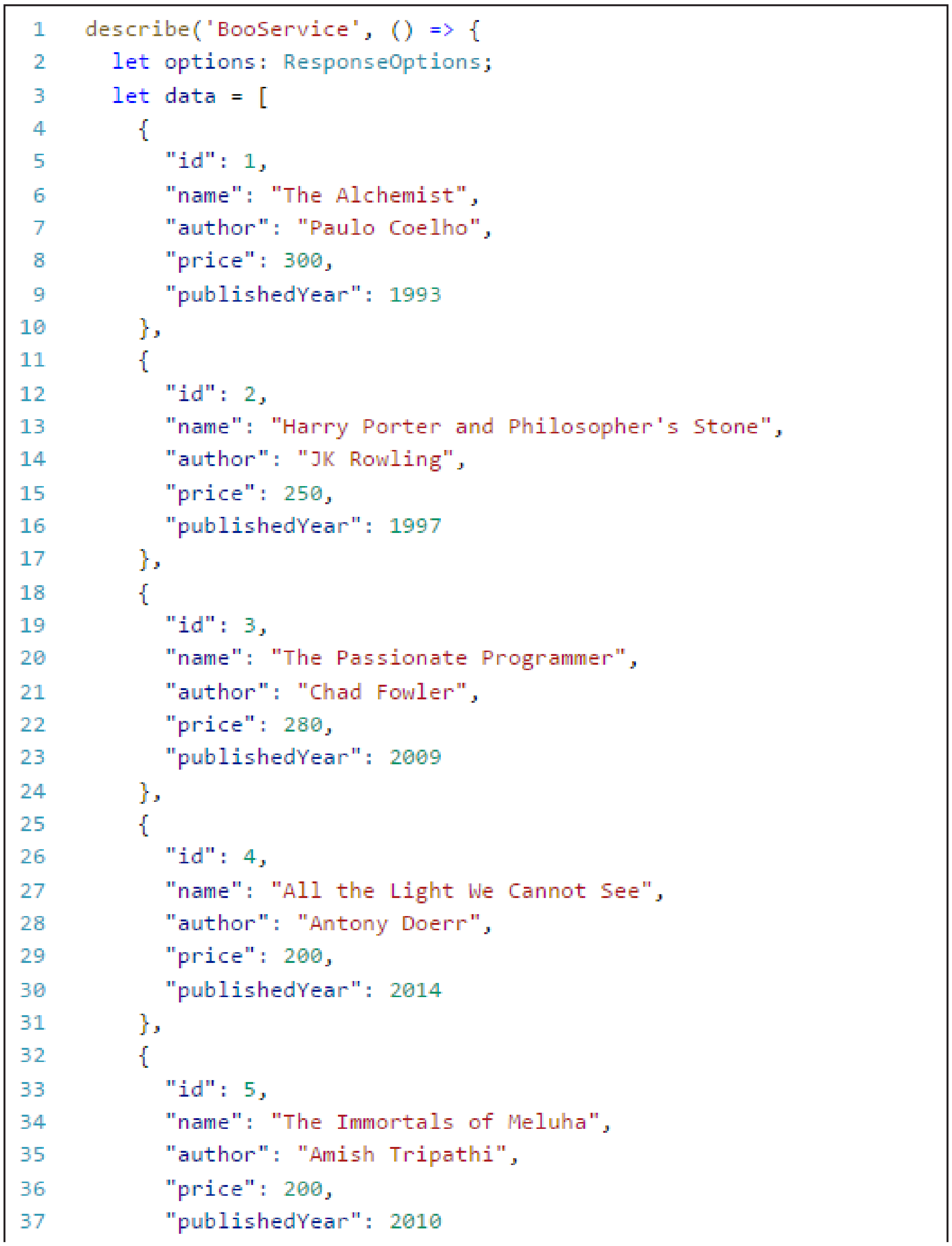
Listing 38.2: *The TestBed environment set*



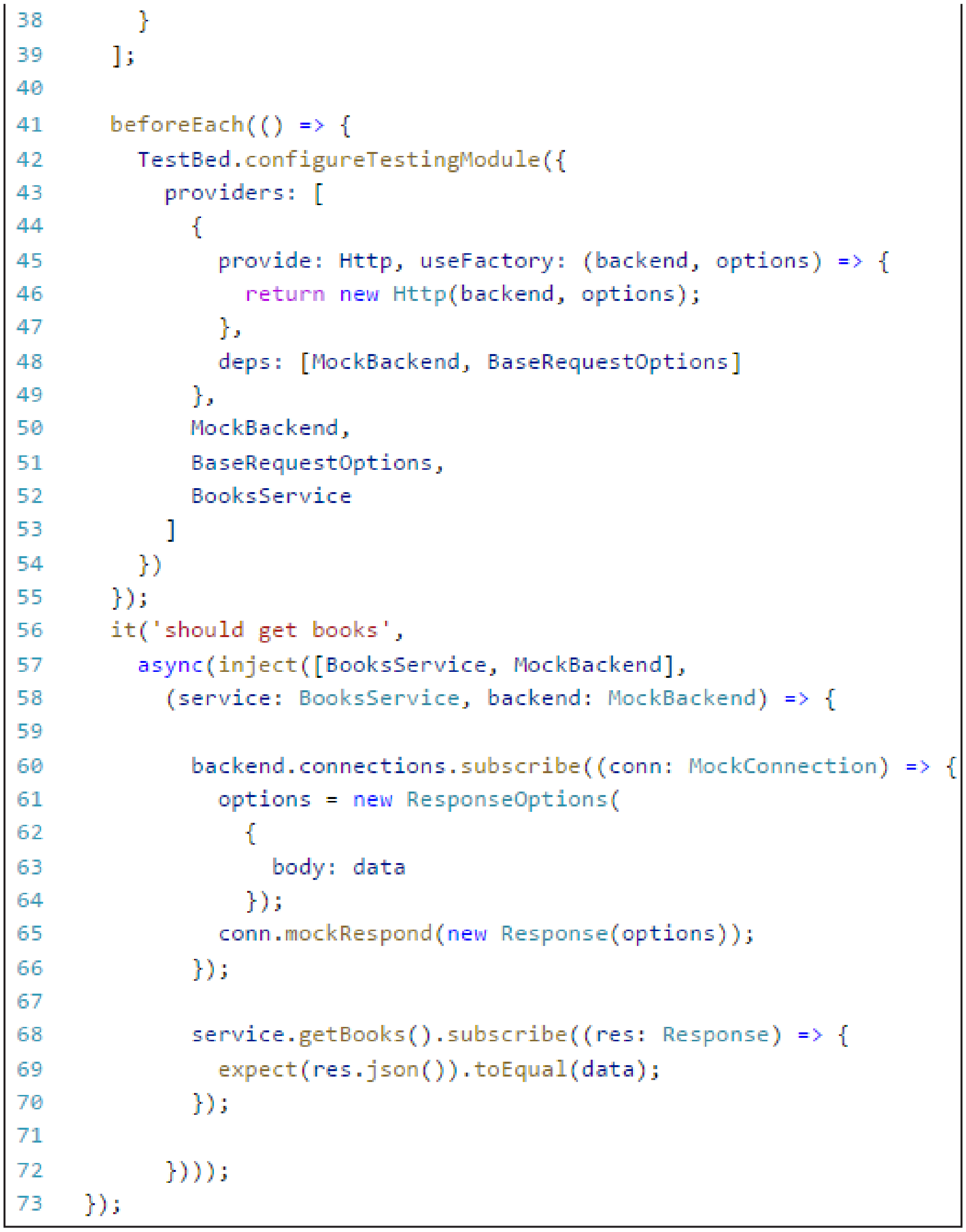
```
TestBed.initTestEnvironment(
    BrowserDynamicTestingModule, platformBrowserDynamicTesting());
```

The test using Jasmine functions is implemented using the code in Listing 38.3:

Listing 38.3: *The Http call test*



```
describe('BooService', () => {
  let options: ResponseOptions;
  let data = [
    {
      "id": 1,
      "name": "The Alchemist",
      "author": "Paulo Coelho",
      "price": 300,
      "publishedYear": 1993
    },
    {
      "id": 2,
      "name": "Harry Porter and Philosopher's Stone",
      "author": "JK Rowling",
      "price": 250,
      "publishedYear": 1997
    },
    {
      "id": 3,
      "name": "The Passionate Programmer",
      "author": "Chad Fowler",
      "price": 280,
      "publishedYear": 2009
    },
    {
      "id": 4,
      "name": "All the Light We Cannot See",
      "author": "Antony Doerr",
      "price": 200,
      "publishedYear": 2014
    },
    {
      "id": 5,
      "name": "The Immortals of Meluha",
      "author": "Amish Tripathi",
      "price": 200,
      "publishedYear": 2010
```



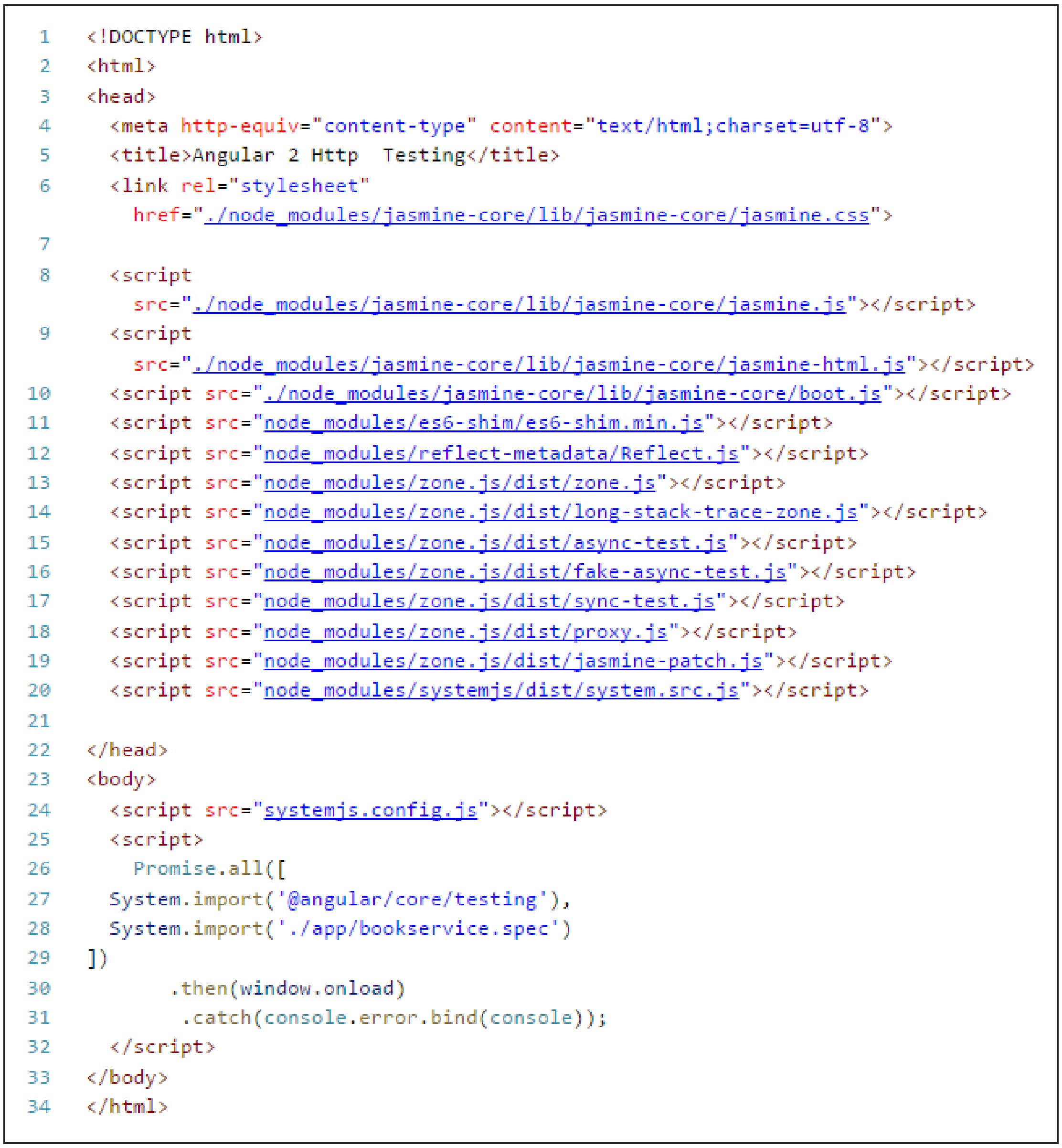
```
    }
  ];

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        {
          provide: Http, useFactory: (backend, options) => {
            return new Http(backend, options);
          },
          deps: [MockBackend, BaseRequestOptions]
        },
        MockBackend,
        BaseRequestOptions,
        BooksService
      ]
    })
  });
  it('should get books',
    async(inject([BooksService, MockBackend],
      (service: BooksService, backend: MockBackend) => {

        backend.connections.subscribe((conn: MockConnection) => {
          options = new ResponseOptions(
            {
              body: data
            });
          conn.mockRespond(new Response(options));
        });

        service.getBooks().subscribe((res: Response) => {
          expect(res.json()).toEqual(data);
        });

      })));
});
```

The code shown in Listing 38.3 declares the ResponseOptions object. This object defines the expected response. The **data** array is declared to contain the data value against which the response will be compared. The beforeEach function configures the required providers using **configureTestingModule** function. These providers are: Http, MockBackend, BaseRequestOptions, and BookService. The provider sets dependency on the MockBackend and BaseRequestOptions modules to mock the Http requests.

Since the method to be tested makes an HTTP call using an Angular 2 service, the it() function performs an asynchronous test by injecting BookService and MockBackend modules. The function subscribes to the backend connection using MockConnection, and sets the ResponseOptions with the expected response body. The MockConnection object further mocks the response using its **mockRepond()** method. Finally the getBooks() method from the service is called, and expectations for the test are set.

Add httptest.html to the project with the following code:

Listing 38.4: *The Html file for testing result*



```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="content-type" content="text/html;charset=utf-8">
  <title>Angular 2 Http  Testing</title>
  <link rel="stylesheet"
    href="./node_modules/jasmine-core/lib/jasmine-core/jasmine.css">

  <script
    src="./node_modules/jasmine-core/lib/jasmine-core/jasmine.js"></script>
  <script
    src="./node_modules/jasmine-core/lib/jasmine-core/jasmine-html.js"></script>
  <script src="./node_modules/jasmine-core/lib/jasmine-core/boot.js"></script>
  <script src="node_modules/es6-shim/es6-shim.min.js"></script>
  <script src="node_modules/reflect-metadata/Reflect.js"></script>
  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/zone.js/dist/long-stack-trace-zone.js"></script>
  <script src="node_modules/zone.js/dist/async-test.js"></script>
  <script src="node_modules/zone.js/dist/fake-async-test.js"></script>
  <script src="node_modules/zone.js/dist/sync-test.js"></script>
  <script src="node_modules/zone.js/dist/proxy.js"></script>
  <script src="node_modules/zone.js/dist/jasmine-patch.js"></script>
  <script src="node_modules/systemjs/dist/system.src.js"></script>

</head>
<body>
  <script src="systemjs.config.js"></script>
  <script>
    Promise.all([
  System.import('@angular/core/testing'),
  System.import('./app/bookservice.spec')
])
       .then(window.onload)
        .catch(console.error.bind(console));
  </script>
</body>
</html>
```

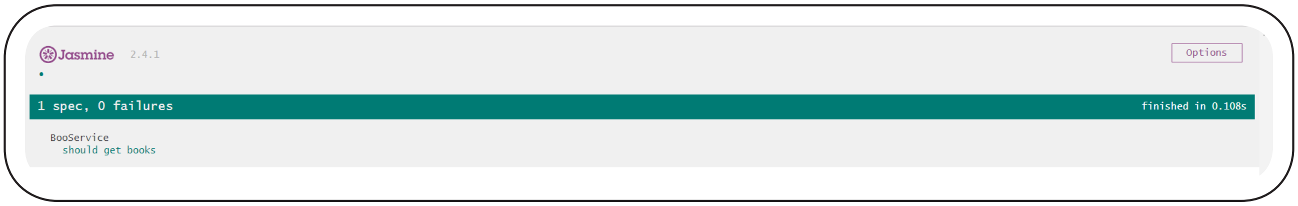
Run the application using the following command:

Npm run start

The Test can be viewed in the browser using the following url:

http://localhost:3000/httptest.html

The result is shown in the following image:



```
1 spec, 0 failures                                finished in 0.108s

  BooService
    should get books
```

Conclusion

Data is the most important thing a user would want to see in an application and calls to HTTP APIs is the most widely used technique to get this data. These calls are crucial, as they result into the content that gets displayed to the users. To ensure that the right APIs are called, they have to be tested. Angular's *MockBackend* helps in doing that. This chapter demonstrated how the HTTP calls can be tested using *MockBackend*.

Chapter 39

Model Driven Form Testing

FORMS ARE AN IMPORTANT component of any Web Application. Forms contain the UI and Validation rules to ensure that the end user can easily interact with a web application and enter data in the desired format.

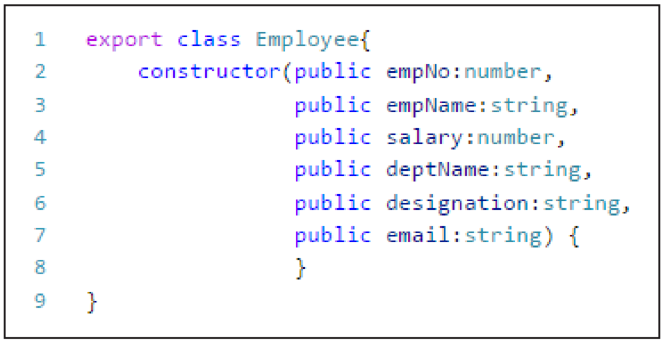
Angular 2 forms, validations, and custom validations have been reviewed in Chapters 21 through 24. This chapter will focus on testing in Model-driven forms.

Model-driven forms provide isolation between Model-Properties and the Form UI elements. This feature makes the form more testable. Tests can be written on the form logic and the form can be tested against its 'valid' and 'invalid' status.

The **FormGroup** object, discussed in chapter 22 through 24, contains collection of **FormControl**. This object contains mapping between DOM elements and the model properties, along with validation rules. The mapping is handled using the **formControlName** attribute of the DOM elements. Using this approach, the model validation code is isolated from the DOM to make it more testable.

This chapter builds on Chapter 24. Chapter 24 has a model class defined in employee. model.ts.

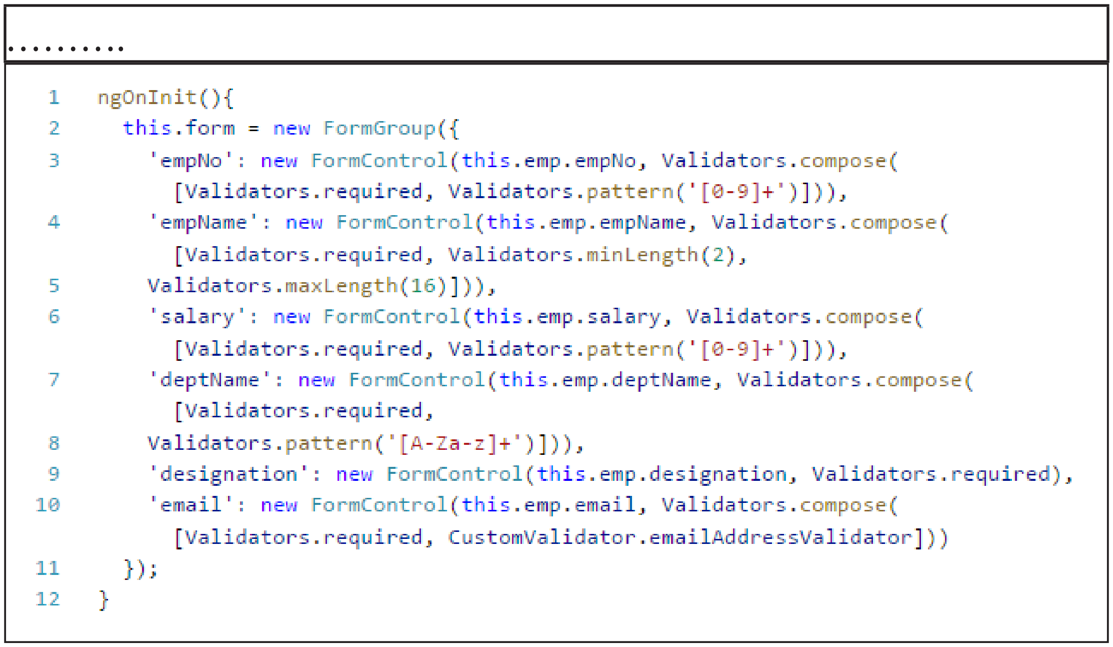
Listing 39.1: *The model class*



```
export class Employee{
    constructor(public empNo:number,
                public empName:string,
                public salary:number,
                public deptName:string,
                public designation:string,
                public email:string) {
                }
}
```

The EmployeeComponent has an ngOnInit() method, which declares a FormGroup instance using FormControl collection, and validation rules as shown in the following listing.

Listing 39.2: *The EmployeeComponent with validation code*

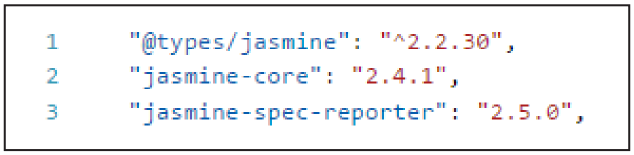


```
..........
ngOnInit(){
  this.form = new FormGroup({
    'empNo': new FormControl(this.emp.empNo, Validators.compose(
      [Validators.required, Validators.pattern('[0-9]+')])),
    'empName': new FormControl(this.emp.empName, Validators.compose(
      [Validators.required, Validators.minLength(2),
    Validators.maxLength(16)])),
    'salary': new FormControl(this.emp.salary, Validators.compose(
      [Validators.required, Validators.pattern('[0-9]+')])),
    'deptName': new FormControl(this.emp.deptName, Validators.compose(
      [Validators.required,
    Validators.pattern('[A-Za-z]+')])),
    'designation': new FormControl(this.emp.designation, Validators.required),
    'email': new FormControl(this.emp.email, Validators.compose(
      [Validators.required, CustomValidator.emailAddressValidator]))
  });
}
```

Please refer to Chapter 24 for more information on the EmployeeComponent class.

To load Jasmine packages for the testing application, package.json must contain the following dependencies shown in Listing 39.3:

Listing 39.3: *The modification in package.json*



```
    "@types/jasmine": "^2.2.30",
    "jasmine-core": "2.4.1",
    "jasmine-spec-reporter": "2.5.0",
```

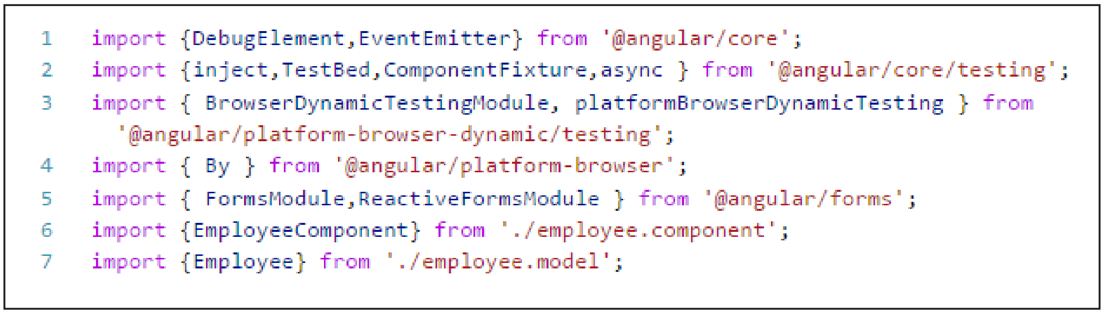
The project contains the employee.html file. This defines the form data, bound with model properties, and their validation error messages. The EmployeeComponent uses the employee.html and the @Component's **templateUrl** property. While testing this component, the test must compile the html template first and then create the instance of the component class.

The project will be added to the employee.component.spec file; it will contain the required logic for the form validation.

The test must import required modules from Angular 2 test libraries. These modules will be used to read DOM elements from the html form. To successfully test the form with databinding, the test code must import Angular 2 forms package.

The following listing shows modules and packages imported in the test file:

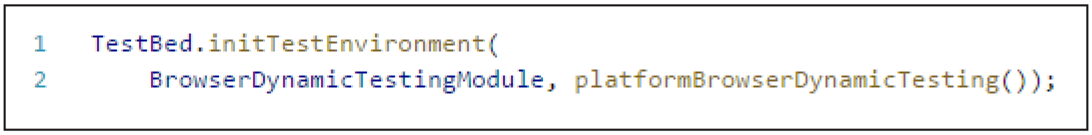
Listing 39.4: *Packages imported in employee.component.spec.ts file*



```
import {DebugElement,EventEmitter} from '@angular/core';
import {inject,TestBed,ComponentFixture,async } from '@angular/core/testing';
import { BrowserDynamicTestingModule, platformBrowserDynamicTesting } from
  '@angular/platform-browser-dynamic/testing';
import { By } from '@angular/platform-browser';
import { FormsModule,ReactiveFormsModule } from '@angular/forms';
import {EmployeeComponent} from './employee.component';
import {Employee} from './employee.model';
```

The following listing shows test environment initialization using the TestBed class:

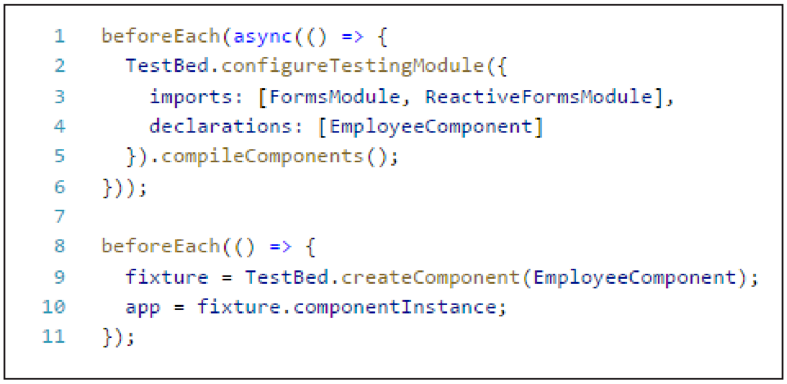
Listing 39.5: *The test environment setting*



```
TestBed.initTestEnvironment(
    BrowserDynamicTestingModule, platformBrowserDynamicTesting());
```

The EmployeeComponent uses employee.html as an external template. The test must first compile the template, and then the Component can be instantiated. The template compilation is performed asynchronously using the **compileComponents()** method of the TestBed class. The TestBed class must import all required modules which are used by the html template. Once the compilation is done successfully, the component can be instantiated. The following listing contains the code for compilation, and the component instance creation.

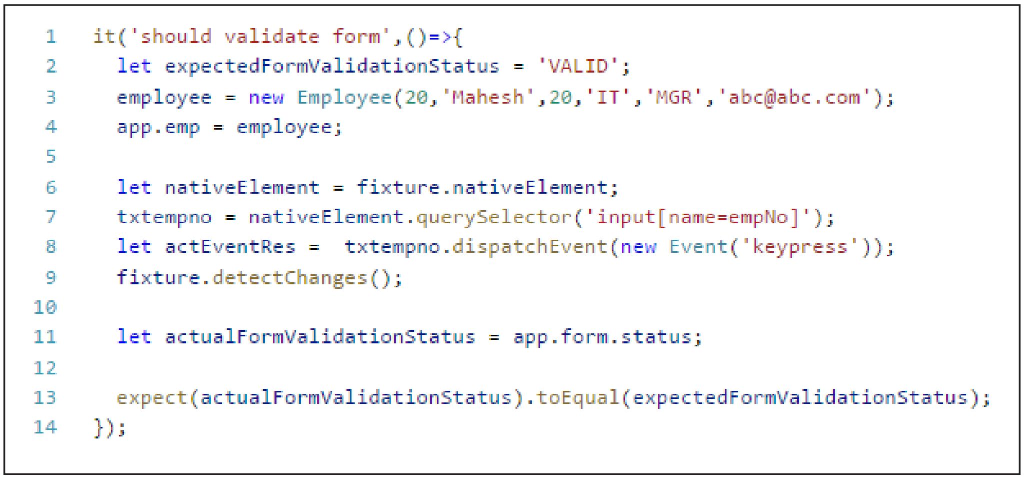
Listing 39.6: *External Template compilation and component instance creation*



```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    imports: [FormsModule, ReactiveFormsModule],
    declarations: [EmployeeComponent]
  }).compileComponents();
}));

beforeEach(() => {
  fixture = TestBed.createComponent(EmployeeComponent);
  app = fixture.componentInstance;
});
```

Since this chapter is scoped for testing Model-Driven form, in the test case the Employee object is instantiated with constructor initialization. The test case also reads the input element from the Html form. The element name is 'empNo' and it raises the **keypress** event. The validation is executed when the databinding is triggered for the element. Listing 39.7 shows the code for the test.

Listing 39.7: *The test case*

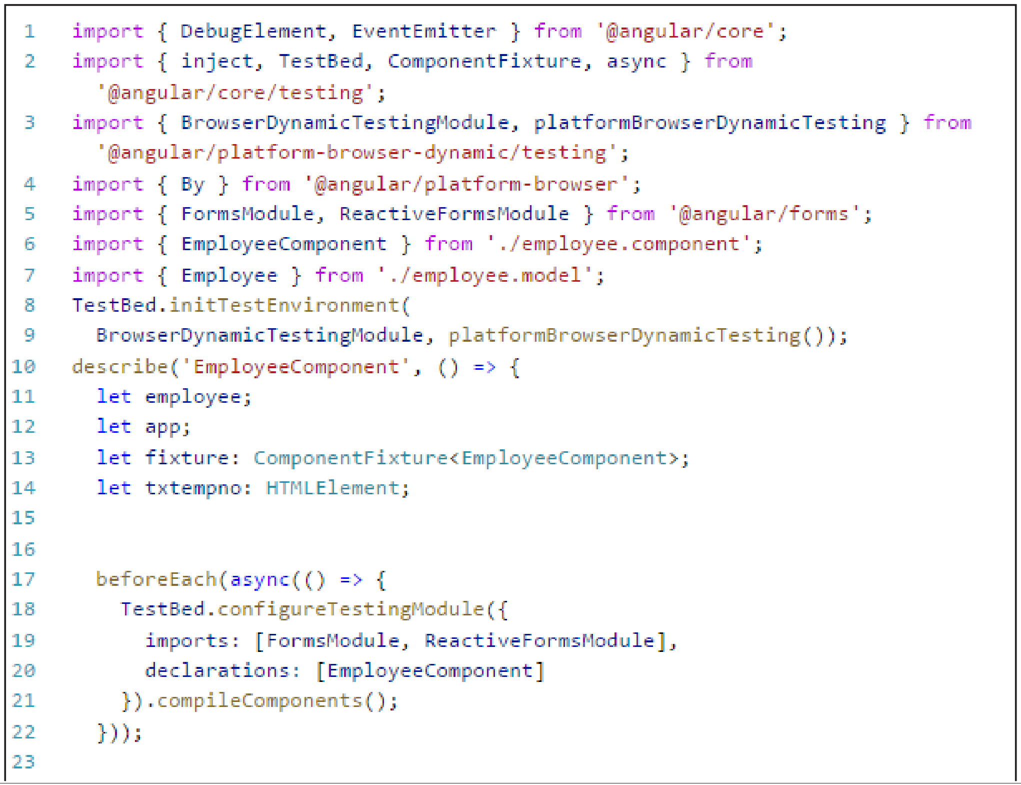


```
it('should validate form',()=>{
  let expectedFormValidationStatus = 'VALID';
  employee = new Employee(20,'Mahesh',20,'IT','MGR','abc@abc.com');
  app.emp = employee;

  let nativeElement = fixture.nativeElement;
  txtempno = nativeElement.querySelector('input[name=empNo]');
  let actEventRes =  txtempno.dispatchEvent(new Event('keypress'));
  fixture.detectChanges();

  let actualFormValidationStatus = app.form.status;

  expect(actualFormValidationStatus).toEqual(expectedFormValidationStatus);
});
```

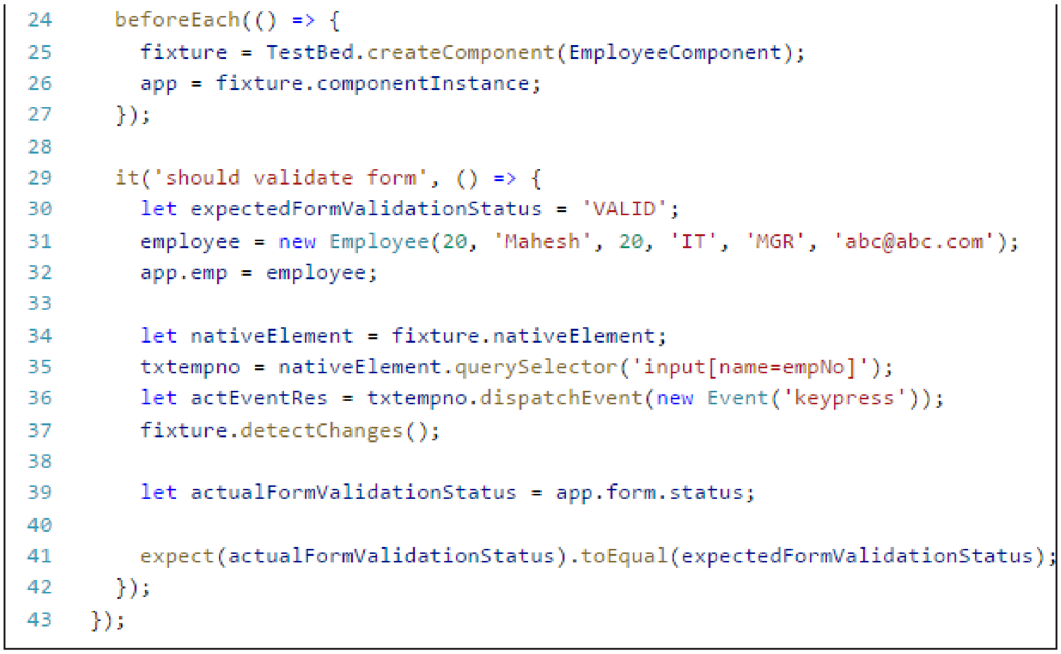
The above code declares a local variable expectedFormValidationStatus; the value is set to VALID. This variable will be used to represent the expectations set. The code declares the Employee instance, with a constructor initialization for properties of the Employee class. The fixture extracts the input element and emits its keypress event. The detectChanges() function of the ComponentFixture class triggers the databinding. After the databinding is changed, the status of the form is read; if it is valid, the test will be successfully executed, otherwise it will fail.

The following code listing shows the complete code:

Listing 39.8: *The complete test code*



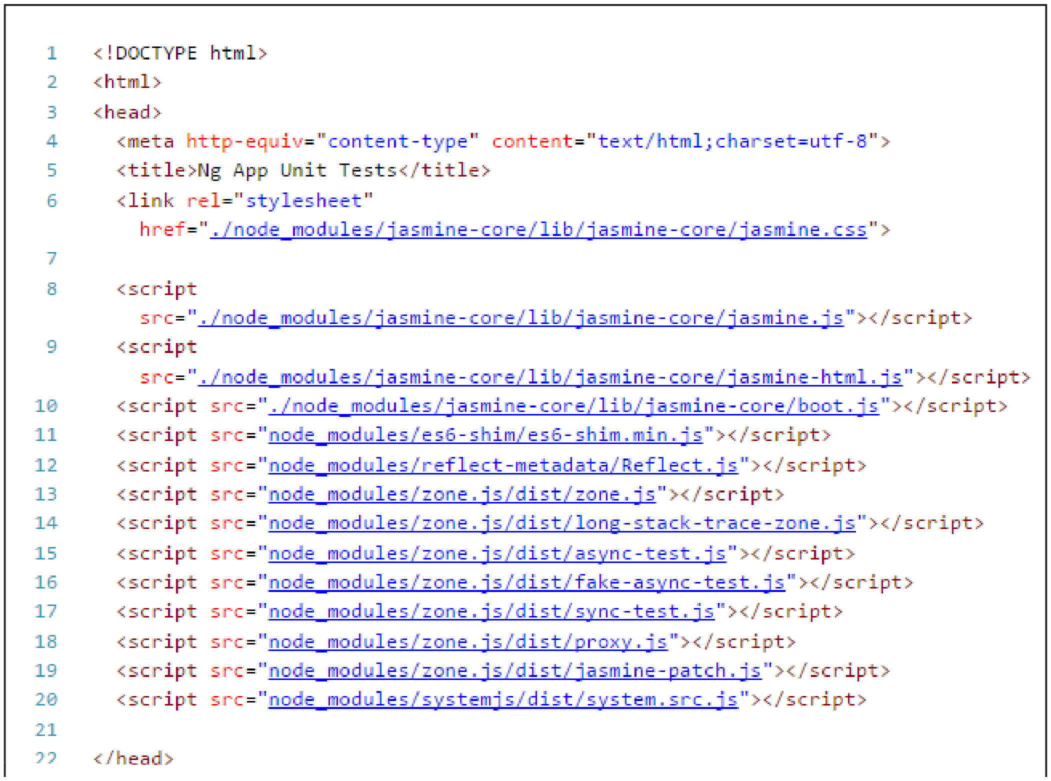
```
import { DebugElement, EventEmitter } from '@angular/core';
import { inject, TestBed, ComponentFixture, async } from
  '@angular/core/testing';
import { BrowserDynamicTestingModule, platformBrowserDynamicTesting } from
  '@angular/platform-browser-dynamic/testing';
import { By } from '@angular/platform-browser';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { EmployeeComponent } from './employee.component';
import { Employee } from './employee.model';
TestBed.initTestEnvironment(
  BrowserDynamicTestingModule, platformBrowserDynamicTesting());
describe('EmployeeComponent', () => {
  let employee;
  let app;
  let fixture: ComponentFixture<EmployeeComponent>;
  let txtempno: HTMLElement;


  beforeEach(async(() => {
    TestBed.configureTestingModule({
      imports: [FormsModule, ReactiveFormsModule],
      declarations: [EmployeeComponent]
    }).compileComponents();
  }));

```



```
  beforeEach(() => {
    fixture = TestBed.createComponent(EmployeeComponent);
    app = fixture.componentInstance;
  });

  it('should validate form', () => {
    let expectedFormValidationStatus = 'VALID';
    employee = new Employee(20, 'Mahesh', 20, 'IT', 'MGR', 'abc@abc.com');
    app.emp = employee;

    let nativeElement = fixture.nativeElement;
    txtempno = nativeElement.querySelector('input[name=empNo]');
    let actEventRes = txtempno.dispatchEvent(new Event('keypress'));
    fixture.detectChanges();

    let actualFormValidationStatus = app.form.status;

    expect(actualFormValidationStatus).toEqual(expectedFormValidationStatus);
  });
});
```

Add the formtest.html file to the project for testing the html. The following listing shows the markup of the html file:

Listing 39.9: *The formtest.html*



```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="content-type" content="text/html;charset=utf-8">
  <title>Ng App Unit Tests</title>
  <link rel="stylesheet"
    href="./node_modules/jasmine-core/lib/jasmine-core/jasmine.css">

  <script
    src="./node_modules/jasmine-core/lib/jasmine-core/jasmine.js"></script>
  <script
    src="./node_modules/jasmine-core/lib/jasmine-core/jasmine-html.js"></script>
  <script src="./node_modules/jasmine-core/lib/jasmine-core/boot.js"></script>
  <script src="node_modules/es6-shim/es6-shim.min.js"></script>
  <script src="node_modules/reflect-metadata/Reflect.js"></script>
  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/zone.js/dist/long-stack-trace-zone.js"></script>
  <script src="node_modules/zone.js/dist/async-test.js"></script>
  <script src="node_modules/zone.js/dist/fake-async-test.js"></script>
  <script src="node_modules/zone.js/dist/sync-test.js"></script>
  <script src="node_modules/zone.js/dist/proxy.js"></script>
  <script src="node_modules/zone.js/dist/jasmine-patch.js"></script>
  <script src="node_modules/systemjs/dist/system.src.js"></script>

</head>
```

```

<body>
  <script src="systemjs.config.js"></script>
  <script>
    Promise.all([
      System.import('@angular/core/testing'),
      System.import('./app/employee.component.spec')
    ])
      .then(window.onload)
      .catch(console.error.bind(console));
  </script>
</body>

</html>
```

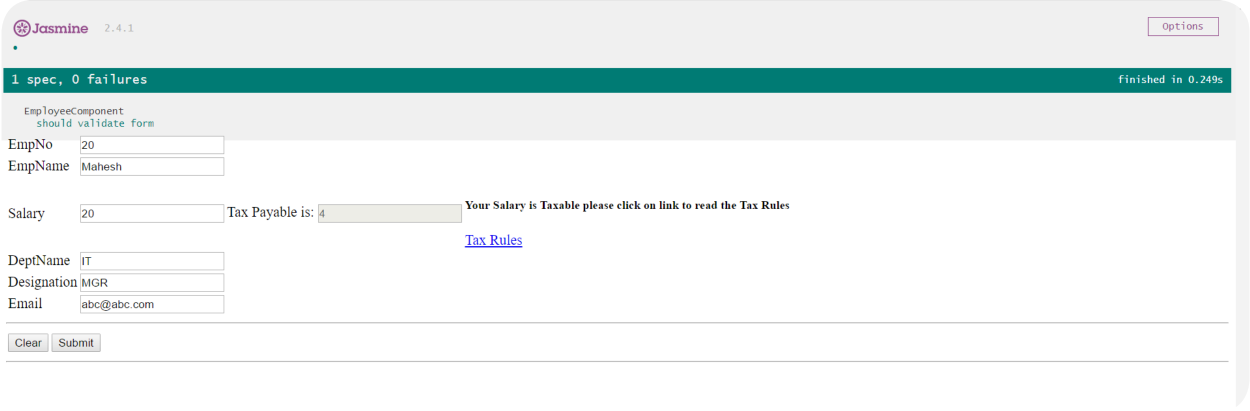
Run the application using the following command:

npm run start

The test result can be seen using the following URL:

http://localhost:3000/formtest.html

Figure 39.1 shows the test result:



1 spec, 0 failures finished in 0.249s

EmployeeComponent
  should validate form

EmpNo 20
EmpName Mahesh
Salary 20 Tax Payable is: 4
Your Salary is Taxable please click on link to read the Tax Rules
Tax Rules
DeptName IT
Designation MGR
Email abc@abc.com

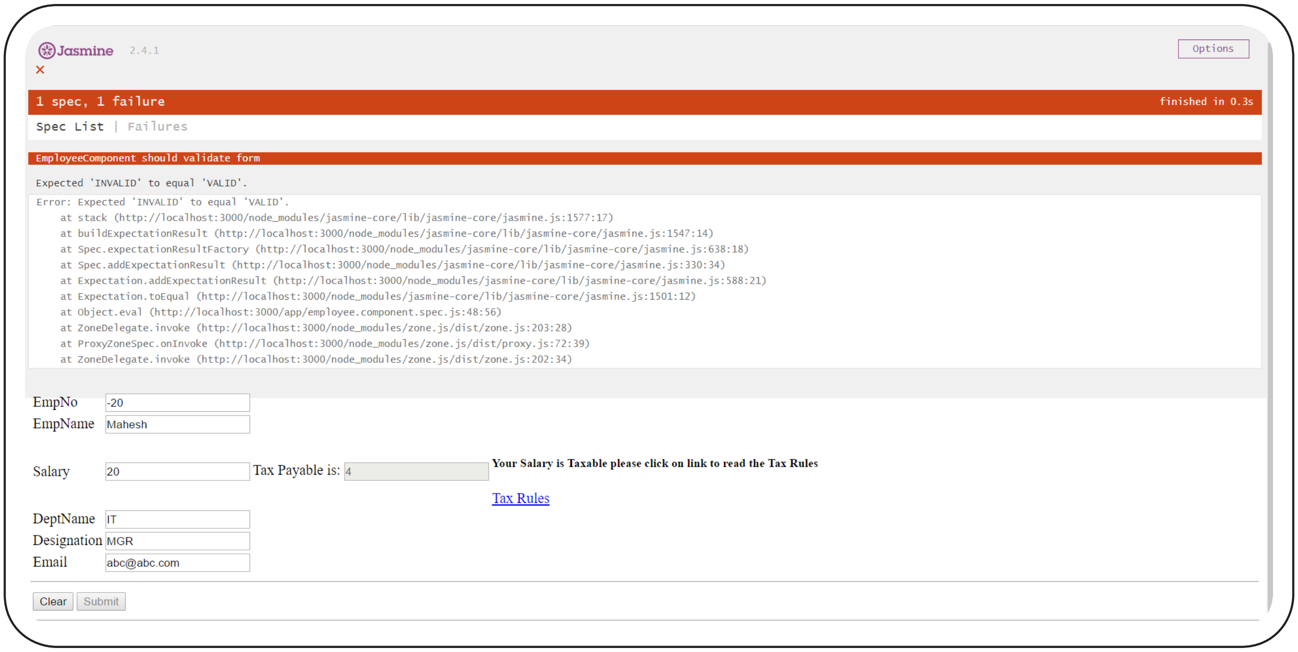
Figure 39.1: *Success test*

Since all of the input elements have valid values per the validation rules, the submit button gets enabled.

Now, give a negative value to the EmpNo in the test case:

```
employee = new Employee(-20,'Mahesh',20,'IT','MGR','abc@abc.com');
```

Run the test. The test will fail as shown in Figure 39.2 and the Submit button will be disabled.



```
1 spec, 1 failure                                                    finished in 0.3s
Spec List | Failures

EmployeeComponent should validate form

Expected 'INVALID' to equal 'VALID'.
Error: Expected 'INVALID' to equal 'VALID'.
    at stack (http://localhost:3000/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:1577:17)
    at buildExpectationResult (http://localhost:3000/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:1547:14)
    at Spec.expectationResultFactory (http://localhost:3000/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:638:18)
    at Spec.addExpectationResult (http://localhost:3000/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:330:34)
    at Expectation.addExpectationResult (http://localhost:3000/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:588:21)
    at Expectation.toEqual (http://localhost:3000/node_modules/jasmine-core/lib/jasmine-core/jasmine.js:1501:12)
    at Object.eval (http://localhost:3000/app/employee.component.spec.js:48:56)
    at ZoneDelegate.invoke (http://localhost:3000/node_modules/zone.js/dist/zone.js:203:28)
    at ProxyZoneSpec.onInvoke (http://localhost:3000/node_modules/zone.js/dist/proxy.js:72:39)
    at ZoneDelegate.invoke (http://localhost:3000/node_modules/zone.js/dist/zone.js:202:34)
```

EmpNo -20
EmpName Mahesh

Salary 20 Tax Payable is: 4 **Your Salary is Taxable please click on link to read the Tax Rules**

Tax Rules

DeptName IT
Designation MGR
Email abc@abc.com

Figure 39.2: *Failed Test*

Conclusion

Model-driven forms allow you to isolate fields from the DOM using FormBuilder, ControlGroup, and Control objects. As these fields are controlled from the code, they have to be tested for correctness. This chapter demonstrated how to test these pieces.

Chapter 40

Angular 2 Debugging

UNTIL THIS POINT, THIS book has demonstrated a number of features of Angular 2 and showcased many examples using these features.

When you start building actual applications using the Angular 2 framework, things may not work the way you want them to. This may happen because of some minor errata induced while writing code, or due to an external factor for like non-responsive REST APIs. In either case, the developer responsible for a piece of code needs to debug the code using the browser's developer tools to understand the reason behind the issue and to find a fix for it.

This chapter will demonstrate how to debug an Angular 2 application using a browser's developer tools, as well as a tool called Augury.

> **NOTE:** This chapter uses the code sample of Chapter 34—Parameterized Routes, Creating Sub-Routes. To follow along, run the sample in the Chrome browser.

Debugging Using Developer Tools

The JavaScript debugger in the browser is a useful tool to debug the client-side script of an application. It provides the various useful options to place a breakpoint, including: inspection of values of variables, showing stack of the functions called, and several other options for debugging.

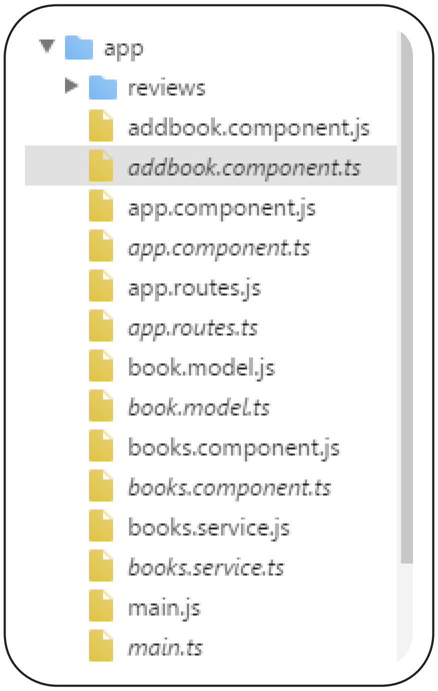
It is possible to debug TypeScript code in the browser through **source maps**. The source map files provide mapping between the converted JavaScript code and the actual TypeScript code. The developer tools of the browser detect the presence of source map files in the deployed code. The source map file and the TypeScript file are downloaded to the browser at the time of debugging.

> **NOTE:** The source map and TypeScript files are not downloaded unless the source map option is enabled in the developer tools and the developer tools are launched in the browser [Ctrl + Shift + I (Windows) or Cmd + Opt + I (Mac)], as they are not needed for the application to run. To enable source maps, check the settings of the developer tools in your browser. Here's a useful link: http://bit.ly/2dCCjhN.

The *tsconfig.json* file has an option to enable source map files when the TypeScript code is converted to JavaScript. If you check the *tsconfig.json* file in any of the samples in this book, you will find the following statement:

```
"sourceMap": true
```

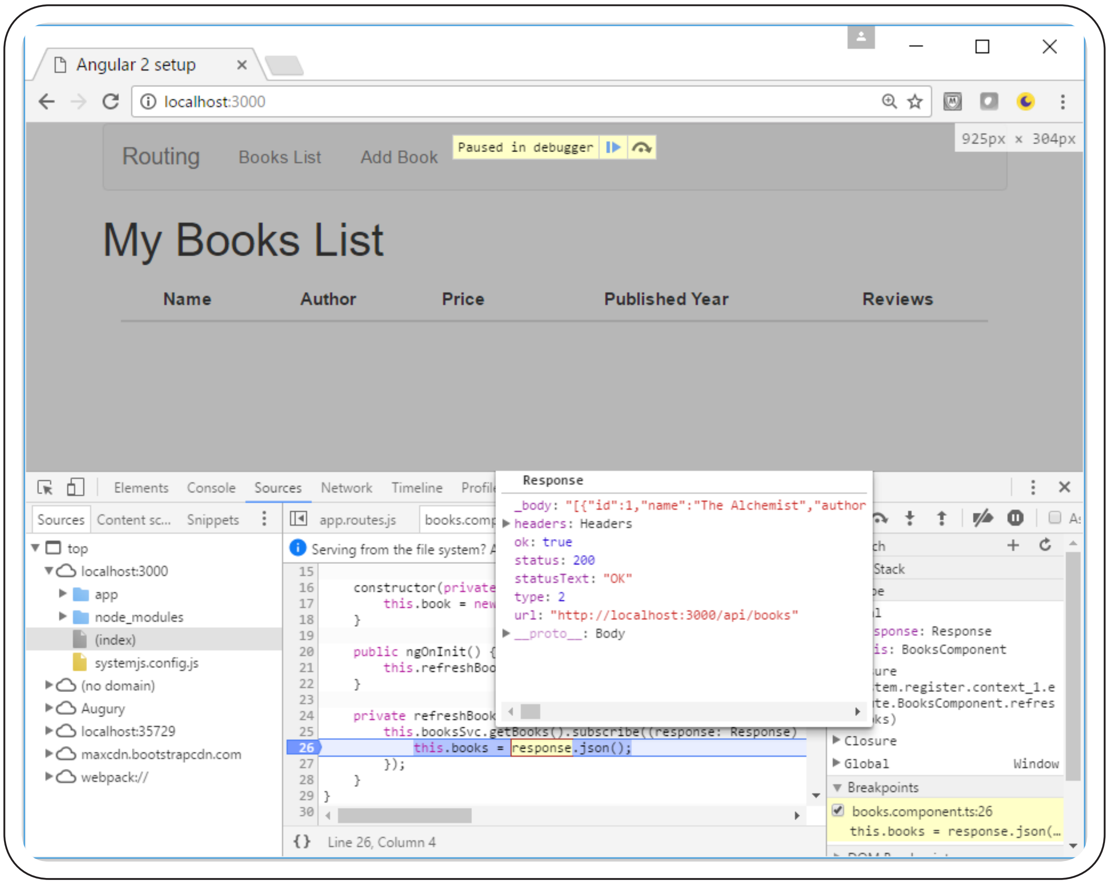
When the *sourceMap* option is set to true in the configuration, the TypeScript compiler produces a *file.js.map* file for every TypeScript file it compiles. If you haven't already noticed this file, you may go back to one of the folders of the samples for reference. When an application runs, open the developer tools, browse the JavaScript source code of the application, and the TypeScript files will be in the browser. Figure 40.1 shows the code of Chapter 34 in the developer tools.



```
▼ app
  ▶ reviews
    addbook.component.js
    addbook.component.ts
    app.component.js
    app.component.ts
    app.routes.js
    app.routes.ts
    book.model.js
    book.model.ts
    books.component.js
    books.component.ts
    books.service.js
    books.service.ts
    main.js
    main.ts
```

Figure 40.1: *Code of Chapter 34 in source tab of developer tools*

The browser gets both JavaScript and TypeScript versions of every file. When you try to place a breakpoint on a statement in the JavaScript file, you will see that it takes you to the corresponding statement in the TypeScript file. Upon refreshing the page (with the developer tools still open), the control will stop at the statement with the breakpoint.



```
    constructor(private
        this.book = new
    }

    public ngOnInit() {
        this.refreshBoo
    }

    private refreshBook
        this.booksSvc.getBooks().subscribe((response: Response)
            this.books = response.json();
        });
    }
}
```

```
Response
_body: "[{"id":1,"name":"The Alchemist","author
▶ headers: Headers
ok: true
status: 200
statusText: "OK"
type: 2
url: "http://localhost:3000/api/books"
▶ __proto__: Body
```

```
sponse: Response
is: BooksComponent
ure
tem.register.context_1.e
te.BooksComponent.refres
ks)
```

Figure 40.2: Debugger stopped at a breakpoint

The next part is the same as debugging JavaScript code on the browser. Although JavaScript debugging in detail is beyond the scope of this book, you can find a great amount of online documentation on this subject.

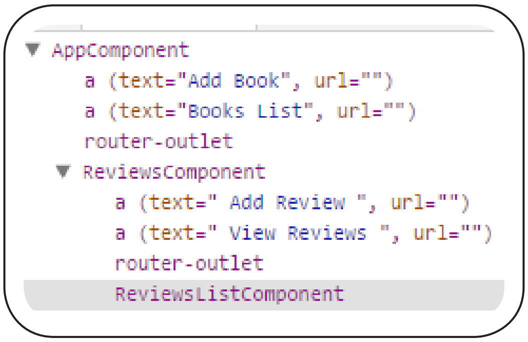
Using Augury

Augury is a Chrome extension, developed by *rangle.io*, for debugging Angular 2 applications. It is similar to Batarang, which was used to debug Angular 1 applications. Augury shows component trees, route trees, details of the components, and it allows you to play with the values of the fields in the component.

Augury can be installed from the *Chrome Web Store*. It adds a tab named Augury to the Chrome developer tools; this tab is used to show the information about the application. It has two inner tabs named Component Tree, Router Tree and NgModules, which will be explained in the following sub sections.

Component Tree

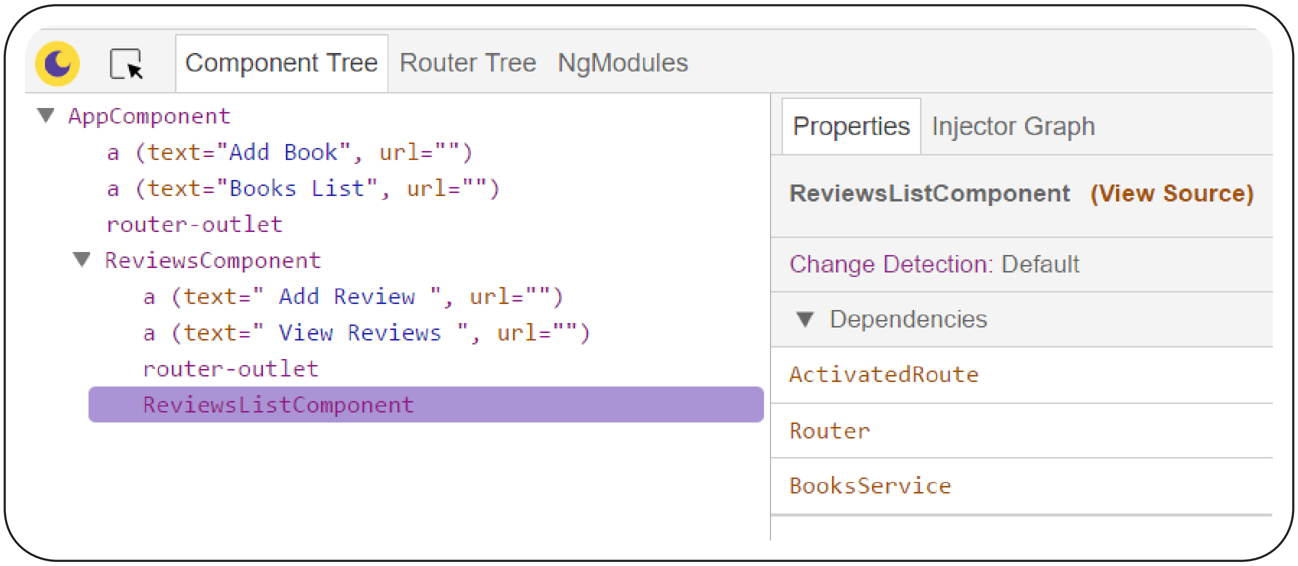
An Angular 2 application is a tree of components. Augury reads the hierarchy in which they are rendered and creates an interactive view using the data. Figure 40.3 shows the hierarchy of the components when a user enters the Reviews List view.



```
▼ AppComponent
    a (text="Add Book", url="")
    a (text="Books List", url="")
    router-outlet
  ▼ ReviewsComponent
      a (text=" Add Review ", url="")
      a (text=" View Reviews ", url="")
      router-outlet
      ReviewsListComponent
```

Figure 40.3: *A sample components tree in Augury*

As shown in Figure 40.3, it lists all components and the HTML elements on the page. Details of a component can be inspected in the right pane under the Component Tree tab by clicking a node of the component. Figure 40.4 shows the details of *ReviewsListComponent*.



```
▼ AppComponent
    a (text="Add Book", url="")
    a (text="Books List", url="")
    router-outlet
  ▼ ReviewsComponent
      a (text=" Add Review ", url="")
      a (text=" View Reviews ", url="")
      router-outlet
      ReviewsListComponent
```

ReviewsListComponent (View Source)

Change Detection: Default

▼ Dependencies

ActivatedRoute

Router

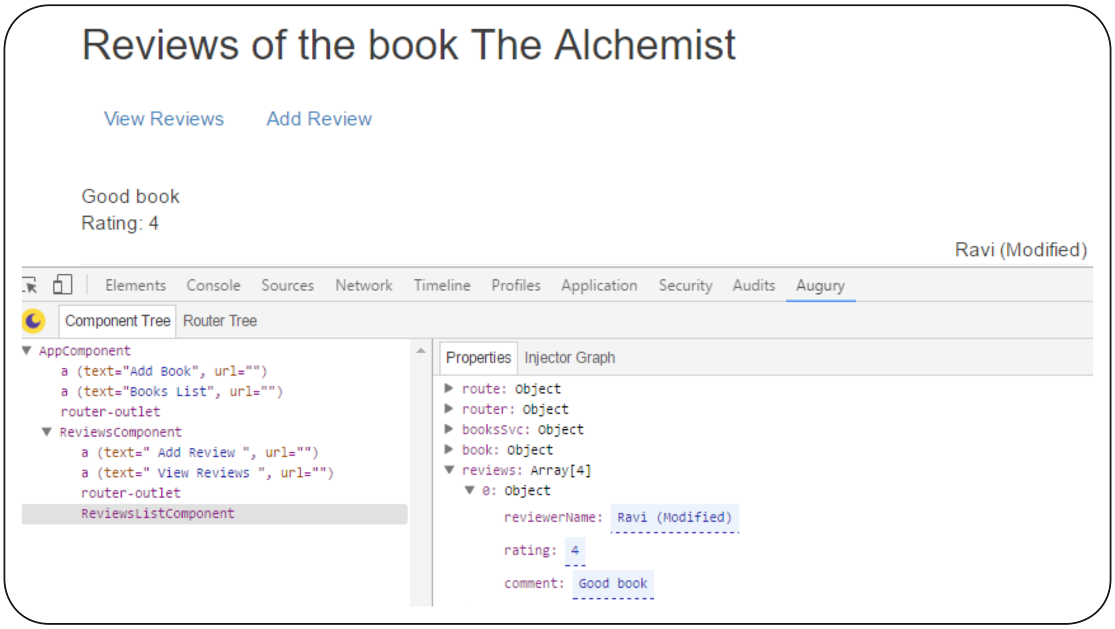
BooksService

Figure 40.4: *Properties of ReviewsListComponent*

Following are the possible operations in this view:

- Double clicking the component node takes you to the HTML element of the component
- View Source link in the Properties tab takes you to the TypeScript code of the component
- The accordion state shows values of the fields in the component object. You can see values of these objects by expanding the nodes
- The Dependencies accordion shows the list of dependencies injected into the component

The values of the objects can be modified in the State accordion, and the modified values will be reflected in the view immediately. Figure 40.5 modifies a value and shows the result on the page.



Reviews of the book The Alchemist

View Reviews   Add Review

Good book
Rating: 4

Ravi (Modified)

```
▼ AppComponent
    a (text="Add Book", url="")
    a (text="Books List", url="")
    router-outlet
  ▼ ReviewsComponent
      a (text=" Add Review ", url="")
      a (text=" View Reviews ", url="")
      router-outlet
      ReviewsListComponent
```

```
▶ route: Object
▶ router: Object
▶ booksSvc: Object
▶ book: Object
▼ reviews: Array[4]
  ▼ 0: Object
      reviewerName: Ravi (Modified)
      rating: 4
      comment: Good book
```

Figure 40.5: *Modified value in Augury*

The Injector Graph tab shows the list of dependencies of the selected component and their hierarchy. The dependency tree starts with the root-most component and it shows the source components where they are registered. If the providers of the dependencies are registered at the module level, the first component of the module becomes the source for the dependency. If a provider is registered in a component, the component becomes the source. Figure 40.6 shows the dependency graph. Here, the dependency *SampleService* is registered in *ReviewsComponent,* and the rest of the dependencies are registered in the module. So the graph shows a subtree for *SampleService*.

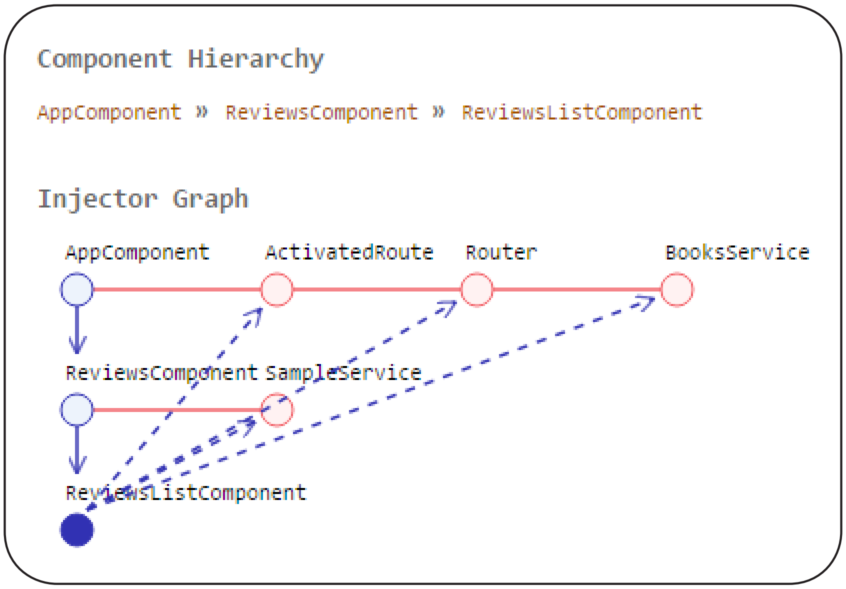


Figure 40.6: *Injector Graph*

Router Tree

The router tree tab shows a tree of the routes configured in the application. To plot this tree, Augury needs the *Router* service to be injected in the root component of the application, and the name of the object injected to be *router*. As of this writing, the router tree plots only one level of the tree and, for the default and child routes, it shows a leaf node named *no-name-route*. There is a bug registered in the GitHub repository for the child routes. Hopefully this will be remedied in the near future. Figure 40.7 shows the router tree of the code in Chapter 34.

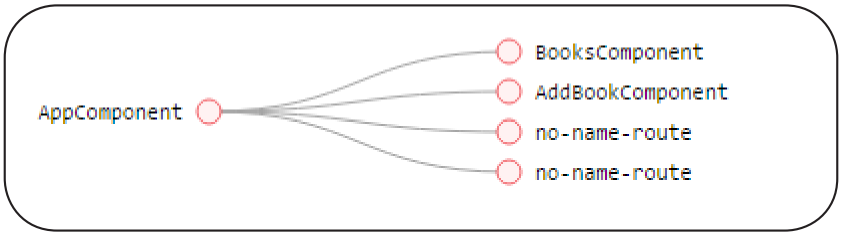
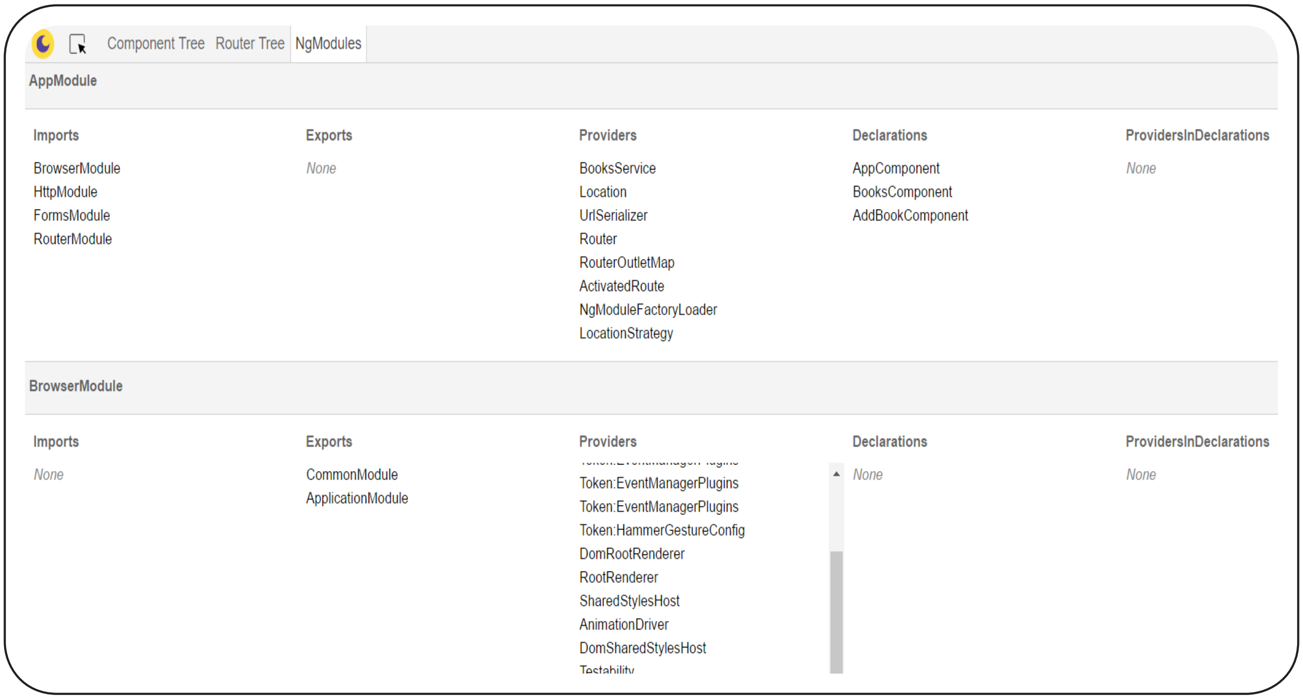


Figure 40.7: *Router Tree*

NgModules

The NgModules tab shows the list of modules currently loaded on the page and their details. The details include the modules imported by the module, the members exported by the module, the services added to the providers section in the module, the blocks added to the declaration section of the module and the ProvidersInDeclaration. Figure 40.8 shows two of the modules loaded in the reviews page of the sample:



AppModule

Imports	Exports	Providers	Declarations	ProvidersInDeclarations
BrowserModule HttpModule FormsModule RouterModule	*None*	BooksService Location UrlSerializer Router RouterOutletMap ActivatedRoute NgModuleFactoryLoader LocationStrategy	AppComponent BooksComponent AddBookComponent	*None*

BrowserModule

Imports	Exports	Providers	Declarations	ProvidersInDeclarations
None	CommonModule ApplicationModule	Token:EventManagerPlugins Token:EventManagerPlugins Token:HammerGestureConfig DomRootRenderer RootRenderer SharedStylesHost AnimationDriver DomSharedStylesHost Testability	*None*	*None*

Figure 40.8: *Modules*

Conclusion

Effective debugging saves time and clarifies an issue that has occurred in the application. Browser developer tools can be used to debug TypeScript code of Angular 2 applications. The chapter also showed how Augury can be used to inspect Angular 2 components and its objects.

Chapter

41

Building and Deploying Angular 2 App using WebPack

CODE IN FRONT-END APPLICATIONS usually spans across several files. In addition to the thousands of lines of JavaScript code, these applications have CSS files, HTML template files, static images, possibly JSON files, and various others. Developers working on an application may use pre-processing languages like TypeScript, CoffeeScript, SASS, LESS or similar technologies to make their work productive.

The build setup of the application must be defined with all of these technologies in mind. The build system should give users the flexibility to configure the build setup targeting development and production workflows.

Webpack is a module loader and bundler that solves this problem. *Webpack considers every file in the application as a module*. By default, it treats all files as JavaScript modules, so if you have other file types in your application, Webpack must be taught to recognize them. This is done using extensibility point, which Webpack provides through *loaders*. Loaders are the plugins that make Webpack understand the way it has to convert a file into a module. The Webpack team has authored loaders for most of the common scenarios and the community has created a large number of loaders, too.

Setting up Webpack

Like any other Node.js based task runner, Webpack can be installed in the project or at the system level using npm. The following command installs Webpack globally:

```
> npm install –g webpack
```

To check if the installation was successful, you may run the following command to check for the version of Webpack:

```
> webpack --version
```

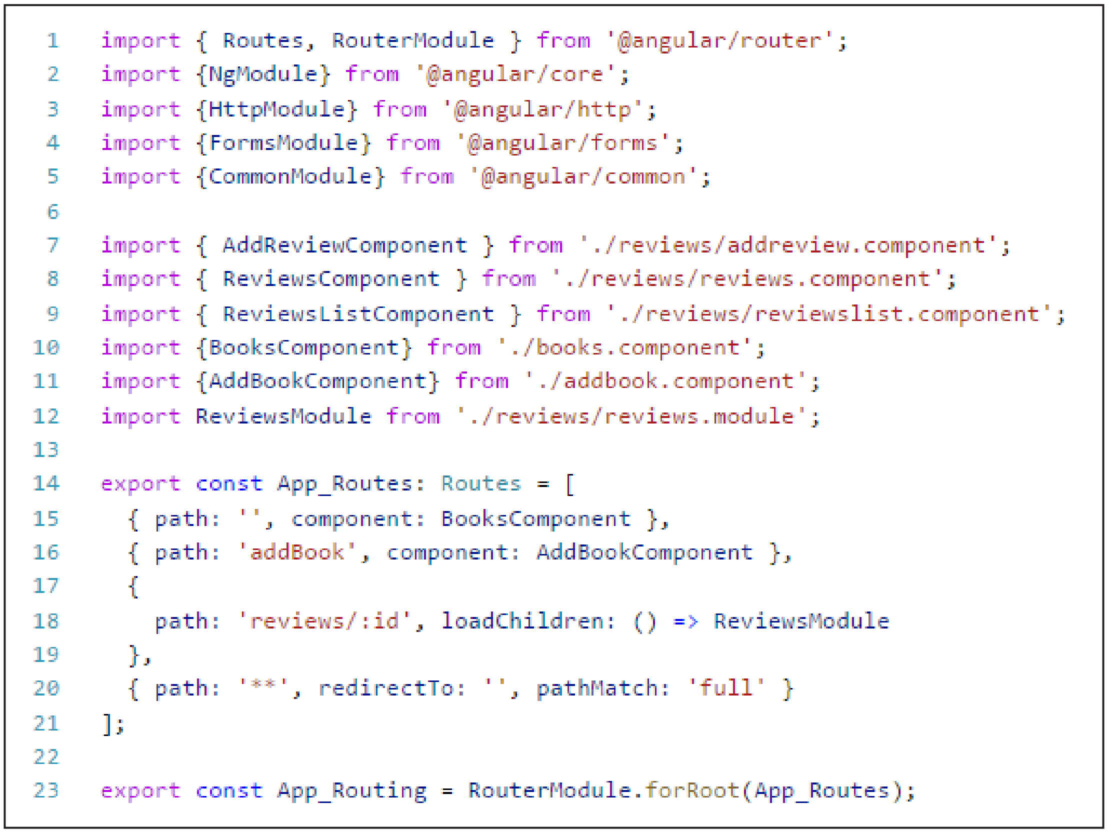
As Webpack is a dependency to be used during development and not during production; it must be installed as a dev dependency in the application. The following command has to be used to install Webpack in a project:

```
> npm install webpack --save-dev
```

Setting up the sample aplication

This chapter uses a slightly modified version of the sample built at the end of Chapter 34. A modification will be made to the way children routes of the reviews module are loaded. The sample in Chapter 34 specifies path of the module file in the *loadChildren* property, which loads the module related files dynamically when the reviews route is being loaded. It is difficult to make this work with Webpack, so it must be modified to load the module *before* the route is registered. Listing 41.1 shows the modified routes.

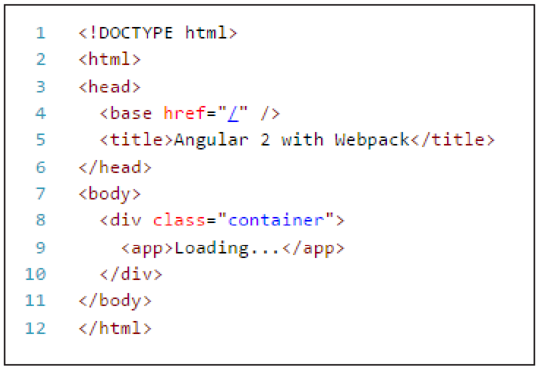
Listing 41.1: *Modified app.routes.ts*



```
import { Routes, RouterModule } from '@angular/router';
import {NgModule} from '@angular/core';
import {HttpModule} from '@angular/http';
import {FormsModule} from '@angular/forms';
import {CommonModule} from '@angular/common';

import { AddReviewComponent } from './reviews/addreview.component';
import { ReviewsComponent } from './reviews/reviews.component';
import { ReviewsListComponent } from './reviews/reviewslist.component';
import {BooksComponent} from './books.component';
import {AddBookComponent} from './addbook.component';
import ReviewsModule from './reviews/reviews.module';

export const App_Routes: Routes = [
  { path: '', component: BooksComponent },
  { path: 'addBook', component: AddBookComponent },
  {
    path: 'reviews/:id', loadChildren: () => ReviewsModule
  },
  { path: '**', redirectTo: '', pathMatch: 'full' }
];

export const App_Routing = RouterModule.forRoot(App_Routes);
```

The SystemJS module loader is not needed as Webpack will convert all of the modules into its own CommonJS-based module system. It is safe to remove the npm package installing the SystemJS package, the SystemJS configuration file, and the script and style references specified in *index.html*. Webpack adds the required script and style reference tags dynamically to the file *index.html*. Now the *index.html* file is a plain HTML file with an element for the main component of the Angular2 application. Listing 41.2 shows the modified index.html file:

Listing 41.2—Modified index.html file



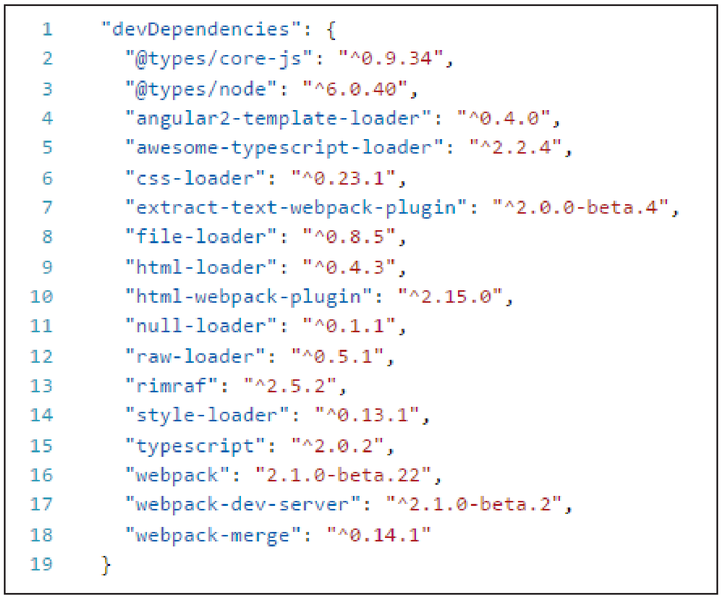
```
<!DOCTYPE html>
<html>
<head>
  <base href="/" />
  <title>Angular 2 with Webpack</title>
</head>
<body>
  <div class="container">
    <app>Loading...</app>
  </div>
</body>
</html>
```

Webpack has its own web server to be used for development. To use it without any difficulty, the server pieces of the sample have to be moved into a different application. The sample code of this chapter has two folders: one containing the client-side Angular code and another containing the server side Koa.js code. The server code enables CORS to allow the client application use the APIs from a different domain. Check the *server* folder in the sample code of this chapter to learn more about the APIs.

Using Webpack in the Sample

Webpack requires a set of packages, including: Webpack, Webpack dev server, and a set of loaders to make Webpack understand about the files that the application uses. Replace the *devDevepdencies* section of the *package.json* file with the following:

Listing 41.3 -



```
"devDependencies": {
  "@types/core-js": "^0.9.34",
  "@types/node": "^6.0.40",
  "angular2-template-loader": "^0.4.0",
  "awesome-typescript-loader": "^2.2.4",
  "css-loader": "^0.23.1",
  "extract-text-webpack-plugin": "^2.0.0-beta.4",
  "file-loader": "^0.8.5",
  "html-loader": "^0.4.3",
  "html-webpack-plugin": "^2.15.0",
  "null-loader": "^0.1.1",
  "raw-loader": "^0.5.1",
  "rimraf": "^2.5.2",
  "style-loader": "^0.13.1",
  "typescript": "^2.0.2",
  "webpack": "2.1.0-beta.22",
  "webpack-dev-server": "^2.1.0-beta.2",
  "webpack-merge": "^0.14.1"
}
```

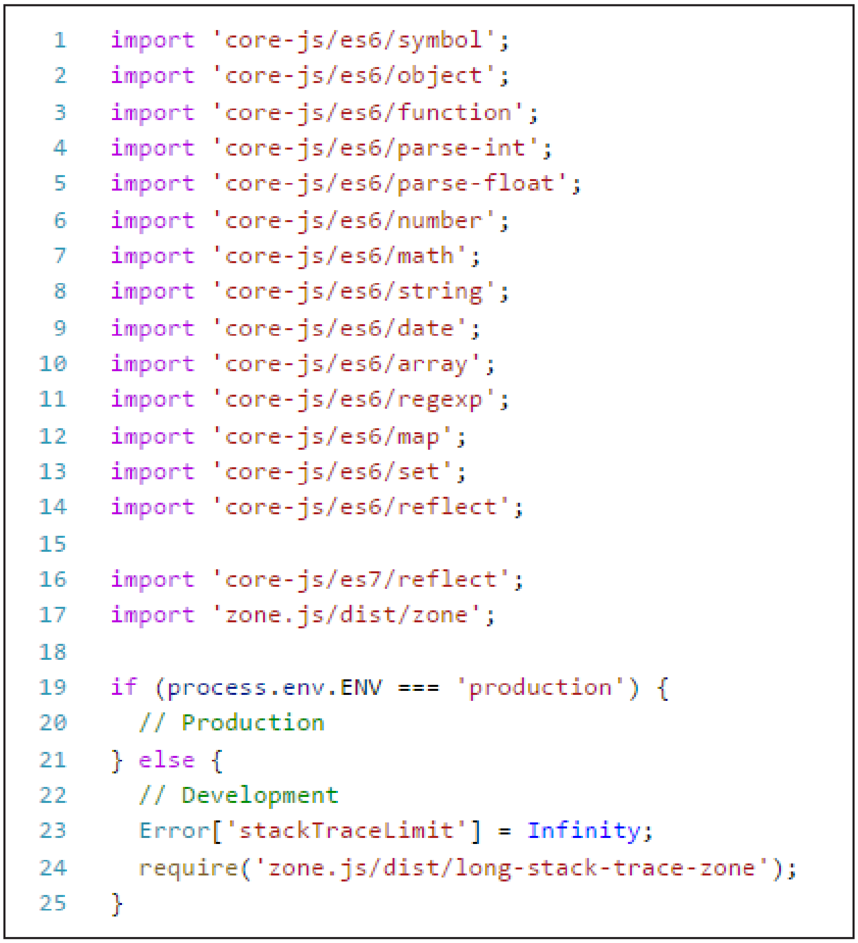
The following bullet points explain the role of the packages installed for Webpack in listing 41.3:

- **webpack:** Installs Webpack in the application
- **webpack-dev-server:** The development server of Webpack. It launches the application on a light-weight server. It keeps watching the source files for any changes and runs the Webpack bundling process whenever it finds a change in the source
- **webpack-merge:** This package allows merging of multiple Webpack configurations into one. It helps in not repeating the Webpack configuration and extends a configuration from another
- **angular2-template-loader:** To load the HTML templates used in the Angular 2 components, where the component uses *templateUrl* property of the metadata to load the template
- **awesome-typescript-loader:** To transpile TypeScript code to JavaScript using the configuration set in the *tsconfig.json* file so that Webpack's JavaScript loader can load it
- **css-loader:** To make Webpack understand CSS files
- **extract-text-webpack-plugin:** A plugin to split the code into multiple bundles
- **file-loader:** A generic file loader, accepts a set of extensions and loads the matched files to the bundle
- **html-loader:** To load HTML template files. It processes the statements that load the templates using a module loading technique, like *require()*
- **html-webpack-plugin:** A Webpack plugin to simplify creation of HTML file to serve the Webpack bundles created in application
- **null-loader:** Loads an empty module
- **raw-loader:** Simply reads and loads a file as a module; it doesn't try to understand what code the file has
- **style-loader:** To dynamically inject <style> or <link> tags to the HTML file

All of these packages will be used in the upcoming sections.

Webpack works based on entry points. An entry point is a JavaScript file that loads a number of other files internally using a module system. The application will have three entry points: one to load the polyfills (like core-js and rxjs), one to load the libraries required by the application, and one to load the application's code. The file *main.ts* in the *app* folder can be used as the entry point for the application. Two more files must be added to define the entry points for the other two chains of files. Add a file named *polyfills.ts* to the app folder and add the following code to it:

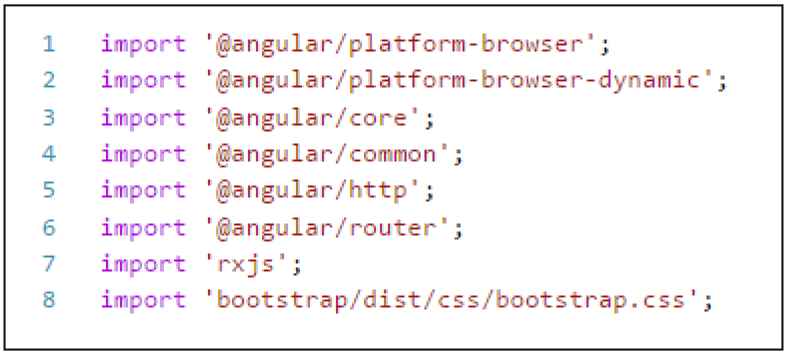
Listing 41.4: *Code of polyfills.ts*



```
import 'core-js/es6/symbol';
import 'core-js/es6/object';
import 'core-js/es6/function';
import 'core-js/es6/parse-int';
import 'core-js/es6/parse-float';
import 'core-js/es6/number';
import 'core-js/es6/math';
import 'core-js/es6/string';
import 'core-js/es6/date';
import 'core-js/es6/array';
import 'core-js/es6/regexp';
import 'core-js/es6/map';
import 'core-js/es6/set';
import 'core-js/es6/reflect';

import 'core-js/es7/reflect';
import 'zone.js/dist/zone';

if (process.env.ENV === 'production') {
  // Production
} else {
  // Development
  Error['stackTraceLimit'] = Infinity;
  require('zone.js/dist/long-stack-trace-zone');
}
```

Add another file for the libraries and name it *vendor.ts*. Add the following code to this file:

Listing 41.5: *Code of vendor.ts*



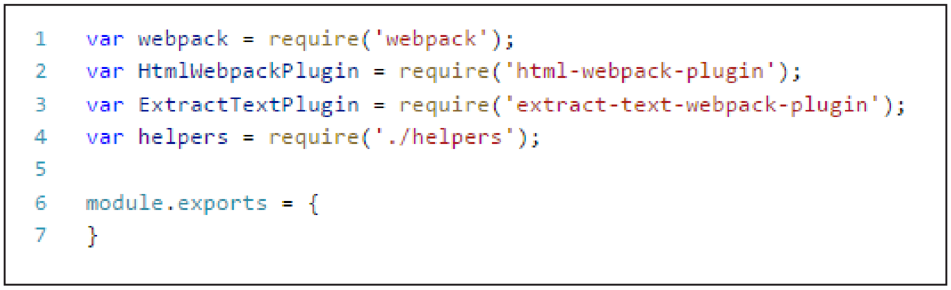
```
import '@angular/platform-browser';
import '@angular/platform-browser-dynamic';
import '@angular/core';
import '@angular/common';
import '@angular/http';
import '@angular/router';
import 'rxjs';
import 'bootstrap/dist/css/bootstrap.css';
```

Configuring Common Tasks

Though the workflows for development and production will vary, they have much in common. The applications must load the same set of files and they have to be processed in the same way. It is better to have a configuration file that can be used for both development and production; the target-specific configuration can be extended from the common configuration using the *webpack-merger* package.

To maintain the Webpack configuration files, add a new folder named *config* and add a new named *webpack.common.js*. It will be processed by a Node.js process, so the code in the file will look similar to any Node.js file. It imports the required objects and exports the Webpack configuration object. Listing 41.6 shows the skeleton code in this file:

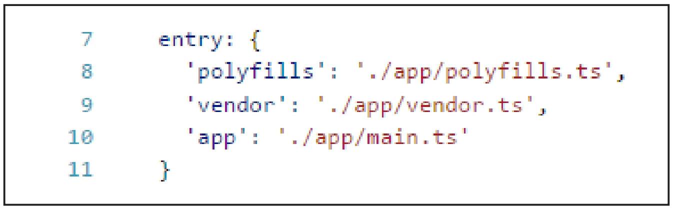
Listing 41.6: *Skeleton of code in webpack.common.js*



```
var webpack = require('webpack');
var HtmlWebpackPlugin = require('html-webpack-plugin');
var ExtractTextPlugin = require('extract-text-webpack-plugin');
var helpers = require('./helpers');

module.exports = {
}
```

The first thing the Webpack needs is the set of entry points as it will be passed as an object with every entry point as a property in the object, and the value pointing to the corresponding file. Listing 41.7 shows the entry points for the sample application.

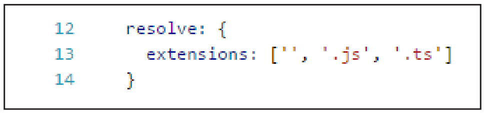
Listing 41.7: *Webpack entry points*



```
entry: {
  'polyfills': './app/polyfills.ts',
  'vendor': './app/vendor.ts',
  'app': './app/main.ts'
}
```

Notice the source code doesn't specify extensions of the modules loaded by the source files. The list of possible extensions must be passed to Webpack so it can load the files correctly during bundling. For this, the following snippet has to be added to the configuration in *webpack.common.js*:

Listing 41.8: *File extensions*



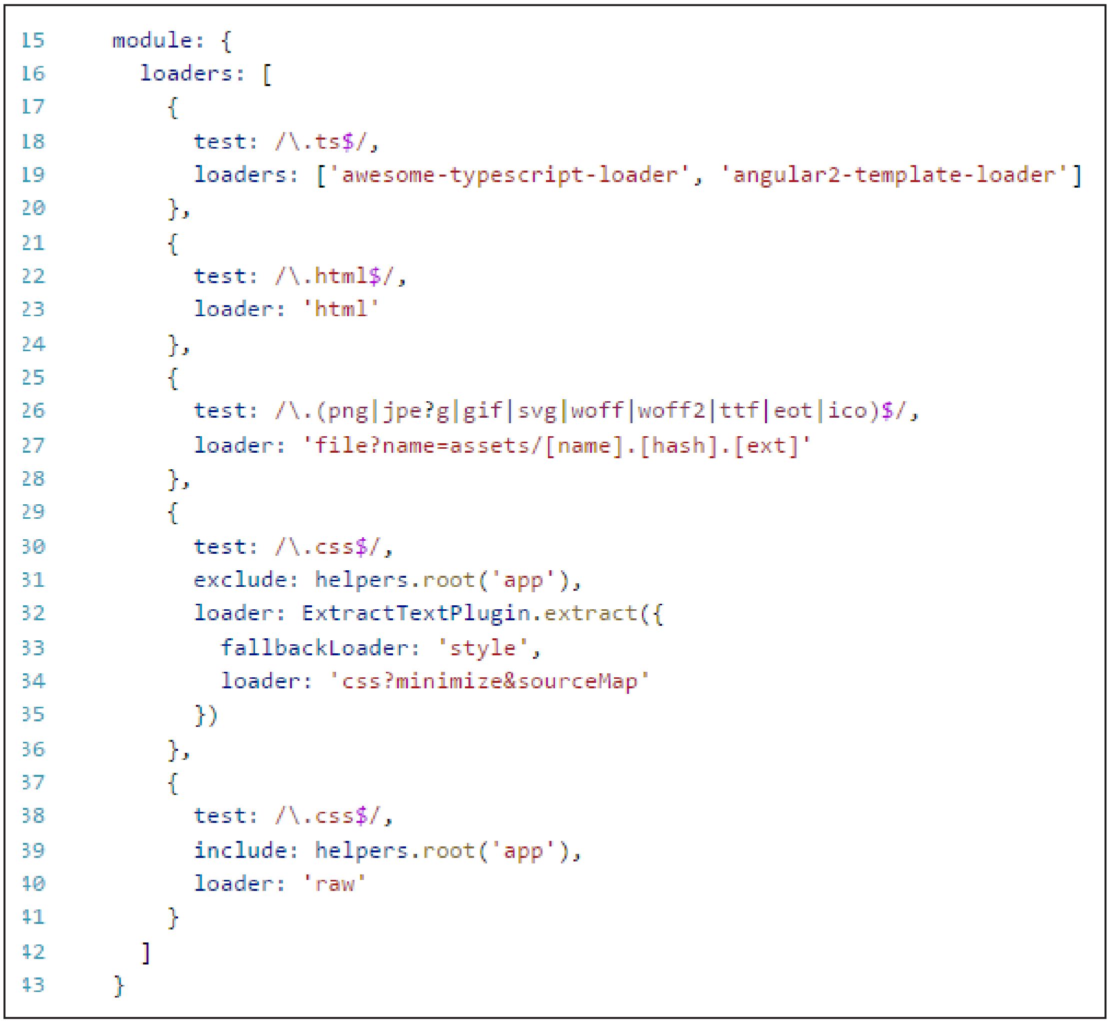
```
resolve: {
  extensions: ['', '.js', '.ts']
}
```

The most important part of Webpack to be configured is the list of loaders to be used. As different application will have different types of files to be loaded, the required set of loaders must be configured as an array. Every entry in the array has to set values to the following properties:

- **test:** A regular expression to match with the extension of the file
- **loader or loaders:** One or more loaders to be used to process the files matched by the expression specified in test
- **include:** Specific set of files in addition to the matching list to be included in the bundle
- **exclude:** Specific set of files out of the matching list to be excluded from the bundle

Listing 41.9 shows the configuration of loaders.

Listing 41.9: *Loaders configuration in webpack.common.js*



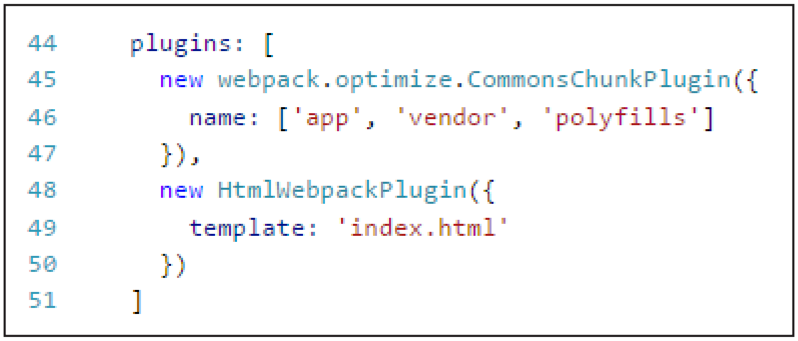
```
    module: {
      loaders: [
        {
          test: /\.ts$/,
          loaders: ['awesome-typescript-loader', 'angular2-template-loader']
        },
        {
          test: /\.html$/,
          loader: 'html'
        },
        {
          test: /\.(png|jpe?g|gif|svg|woff|woff2|ttf|eot|ico)$/,
          loader: 'file?name=assets/[name].[hash].[ext]'
        },
        {
          test: /\.css$/,
          exclude: helpers.root('app'),
          loader: ExtractTextPlugin.extract({
            fallbackLoader: 'style',
            loader: 'css?minimize&sourceMap'
          })
        },
        {
          test: /\.css$/,
          include: helpers.root('app'),
          loader: 'raw'
        }
      ]
    }
```

Listing 41.9 configures loaders for the TypeScript files, CSS files, HTML files, and the font and icon files. Following are some notable points in the loaders configuration:

- There are two loaders associated with TypeScript files. One is to transpile the TypeScript files, and the other is to load Angular 2 templates. The *angular2-template-loader* loader is passed to parse the HTML template URLs in the components, and load them to the bundle
- There are two loader configurations for the CSS files. Out of them, one loads the CSS from an external library; it uses *ExtractTextPlugin* to avoid adding of this style in the bundle and to keep it separate. The other loader injects the <link> tag to the HTML file. The second configuration to load the CSS files looks for all CSS files inside the source code, and loads them into the application bundle using the raw loader. The CSS of these files will be embedded in the JavaScript file containing the application code.

The last element to be added to the common file is the list of plugins (it requires two) to process the files. It needs the *CommonsChunkPlugin* to create the three different bundles and store them in their respective files. The *HtmlWebpackPlugin* is used to insert the required script and style tags to the HTML file.

Listing 41.10: *Plugins in the common file*

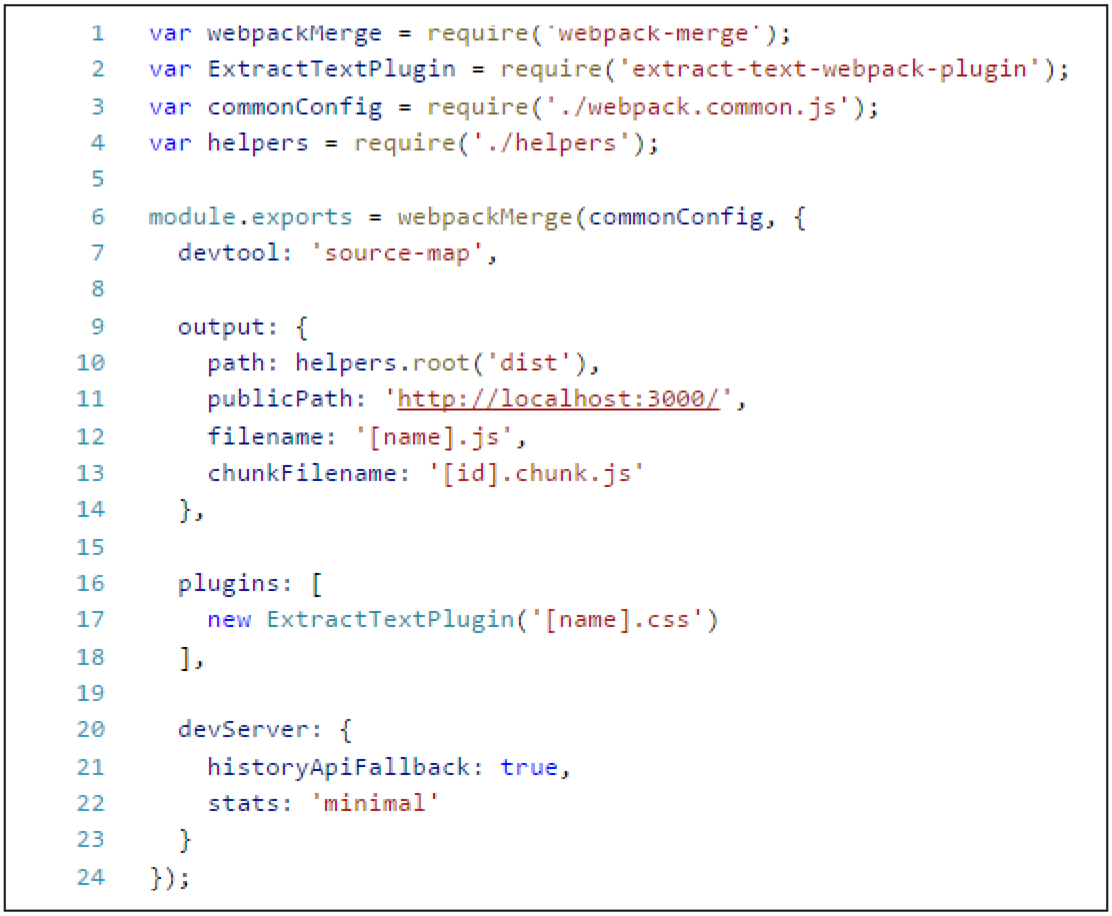


```
  plugins: [
    new webpack.optimize.CommonsChunkPlugin({
      name: ['app', 'vendor', 'polyfills']
    }),
    new HtmlWebpackPlugin({
      template: 'index.html'
    })
  ]
```

Using Webpack for Development

As previously mentioned, the application requires two different configurations for development and production. The configuration for development will store the bundle files in the memory and will start the Webpack's dev server to serve the application. Listing 41.11 shows the code of this file.

Listing 41.11: *Code of webpack.dev.js*



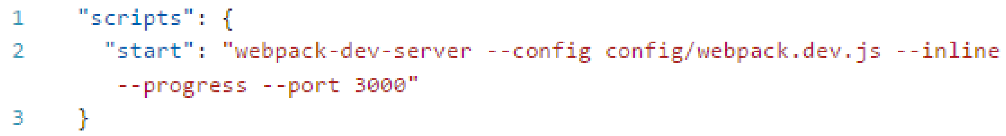
```
var webpackMerge = require('webpack-merge');
var ExtractTextPlugin = require('extract-text-webpack-plugin');
var commonConfig = require('./webpack.common.js');
var helpers = require('./helpers');

module.exports = webpackMerge(commonConfig, {
  devtool: 'source-map',

  output: {
    path: helpers.root('dist'),
    publicPath: 'http://localhost:3000/',
    filename: '[name].js',
    chunkFilename: '[id].chunk.js'
  },

  plugins: [
    new ExtractTextPlugin('[name].css')
  ],

  devServer: {
    historyApiFallback: true,
    stats: 'minimal'
  }
});
```

Clearly, it extends the configuration from the common configuration. As this is for development, it generates the source map files to ease the process of debugging. To run the application, the following command has to be executed from a command prompt:

```
> webpack-dev-server --config config/webpack.dev.js --inline --progress --port 3000
```

The above statement starts the Webpack development server on port 3000 based on the configuration set in the file *webpack.dev.js*. Open a browser and change the URL to *http://localhost:3000* to see the demo working. Remember to start the server before loading the client application on the browser.

As the *webpack-dev-server* command is too long to type, it would be better to store it in the npm scripts and use an alias to run it. Add a script section to *package.json* file and add the following statement to it:



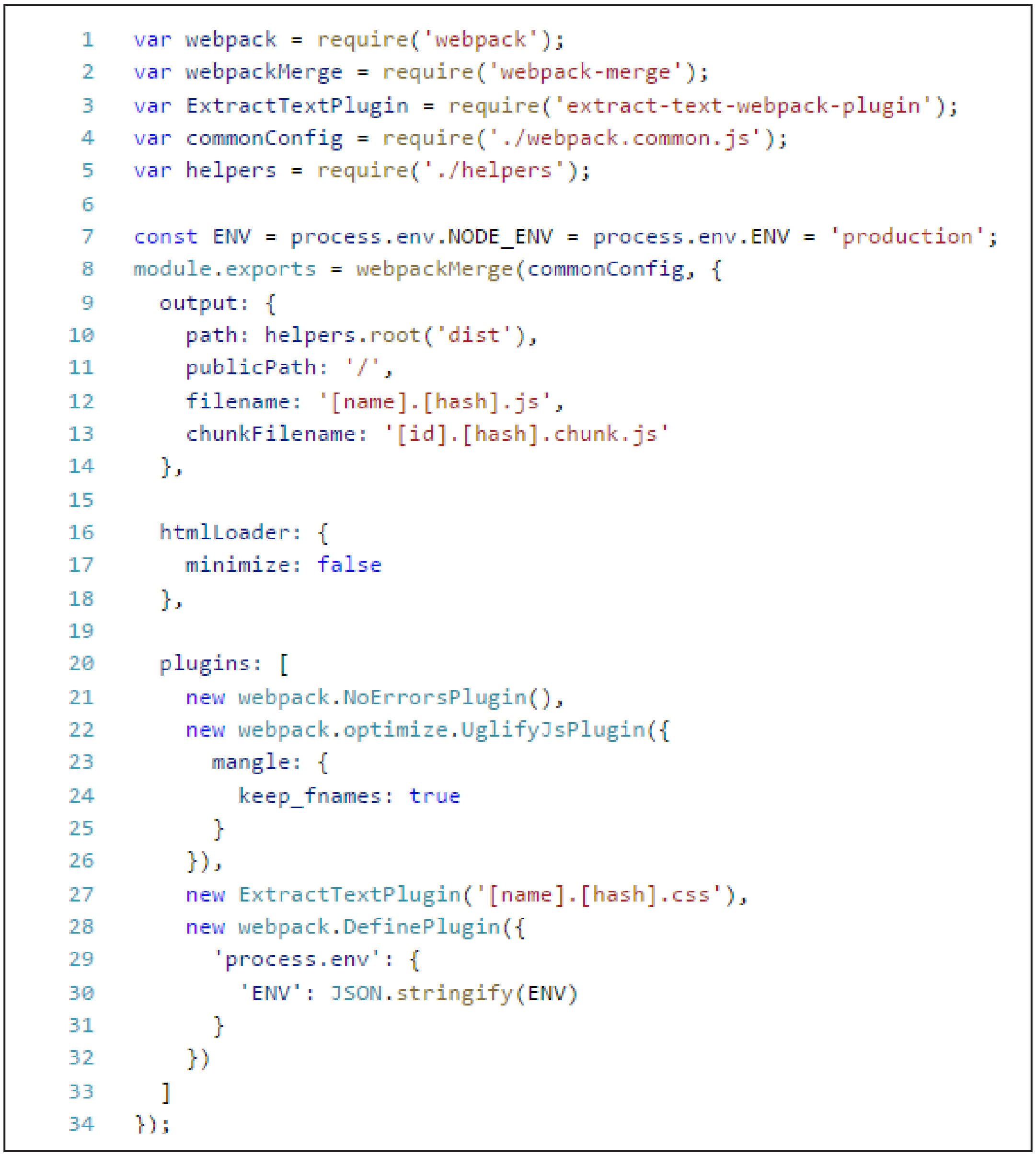
```
"scripts": {
  "start": "webpack-dev-server --config config/webpack.dev.js --inline
   --progress --port 3000"
}
```

Using Webpack for Production

The production configuration file stores the bundled files in the *dist* folder. To bust the cache of the files deployed to production, it generates a hash code for every file. The code is appended to the files so that their names will be unique after every build. This configuration doesn't generate source maps. It uses the *uglify* plugin to minify the JavaScript code generated from the libraries and the source. To ensure zero faults in the bundles, it uses the *NoErrorsPlugin*. This plugin prevents producing the assets when the bundling process encounters an error.

Add a new file to the *config* folder and name it *webpack.prod.js*. Add the following code to this file:

Listing 41.12: *Code of webpack.prod.js*



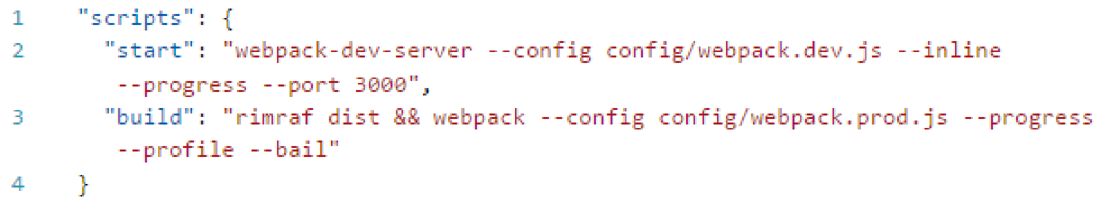
```
var webpack = require('webpack');
var webpackMerge = require('webpack-merge');
var ExtractTextPlugin = require('extract-text-webpack-plugin');
var commonConfig = require('./webpack.common.js');
var helpers = require('./helpers');

const ENV = process.env.NODE_ENV = process.env.ENV = 'production';
module.exports = webpackMerge(commonConfig, {
  output: {
    path: helpers.root('dist'),
    publicPath: '/',
    filename: '[name].[hash].js',
    chunkFilename: '[id].[hash].chunk.js'
  },

  htmlLoader: {
    minimize: false
  },

  plugins: [
    new webpack.NoErrorsPlugin(),
    new webpack.optimize.UglifyJsPlugin({
      mangle: {
        keep_fnames: true
      }
    }),
    new ExtractTextPlugin('[name].[hash].css'),
    new webpack.DefinePlugin({
      'process.env': {
        'ENV': JSON.stringify(ENV)
      }
    })
  ]
});
```

To run the bundling process, the webpack command must be supplied with the configuration file *webpack.prod.js*. Following is the command to generate the build files:

webpack --config config/webpack.prod.js --progress --profile --bail

On running this command, files are produced and and stored in the *dist* folder. These files can be used to deploy the static web application anywhere. To create an updated bundle after modifying a few of the source files, it would be best to delete the *dist* folder and generate it again. As it involves multiple steps, it would be convenient to have an npm script for it. The following is the updated script section of *package.json:*

Now the application can be built using the command:



```
"scripts": {
  "start": "webpack-dev-server --config config/webpack.dev.js --inline
    --progress --port 3000",
  "build": "rimraf dist && webpack --config config/webpack.prod.js --progress
    --profile --bail"
}
```

> npm run build

Conclusion

Webpack is one of the most widely used front-end module bundlers. As shown here, it can be used quite efficiently for both development and production

Chapter

42

Building Secure Angular 2 apps

SECURITY ISSUES ARISE FROM weaknesses on both the server and client ends. Security is one of the most vital features to be considered regarding data in an application. Single Page Applications (SPA) use significant amount of data from a server through REST APIs. Anyone with information about the URL and the format of data exposed via a REST API can easily call it using an HTTP client tool like Fiddler or Postman. Depending on the information exposed via these APIs, it is important to secure these APIs and consume them from client applications in a secured manner.

This chapter will discuss how to consume secured APIs in an Angular 2 application.

Like any secured application you may have seen, the secured parts of an SPA shouldn't be accessible unless the user logs in. Every logged in user gets a JSON Web Token (a.k.a JWT) from the server. If you are not familiar with how JWT works, you may *read this article from Auth0*. The server must send this token to a secured API with every call. The next section explains the process of setting up the Koa.js server with JWT, and using that in an Angular 2 application.

For the sake of simplicity in our example, we will assume that a user who has logged in has access to the entire application.

NOTE: With JavaScript, almost any authentication system can be compromised, simply because the code runs directly in the browser. It is always prudent to implement server-side security in addition to client-side security.

For the sample code of this chapter, the APIs have to be secured. If you want to follow along, you may download the code from the folder *before* in the downloadable content of this post. This folder contains an Angular 2 application and a Node.js server that exposes a few REST APIs. Some of these APIs are secured and their consumption in the application has to be modified. The following sections will do so.

Creating a Login Page

The application needs a login page. For this it needs:

- a component
- a route to be associated with the page
- a service to handle the logic of login/logout, as well as to provide login information to the application.

Add a new folder named *login* to the app folder. All files related to login and authentication will be stored in this folder.

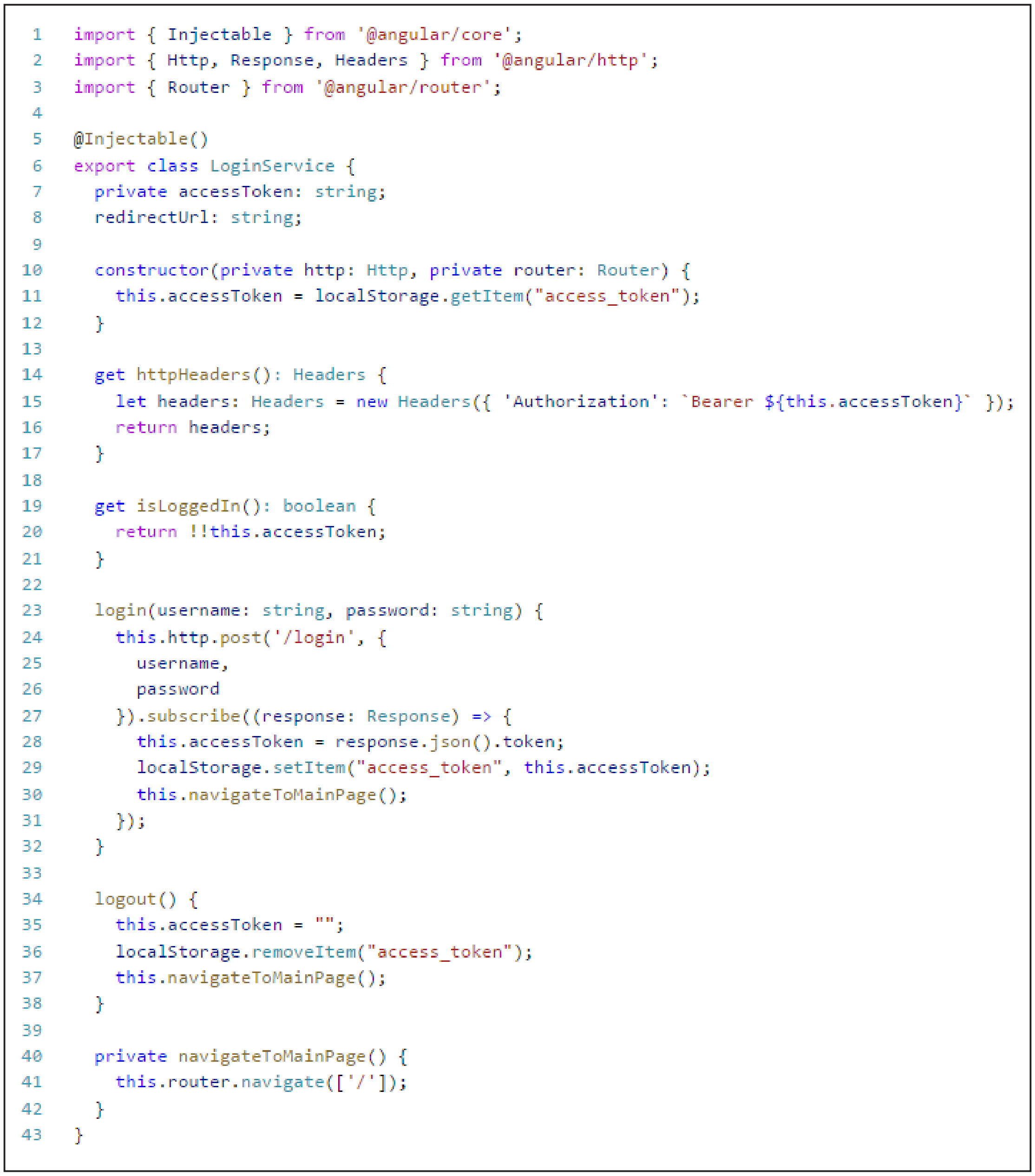
Login Service

It would be wise to store the functionality of interacting with the login API and the login information in a Service. This information will be used in other components of the application as well. Add a new file to this folder and name it *login.service.ts*. This service will perform the following tasks:

- Accept credentials and send them to the login API; store the access token received in response of the login API in a field, and store it in a local storage
- Remove the access token from local storage when a user logs out
- Serve HTTP headers, which include the JSON web token. These headers must be set to every secured API

Listing 42.1 shows the code of this service:

Listing 42.1: *Code of LoginService*

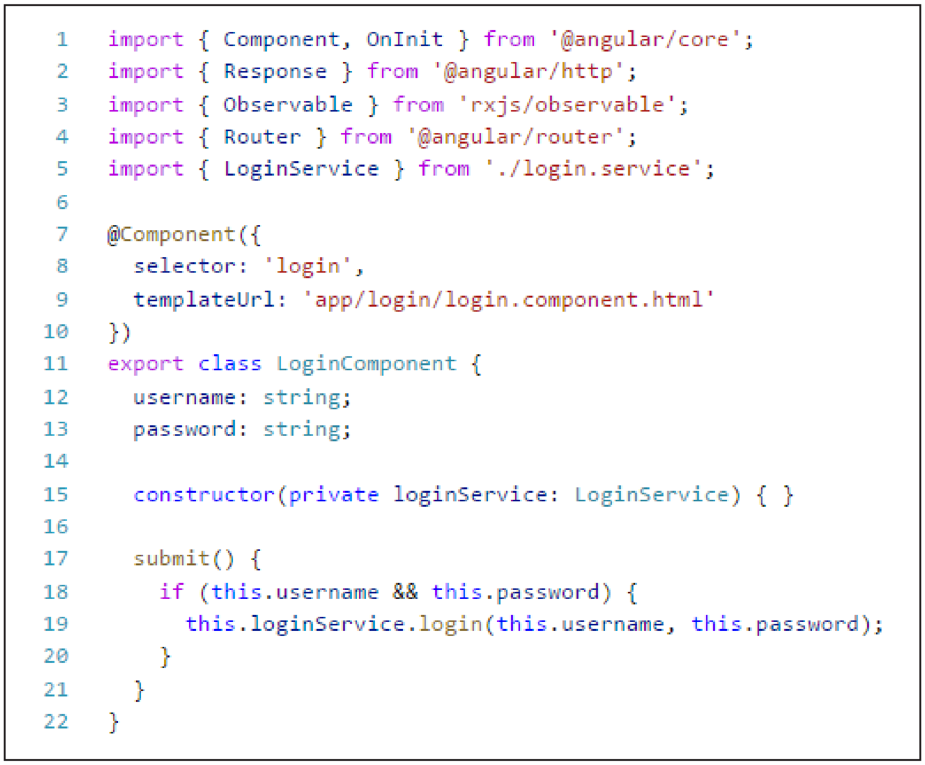


```
import { Injectable } from '@angular/core';
import { Http, Response, Headers } from '@angular/http';
import { Router } from '@angular/router';

@Injectable()
export class LoginService {
  private accessToken: string;
  redirectUrl: string;

  constructor(private http: Http, private router: Router) {
    this.accessToken = localStorage.getItem("access_token");
  }

  get httpHeaders(): Headers {
    let headers: Headers = new Headers({ 'Authorization': `Bearer ${this.accessToken}` });
    return headers;
  }

  get isLoggedIn(): boolean {
    return !!this.accessToken;
  }

  login(username: string, password: string) {
    this.http.post('/login', {
      username,
      password
    }).subscribe((response: Response) => {
      this.accessToken = response.json().token;
      localStorage.setItem("access_token", this.accessToken);
      this.navigateToMainPage();
    });
  }

  logout() {
    this.accessToken = "";
    localStorage.removeItem("access_token");
    this.navigateToMainPage();
  }

  private navigateToMainPage() {
    this.router.navigate(['/']);
  }
}
```

Login Component and Route

The login page accepts the username and password from the user and logs in the user into the application. The page consists of a component (*LoginComponent)* to carry this functionality. The *LoginComponent* makes use of the *LoginService* created in listing 42.1 to pass the credentials of the user to the API. Code of the component is straightforward, as is shown in listing 42.2.

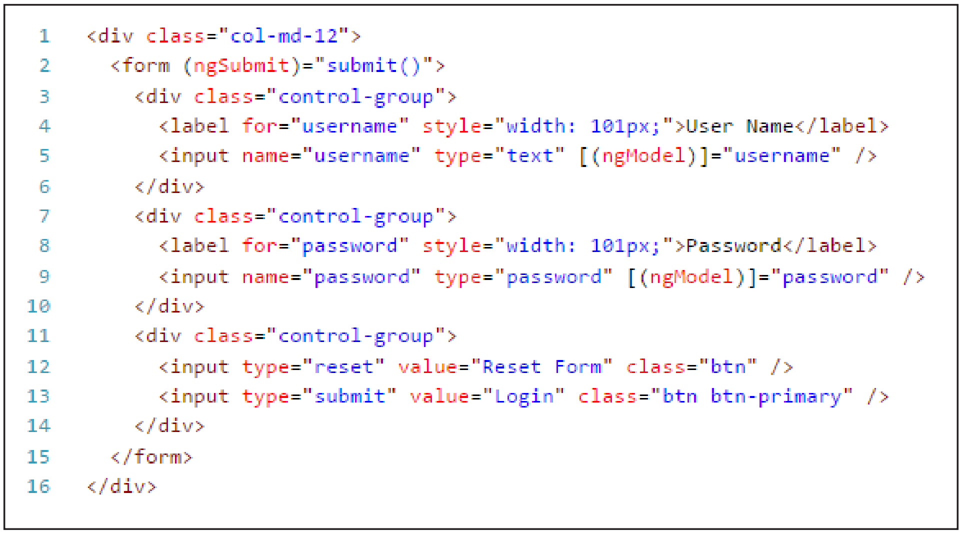
Listing 42.2: *Code of LoginComponent*



```
import { Component, OnInit } from '@angular/core';
import { Response } from '@angular/http';
import { Observable } from 'rxjs/observable';
import { Router } from '@angular/router';
import { LoginService } from './login.service';

@Component({
  selector: 'login',
  templateUrl: 'app/login/login.component.html'
})
export class LoginComponent {
  username: string;
  password: string;

  constructor(private loginService: LoginService) { }

  submit() {
    if (this.username && this.password) {
      this.loginService.login(this.username, this.password);
    }
  }
}
```

The user can enter the credentials using a form contained in the HTML template of this page. The form calls the *submit* method defined in the *LoginComponent* on submission. Listing 42.3 shows the code of the template.

Listing 42.3: *Template of LoginComponent*



```
<div class="col-md-12">
  <form (ngSubmit)="submit()">
    <div class="control-group">
      <label for="username" style="width: 101px;">User Name</label>
      <input name="username" type="text" [(ngModel)]="username" />
    </div>
    <div class="control-group">
      <label for="password" style="width: 101px;">Password</label>
      <input name="password" type="password" [(ngModel)]="password" />
    </div>
    <div class="control-group">
      <input type="reset" value="Reset Form" class="btn" />
      <input type="submit" value="Login" class="btn btn-primary" />
    </div>
  </form>
</div>
```

The component is created, but it is not reachable as the application doesn't have a route for it. The listing 42.4 shows the route for this view; it has to be added to the *Routes* array in the file *app.routes.ts*:

Listing 47.4: *Route for login page*

```
{ path: 'signin', component: LoginComponent },
```

The listing 47.5 shows the updated set of routes in the file *app.routes.ts:*

Listing 42.5: *Updated routes*

```
import { Routes, RouterModule } from '@angular/router';
import { BooksComponent } from './books.component';
import { AddBookComponent } from './addbook.component';
import { LoginComponent } from './login/login.component';

export const App_Routes: Routes = [
  { path: '', component: BooksComponent },
  { path: 'signin', component: LoginComponent },
  { path: 'addBook', component: AddBookComponent},
  { path: 'reviews/:id', loadChildren: '/app/reviews/reviews.module' },
  { path: '**', redirectTo: '', pathMatch: 'full' }
];

export const App_Routing = RouterModule.forRoot(App_Routes);
```

Run the application and change the URL to *http://localhost:3000/signin*. You will see the login page in the view. A screenshot of the login page is shown in Figure 42.1.

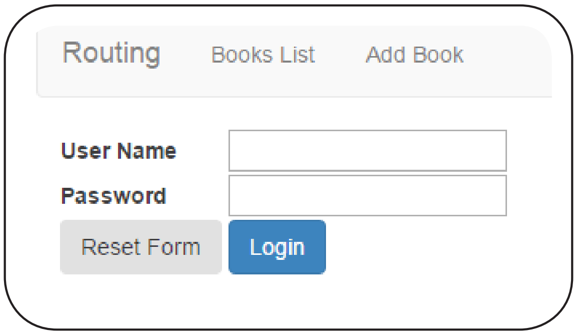


Figure 42.1: *Login page*

Upon signing in, an entry is added to local storage to store the token and redirects the user to the books list page.

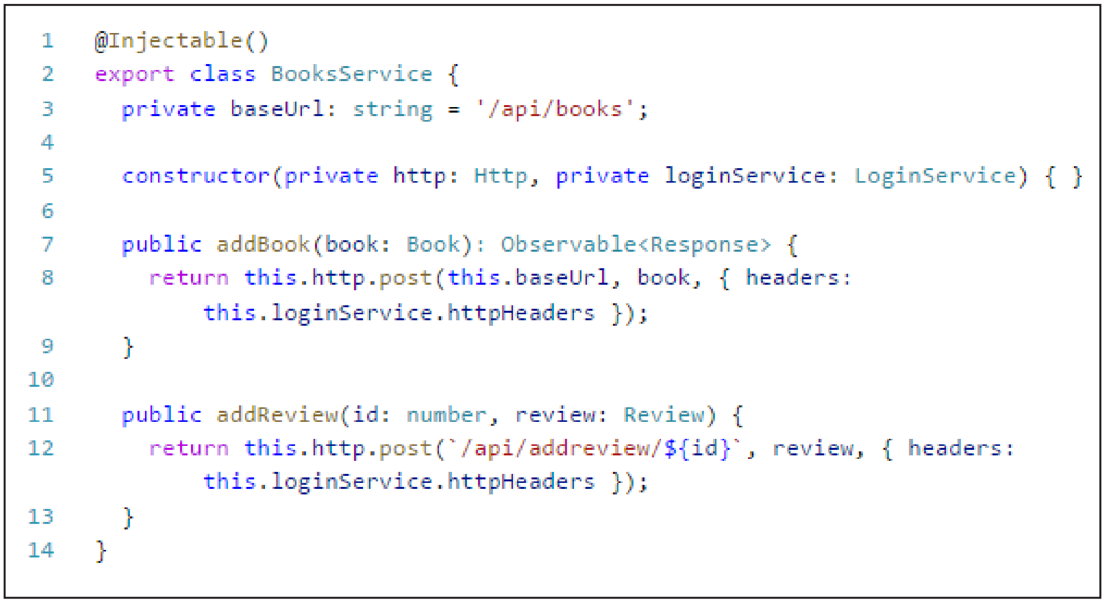
Adding Security to Some Parts of the Application

SENDING AUTHENTICATION TOKEN WITH REQUESTS

Though the application has a login page, when a logged-in user attempts to add a new book or adds a new review, the operation will fail. This is because the bearer token is not sent to the POST APIs performing these actions. The *LoginService* defined in listing 42.1 has a method returning the request headers with the bearer token. The methods of *BookService* have to be modified to use this method.

The *LoginService* has to be injected into the *BookService,* and the methods *addBook* and *addReview* have to use the service to pass the headers to the requests. Listing 42.6 shows the modified portion of the *BookService:*

Listing 42.6: *Modified portion of BookService*



```
@Injectable()
export class BooksService {
  private baseUrl: string = '/api/books';

  constructor(private http: Http, private loginService: LoginService) { }

  public addBook(book: Book): Observable<Response> {
    return this.http.post(this.baseUrl, book, { headers:
        this.loginService.httpHeaders });
  }

  public addReview(id: number, review: Review) {
    return this.http.post(`/api/addreview/${id}`, review, { headers:
        this.loginService.httpHeaders });
  }
}
```

Now the operations of adding a book and adding reviews to an existing book, will work.

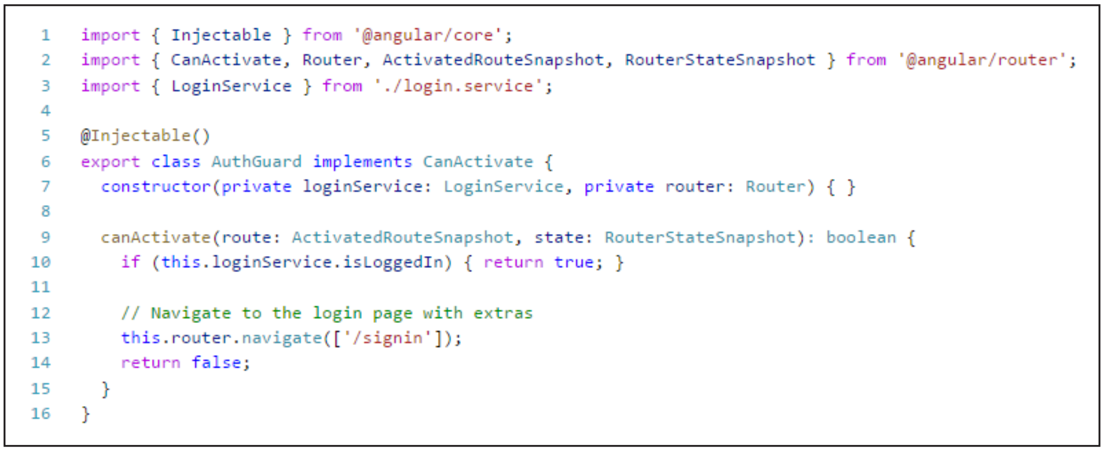
SECURING ROUTES

As the application performs some of its operations in a secured way, it is important to secure the routes performing these operations. A user shouldn't be allowed to visit these routes unless he or she is logged in. Angular 2 provides route guards to hook into different phases of the routes, and to control the corresponding phase based on the current state of the application. A guard is a service in an Angular 2 application. A guard implements a method which gets called automatically when the corresponding route is encountered.

To prevent a route from loading, the route must implement the *canActivate* guard. This guard runs when a route is about to be loaded. The service handling the guard has to implement a method named *canActivate,* as it is invoked by the router. This method can return a Boolean value or an asynchronous object resolving a Boolean value. The route loads if the return value is 'true' and it doesn't load if the return value is 'false'.

Add a new file to the folder login and name it *auth.guard.ts.* Add the following code to it:

Listing 42.7: *Code of AuthGuard*

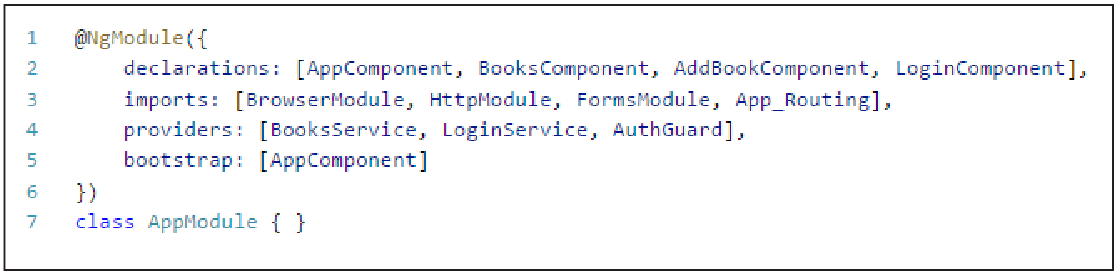


```
import { Injectable } from '@angular/core';
import { CanActivate, Router, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
import { LoginService } from './login.service';

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private loginService: LoginService, private router: Router) { }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    if (this.loginService.isLoggedIn) { return true; }

    // Navigate to the login page with extras
    this.router.navigate(['/signin']);
    return false;
  }
}
```

The class *AuthGuard* in the listing 42.7 implements the interface *CanActivate.* While it is not mandatory to implement it, it aides in accessing tooling support in Visual Studio Code for the method *canActivate*. It uses the get property *isLoggedIn* of *LoginService* to verify that the user is logged in.

This class has to be registered in the *providers* array of the module. The following snippet shows the module after adding *AuthGuard* as a provider.

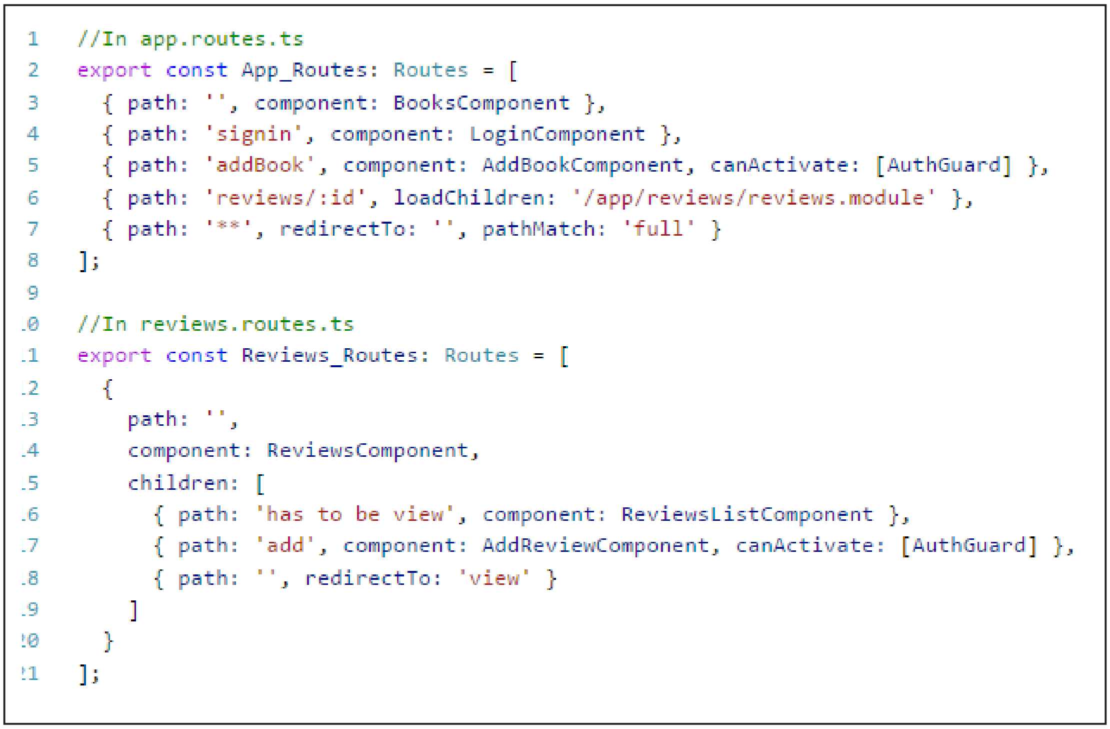
Listing 42.8: *Module with AuthGuard in providers*



```
@NgModule({
    declarations: [AppComponent, BooksComponent, AddBookComponent, LoginComponent],
    imports: [BrowserModule, HttpModule, FormsModule, App_Routing],
    providers: [BooksService, LoginService, AuthGuard],
    bootstrap: [AppComponent]
})
class AppModule { }
```

Now the guard has to be added to the routes. The following snippet shows the modified routes:

Listing 42.9: *Add book and add review routes with guards*



```
//In app.routes.ts
export const App_Routes: Routes = [
  { path: '', component: BooksComponent },
  { path: 'signin', component: LoginComponent },
  { path: 'addBook', component: AddBookComponent, canActivate: [AuthGuard] },
  { path: 'reviews/:id', loadChildren: '/app/reviews/reviews.module' },
  { path: '**', redirectTo: '', pathMatch: 'full' }
];

//In reviews.routes.ts
export const Reviews_Routes: Routes = [
  {
    path: '',
    component: ReviewsComponent,
    children: [
      { path: 'has to be view', component: ReviewsListComponent },
      { path: 'add', component: AddReviewComponent, canActivate: [AuthGuard] },
      { path: '', redirectTo: 'view' }
    ]
  }
];
```

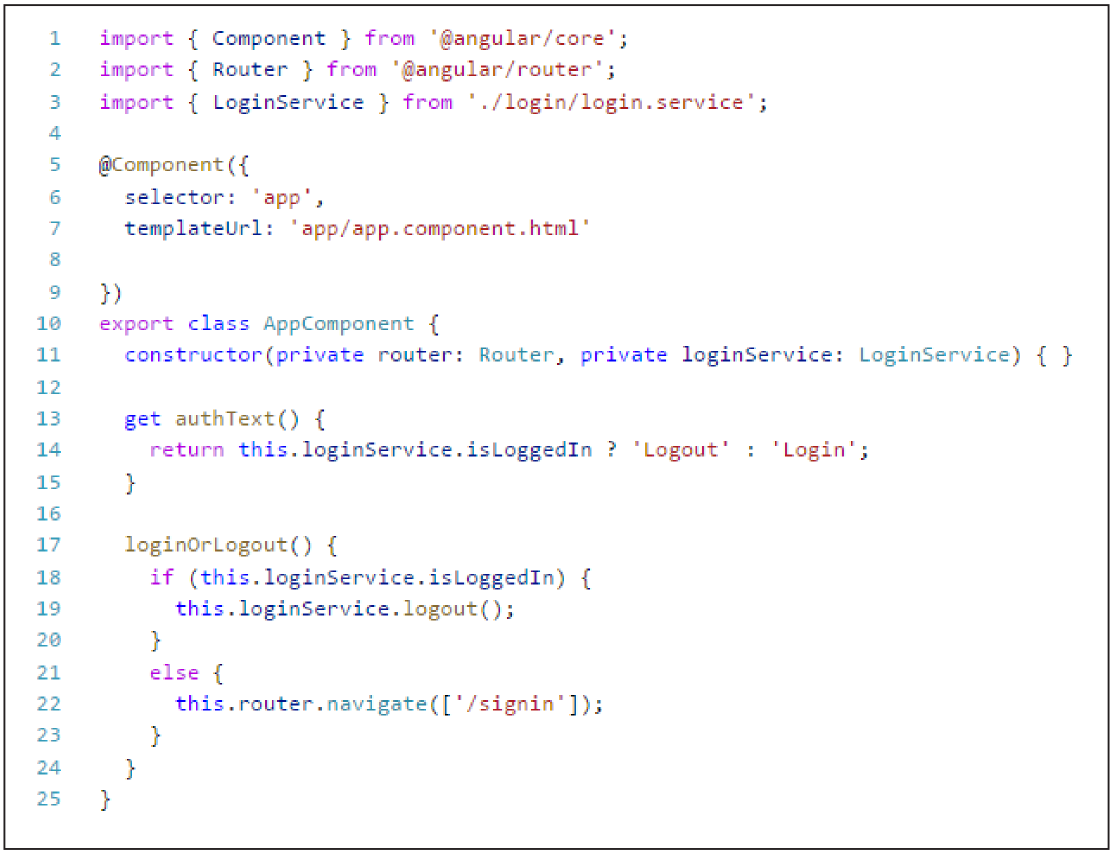
Save the files and run the application. When you try to add a book or add review pages without logging in, you will be redirected to the login page. These pages work after logging in.

Adding Login/Logout Link

As of now, the application doesn't provide a link to login or logout. It will be difficult for users to figure out that they have to login to use the app unless there is a redirection, and an option to logout. A link at the top right corner of the page would make it easy for anyone to know about this functionality. As this will be a common link for all of the pages, it has to go in the *AppComponent*.

Change the code of *AppComponent* as shown in listing 42.10:

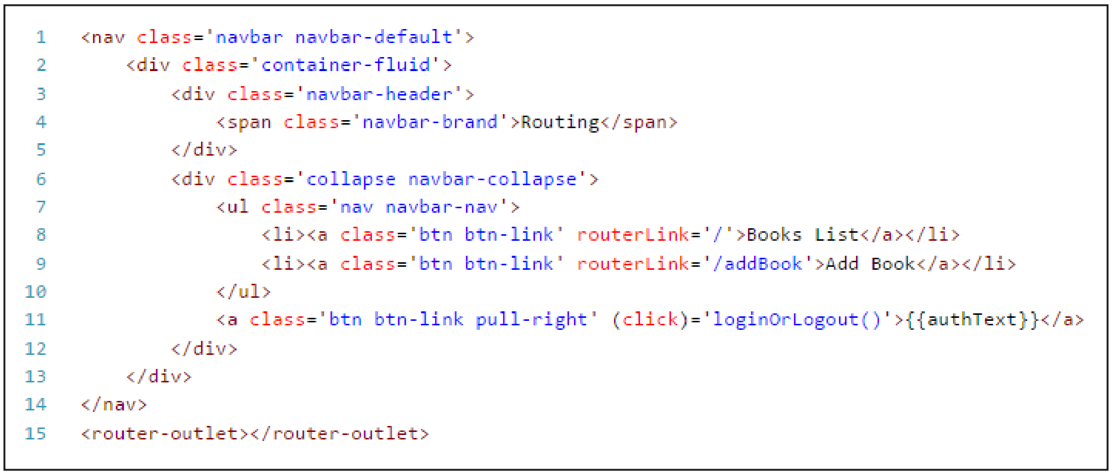
Listing 42.10: *Code of AppComponent*



```
import { Component } from '@angular/core';
import { Router } from '@angular/router';
import { LoginService } from './login/login.service';

@Component({
  selector: 'app',
  templateUrl: 'app/app.component.html'

})
export class AppComponent {
  constructor(private router: Router, private loginService: LoginService) { }

  get authText() {
    return this.loginService.isLoggedIn ? 'Logout' : 'Login';
  }

  loginOrLogout() {
    if (this.loginService.isLoggedIn) {
      this.loginService.logout();
    }
    else {
      this.router.navigate(['/signin']);
    }
  }
}
```

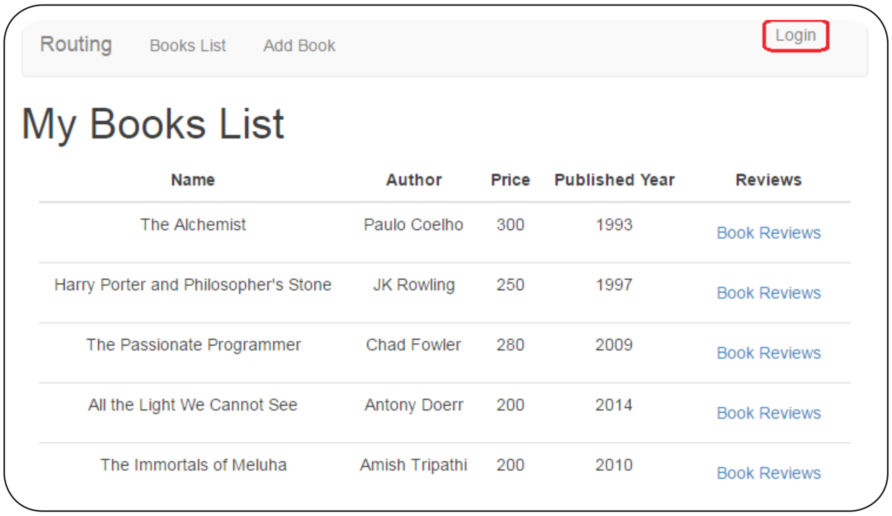
And the listing 42.11 shows the template of the *AppComponent*:

Listing 42.11: *Template of AppComponent*



```
<nav class='navbar navbar-default'>
    <div class='container-fluid'>
        <div class='navbar-header'>
            <span class='navbar-brand'>Routing</span>
        </div>
        <div class='collapse navbar-collapse'>
            <ul class='nav navbar-nav'>
                <li><a class='btn btn-link' routerLink='/'>Books List</a></li>
                <li><a class='btn btn-link' routerLink='/addBook'>Add Book</a></li>
            </ul>
            <a class='btn btn-link pull-right' (click)='loginOrLogout()'>{{authText}}</a>
        </div>
    </div>
</nav>
<router-outlet></router-outlet>
```

Save the files and run the application. The login link should appear on the top right corner of the page as shown in Figure 42.2.



My Books List

Name	Author	Price	Published Year	Reviews
The Alchemist	Paulo Coelho	300	1993	Book Reviews
Harry Porter and Philosopher's Stone	JK Rowling	250	1997	Book Reviews
The Passionate Programmer	Chad Fowler	280	2009	Book Reviews
All the Light We Cannot See	Antony Doerr	200	2014	Book Reviews
The Immortals of Meluha	Amish Tripathi	200	2010	Book Reviews

Figure 42.2: *Home page with login link*

Once a user is logged in, text of the link should change to Logout.

And that’s it. You have successfully created a secured Angular 2 application that requires the user to login before performing any operations.

Conclusion

Security is one of the primary needs of any application. Angular 2 provides enough support to secure rich applications built on the framework.

Chapter 43

Angular 1 to Angular 2 Cheatsheet

THE PREVIOUS CHAPTERS EXPLORED different features of Angular 2, starting from a "Hello World" kind of an application to a secured Angular 2 application, using most of the features of Angular 2. Angular 2 implements the concepts of Angular 1 in a new way and most of the code blocks of Angular 1 can be mapped to their corresponding versions in Angular 2.

This chapter presents a convenient cheat sheet comparing the way the features are implemented in both versions of the framework.

> **NOTE:** The Angular 1 code snippets in this chapter are in JavaScript, and the Angular 2 code snippets are in TypeScript. The choice of language is based on popularity of the language for the corresponding framework.

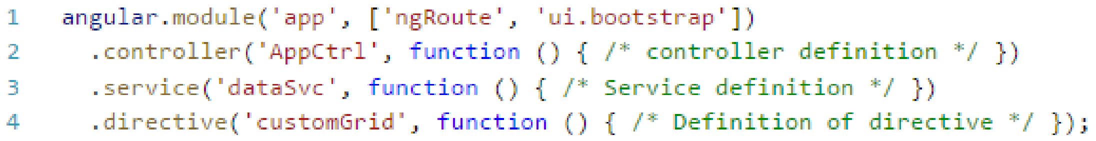
> **NOTE:** After Angular 2 announcement, the library previously known as AngularJS was renamed to Angular. Hence referred to as Angular 1.

Modules

Both Angular 1 and Angular 2 have a module system for the framework.

ANGULAR 1

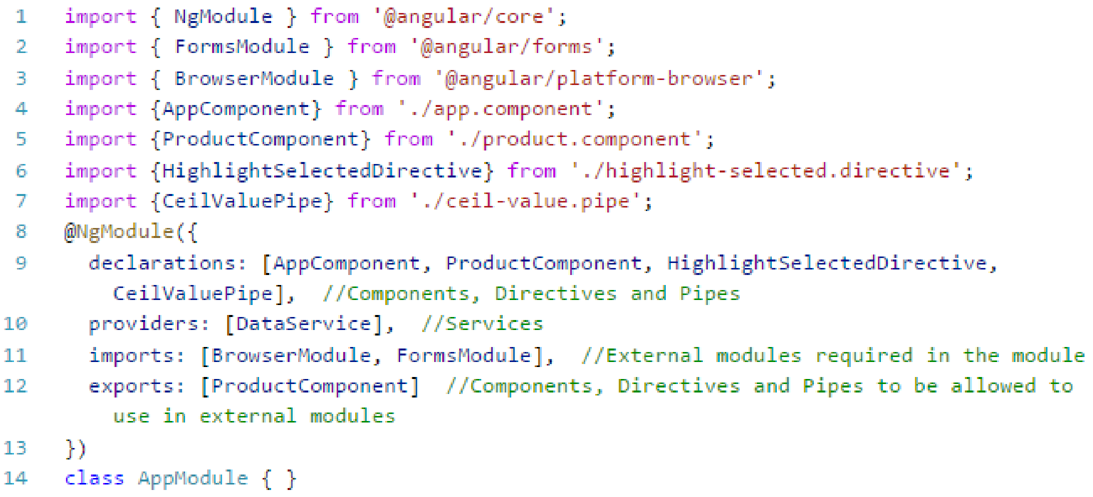
Modules in Angular 1 are used for grouping a related set of controllers, services, factories, directives, and other code blocks. A module can depend on one or more existing modules. An Angular 1 module is defined using the *angular.module* method.



```
angular.module('app', ['ngRoute', 'ui.bootstrap'])
  .controller('AppCtrl', function () { /* controller definition */ })
  .service('dataSvc', function () { /* Service definition */ })
  .directive('customGrid', function () { /* Definition of directive */ });
```

ANGULAR 2

Modules in Angular 2 are used to group a set of related components, directives, pipes, services, and other blocks. A module can use the functionality in other modules. To do so, the other modules must be imported into the metadata definition of the main module. A module can export a set of declarations to make them visible to other modules.



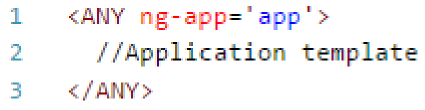
```
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';
import {AppComponent} from './app.component';
import {ProductComponent} from './product.component';
import {HighlightSelectedDirective} from './highlight-selected.directive';
import {CeilValuePipe} from './ceil-value.pipe';
@NgModule({
  declarations: [AppComponent, ProductComponent, HighlightSelectedDirective,
    CeilValuePipe],  //Components, Directives and Pipes
  providers: [DataService],  //Services
  imports: [BrowserModule, FormsModule],  //External modules required in the module
  exports: [ProductComponent]  //Components, Directives and Pipes to be allowed to
    use in external modules
})
class AppModule { }
```

Bootstrapping

To use either of the versions of the framework, the page must be bootstrapped using the root module of the application.

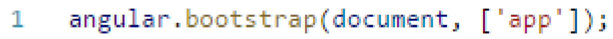
ANGULAR 1

An Angular 1 application can be bootstrapped automatically by using the *ng-app* directive on any of the HTML elements, or it can be manually bootstrapped using the *angular.bootstrap* function.



```
<ANY ng-app='app'>
  //Application template
</ANY>
```

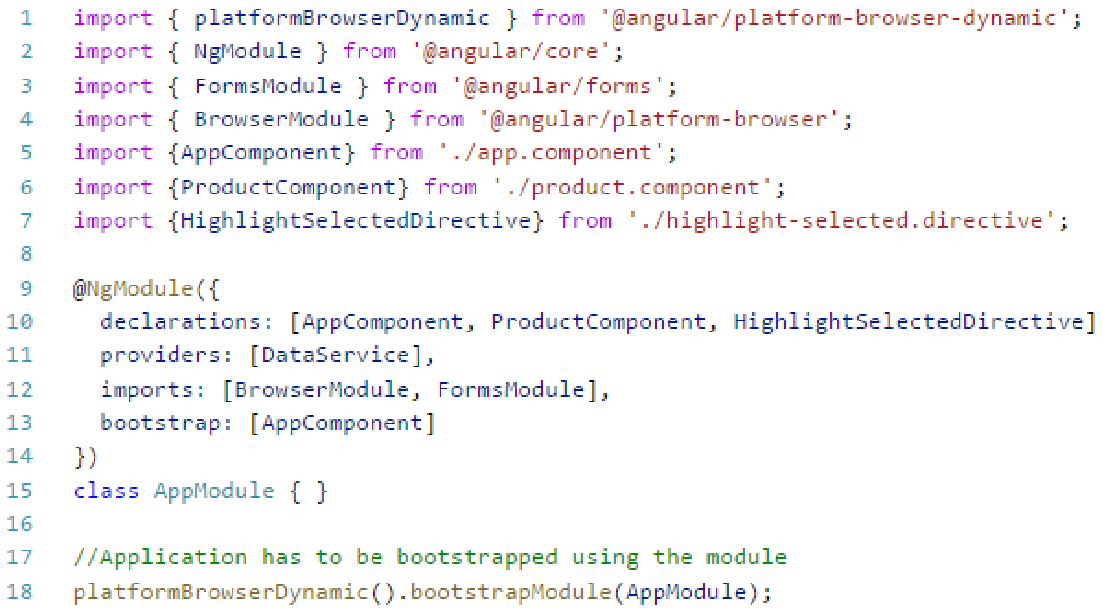
Or,



```
angular.bootstrap(document, ['app']);
```

ANGULAR 2

The main module of an Angular 2 application should have at least one component specified in the *bootstrap* property of the module metadata. The HTML page of the application must contain these components. The main module has to be passed to the platform specific bootstrap function. For browsers, it is *platformBrowserDynamic().bootstrapModule*.



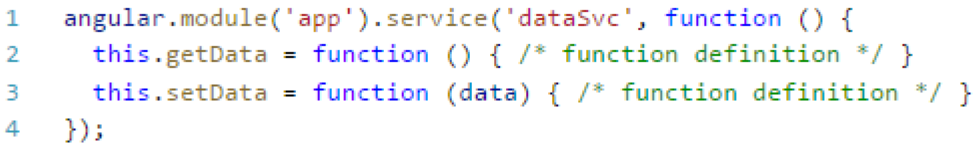
```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';
import {AppComponent} from './app.component';
import {ProductComponent} from './product.component';
import {HighlightSelectedDirective} from './highlight-selected.directive';

@NgModule({
  declarations: [AppComponent, ProductComponent, HighlightSelectedDirective]
  providers: [DataService],
  imports: [BrowserModule, FormsModule],
  bootstrap: [AppComponent]
})
class AppModule { }

//Application has to be bootstrapped using the module
platformBrowserDynamic().bootstrapModule(AppModule);
```

Services

Services are singleton objects that can hold a functionality and which can be used independently anywhere in a module.

ANGULAR 1

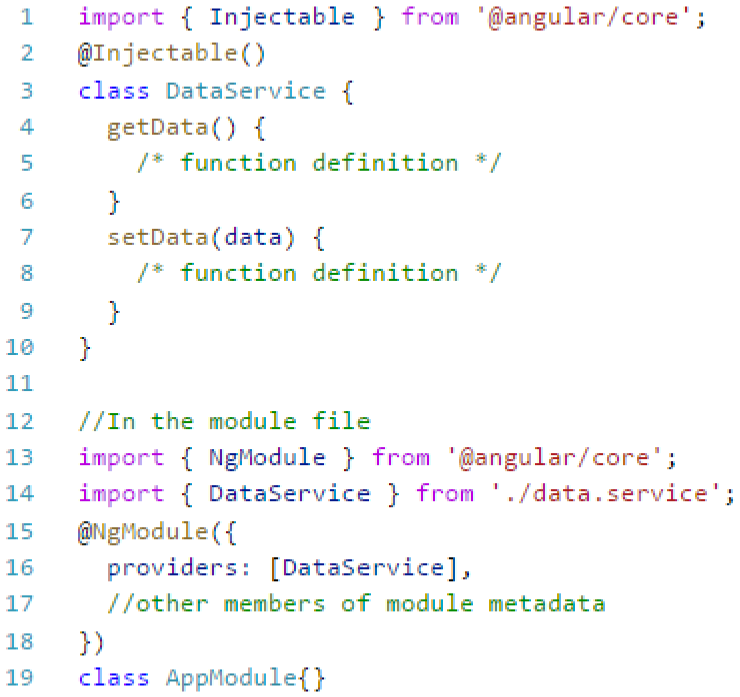
Services in Angular 1 are JavaScript constructor functions. They have to be registered using the *service* method on a module object.



```
angular.module('app').service('dataSvc', function () {
  this.getData = function () { /* function definition */ }
  this.setData = function (data) { /* function definition */ }
});
```

ANGULAR 2

Services in Angular 2 are TypeScript classes. They have to be registered in the *providers* section of a module metadata.



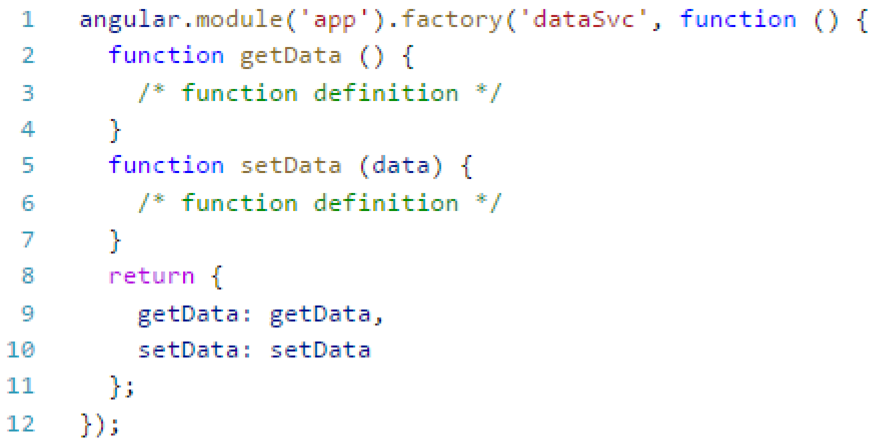
```
import { Injectable } from '@angular/core';
@Injectable()
class DataService {
  getData() {
    /* function definition */
  }
  setData(data) {
    /* function definition */
  }
}

//In the module file
import { NgModule } from '@angular/core';
import { DataService } from './data.service';
@NgModule({
  providers: [DataService],
  //other members of module metadata
})
class AppModule{}
```

Factories

Factories are similar to services. A factory is a function providing an object. The factory function is called only once, and the object returned by this function is served whenever the factory is injected somewhere.

ANGULAR 1

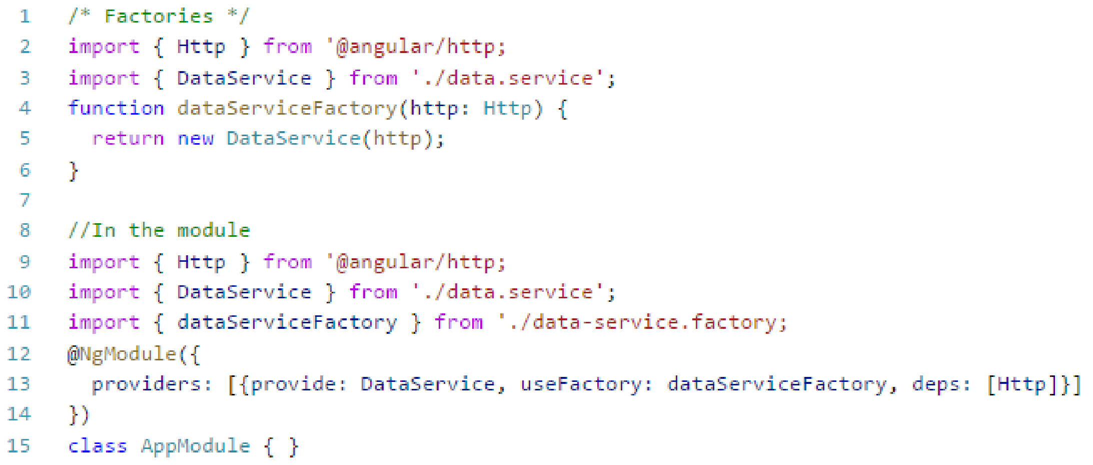
Factories in Angular 1 are registered using the *factory* method on the module object. A factory is a simple function returning the object.



```
angular.module('app').factory('dataSvc', function () {
  function getData () {
    /* function definition */
  }
  function setData (data) {
    /* function definition */
  }
  return {
    getData: getData,
    setData: setData
  };
});
```

ANGULAR 2

In Angular 2, a factory is a function that has to be registered on the module using the *useFactory* property.



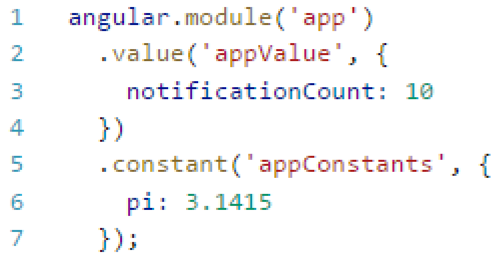
```
/* Factories */
import { Http } from '@angular/http;
import { DataService } from './data.service';
function dataServiceFactory(http: Http) {
  return new DataService(http);
}

//In the module
import { Http } from '@angular/http;
import { DataService } from './data.service';
import { dataServiceFactory } from './data-service.factory;
@NgModule({
  providers: [{provide: DataService, useFactory: dataServiceFactory, deps: [Http]}]
})
class AppModule { }
```

Values or Constants

Values and constants represent simple JavaScript objects, which don't depend on any other code block of the module.

ANGULAR 1

In Angular 1, the constants and values have to be added to the module. Constants can be injected anywhere including providers and config blocks. Values are not available in configuration phase, so they can't be injected in the config and run blocks.

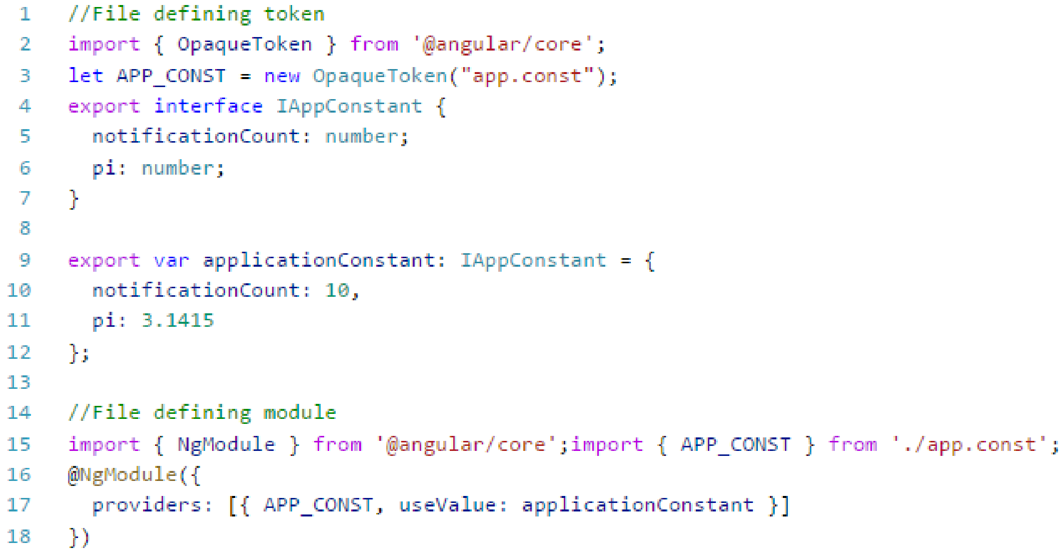


```
angular.module('app')
  .value('appValue', {
    notificationCount: 10
  })
  .constant('appConstants', {
    pi: 3.1415
  });
```

ANGULAR 2

Any value can be made injectable in Angular 2 using *OpaqueToken*. The value has to be registered using an *OpaqueToken*. The value can be made constant using the *const* keyword.

Components (Element directives in Angular 1)

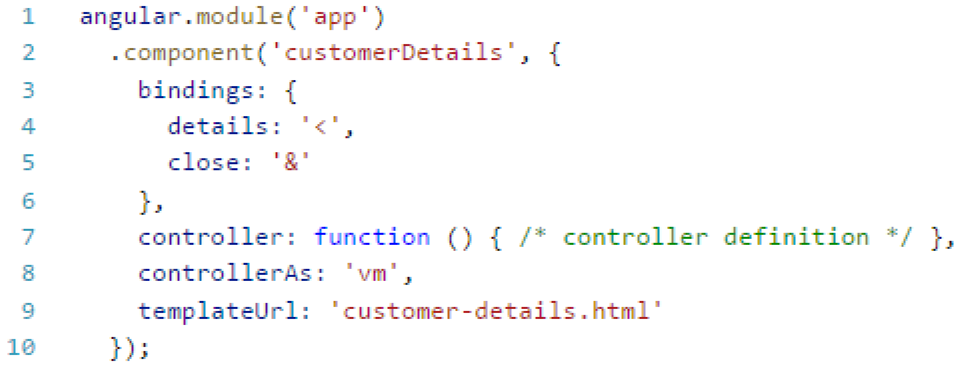


```
//File defining token
import { OpaqueToken } from '@angular/core';
let APP_CONST = new OpaqueToken("app.const");
export interface IAppConstant {
  notificationCount: number;
  pi: number;
}

export var applicationConstant: IAppConstant = {
  notificationCount: 10,
  pi: 3.1415
};

//File defining module
import { NgModule } from '@angular/core';import { APP_CONST } from './app.const';
@NgModule({
  providers: [{ APP_CONST, useValue: applicationConstant }]
})
```

Components are used to create custom HTML elements. These elements are highly reusable and therefore can be easily distributed anywhere in the application. Components also make the HTML more readable and predictable.

ANGULAR 1

A component in Angular 1 is similar to the element directives albeit with a few restrictions. It can accept inputs using *bindings* option in the component definition. The name of the component has to be in camel-cased notation and in HTML it has to be used using the dashed notation.

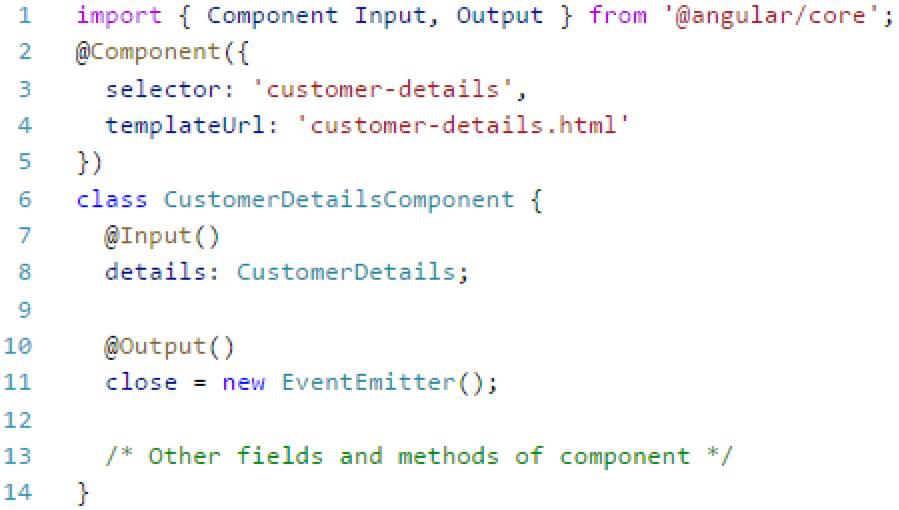


```
angular.module('app')
  .component('customerDetails', {
    bindings: {
      details: '<',
      close: '&'
    },
    controller: function () { /* controller definition */ },
    controllerAs: 'vm',
    templateUrl: 'customer-details.html'
  });
```

ANGULAR 2

An Angular 2 component is a TypeScript class with the *Component* metadata. A selector of the component has to be assigned with the same name as it would be in the HTML.

The components can interact with the containing page using the *Input* properties and *Output* events.



```
import { Component Input, Output } from '@angular/core';
@Component({
  selector: 'customer-details',
  templateUrl: 'customer-details.html'
})
class CustomerDetailsComponent {
  @Input()
  details: CustomerDetails;

  @Output()
  close = new EventEmitter();

  /* Other fields and methods of component */
}
```

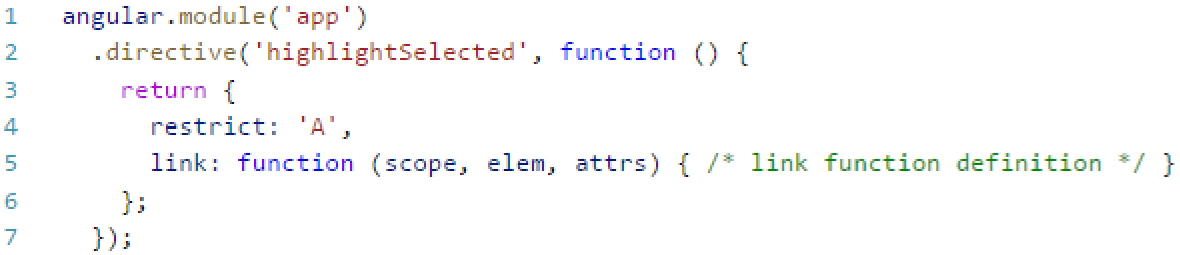
Directives (Attribute directives in Angular 1)

Directives are used to teach new tricks to HTML and to enhance the functionality of HTML elements and components.

ANGULAR 1

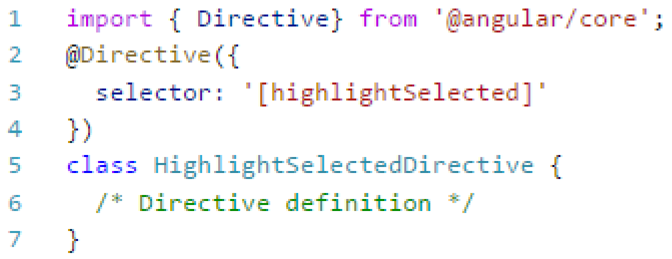
Angular 1's directives must be created using the *directive* method on the module. The directive consists of a directive definition object (DDO), which provides a number of properties to control the way a directive has to work.

ANGULAR 2



```
angular.module('app')
  .directive('highlightSelected', function () {
    return {
      restrict: 'A',
      link: function (scope, elem, attrs) { /* link function definition */ }
    };
  });
```

Directives in Angular 2 are TypeScript classes with the *Directive* decorator applied on them. The selector property of the directive metadata allows restricting the directive to be used as an attribute (an attribute on a given HTML element or a component).



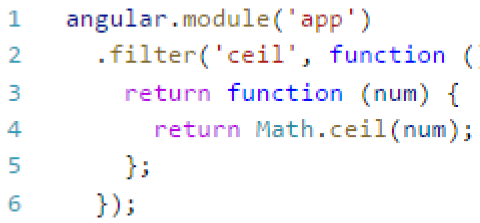
```
import { Directive} from '@angular/core';
@Directive({
  selector: '[highlightSelected]'
})
class HighlightSelectedDirective {
  /* Directive definition */
}
```

Filters (Pipes in Angular 2)

Angular has a rich binding system in both versions of the framework. Filters or pipes provide a way to format the data before displaying it on the page.

ANGULAR 1

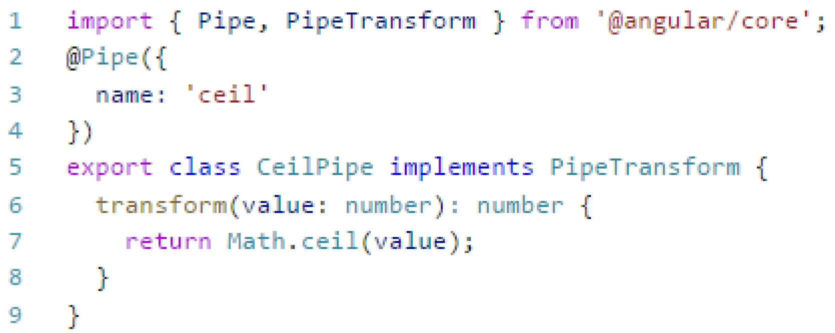
Angular 1 provides a number of built-in filters and it provides an API to write custom filters. The filters have to be defined using the *filter* method on the module object.



```
angular.module('app')
  .filter('ceil', function () {
    return function (num) {
      return Math.ceil(num);
    };
  });
```

ANGULAR 2

Custom pipes in Angular 2 are TypeScript classes with a *Pipe* decorator. The pipe must implement the *transform* method, which has to format the data and return it for binding.



```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'ceil'
})
export class CeilPipe implements PipeTransform {
  transform(value: number): number {
    return Math.ceil(value);
  }
}
```

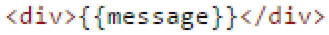
Data Binding: Value, Property, Event, Two-way

Data Binding is one of the most useful features in Angular. The data binding system makes it possible to change the view whenever there is a change in the model value without having to write even a single line of code.

VALUE BINDING

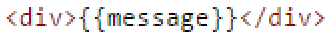
Value binding is used to bind simple values on the pages.

ANGULAR 1



```
<div>{{message}}</div>
```

ANGULAR 2



```
<div>{{message}}</div>
```

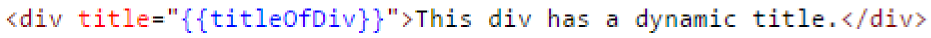
`(Same as Angular 1)`

PROPERTY BINDING

Property binding is used to bind values in the view model to properties on the HTML elements, or components.

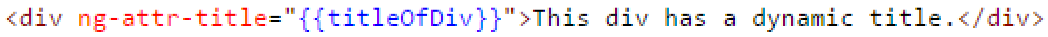
ANGULAR 1

Angular 1 has a number of built-in directives to bind values to the properties.



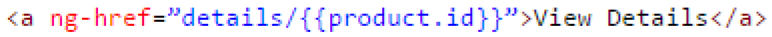
```
<div title="{{titleOfDiv}}">This div has a dynamic title.</div>
```

Or,



```
<div ng-attr-title="{{titleOfDiv}}">This div has a dynamic title.</div>
```

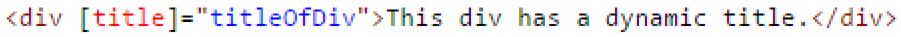
Or, using directives defined for the attributes, like:



```
<a ng-href=”details/{{product.id}}”>View Details</a>
```

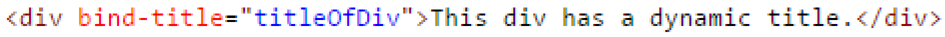
ANGULAR 2

Property binding in Angular 2 is based on a special binding syntax that can be used with any valid DOM property.



```
<div [title]="titleOfDiv">This div has a dynamic title.</div>
```

Or,



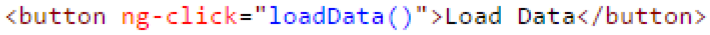
```
<div bind-title="titleOfDiv">This div has a dynamic title.</div>
```

EVENT BINDING

Event binding makes it possible to bind methods in the view model with the events on the HTML elements.

ANGULAR 1

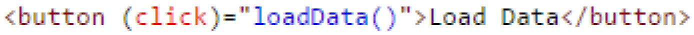
Like property binding, Angular 1 has a number of built-in directives to bind the events.



```
<button ng-click="loadData()">Load Data</button>
```

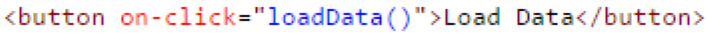
ANGULAR 2

Angular 2's event binding works on the DOM events defined on the target element. Any event can be bound with a method in the view model using the parentheses around the event name.



```
<button (click)="loadData()">Load Data</button>
```

Or,



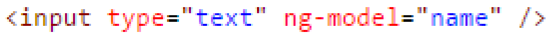
```
<button on-click="loadData()">Load Data</button>
```

TWO-WAY BINDING

Two-way binding keeps the data in the model and the bound value, in sync.

ANGULAR 1

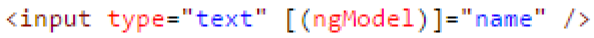
The directive *ng-model* in Angular 1 provides the feature of two-way binding. The textbox shown in the following snippet keeps the value in the textbox in sync with the property value in the view model.



```
<input type="text" ng-model="name" />
```

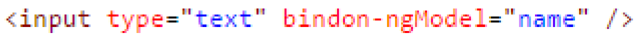
ANGULAR 2

Angular 2 provides the directive *ngModel* for two-way binding. Two-way binding in Angular 2 is a combination of property and event binding, hence the notation has both square brackets and parentheses.



```
<input type="text" [(ngModel)]="name" />
```

Or,



```
<input type="text" bindon-ngModel="name" />
```

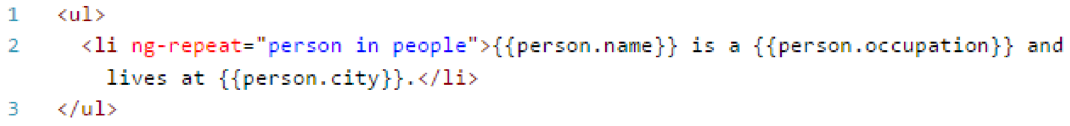
Commonly used Template Blocks

REPEATER STATEMENTS

ANGULAR 1

Angular 1 provides the directive *ng-repeat* to iterate over a set of items in the template. Every instance in the repeater gets a record object and this object is used for binding data on the page.

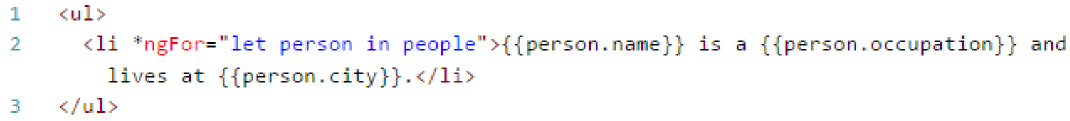
ANGULAR 2



```
<ul>
  <li ng-repeat="person in people">{{person.name}} is a {{person.occupation}} and
    lives at {{person.city}}.</li>
</ul>
```

Angular 2 provides the structural directive *ngFor* to repeat a piece of content in HTML. It is similar to *ng-repeat* in Angular 1, except it uses the ES6 like syntax in the directive.

CONDITIONALLY DISPLAYING OR HIDING A PORTION OF THE PAGE

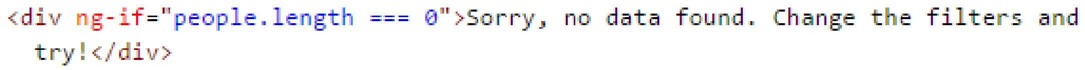


```
<ul>
  <li *ngFor="let person in people">{{person.name}} is a {{person.occupation}} and lives at {{person.city}}.</li>
</ul>
```

ANGULAR 1

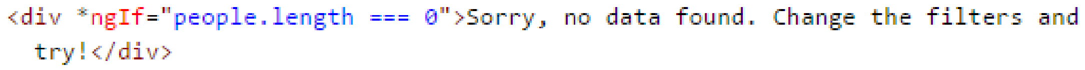
Angular 1's *ng-if* directive can be used to optionally add elements to or remove elements from the page based on the value of a data item.

ANGULAR 2



```
<div ng-if="people.length === 0">Sorry, no data found. Change the filters and try!</div>
```

Angular 2 provides the structural directive *ngIf* to add elements to or remove elements from the DOM of a page dynamically.



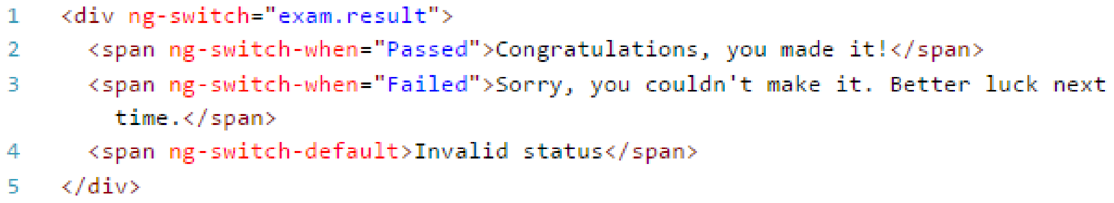
```
<div *ngIf="people.length === 0">Sorry, no data found. Change the filters and try!</div>
```

SWITCHING CONTENT

ANGULAR 1

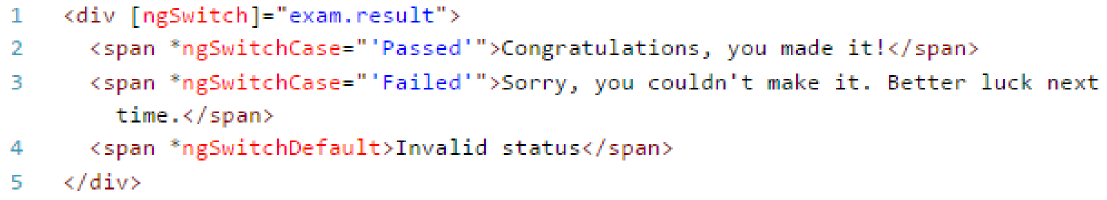
The *ng-switch* directive in Angular 1 allows switching among a set of elements, depending on the value assigned to the model. Every switchable element should have the *ng-switch-when* directive with the value of the model assigned to it. The *ng-switch-default* directive has to be used as fallback.

ANGULAR 2



```
<div ng-switch="exam.result">
  <span ng-switch-when="Passed">Congratulations, you made it!</span>
  <span ng-switch-when="Failed">Sorry, you couldn't make it. Better luck next
    time.</span>
  <span ng-switch-default>Invalid status</span>
</div>
```

Angular 2 provides with the directives *ngSwitch* and *ngSwitchCase* to switch content among a list of elements. *ngSwitchDefault* has to be used for handling the default case.



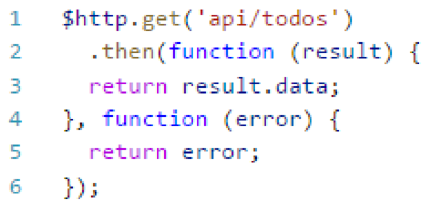
```
<div [ngSwitch]="exam.result">
  <span *ngSwitchCase="'Passed'">Congratulations, you made it!</span>
  <span *ngSwitchCase="'Failed'">Sorry, you couldn't make it. Better luck next
    time.</span>
  <span *ngSwitchDefault>Invalid status</span>
</div>
```

HTTP Calls

Both versions of Angular provide abstractions around XHR to call REST APIs.

ANGULAR 1

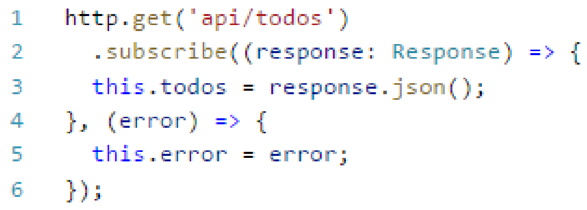
The *$http* service is the abstraction around XHR in Angular 1. It is based on promises and returns a *$q* promise in response. The promise can be used to check for success or failure of the API call.



```
$http.get('api/todos')
  .then(function (result) {
  return result.data;
}, function (error) {
  return error;
});
```

ANGULAR 2

Angular 2 has the module *HttpModule* containing the utilities to call REST APIs. The *Http* service is the abstraction around XHR. Unlike Angular 1, it returns an RxJS *Observable* object, which can be used to handle success and failures of the REST API.



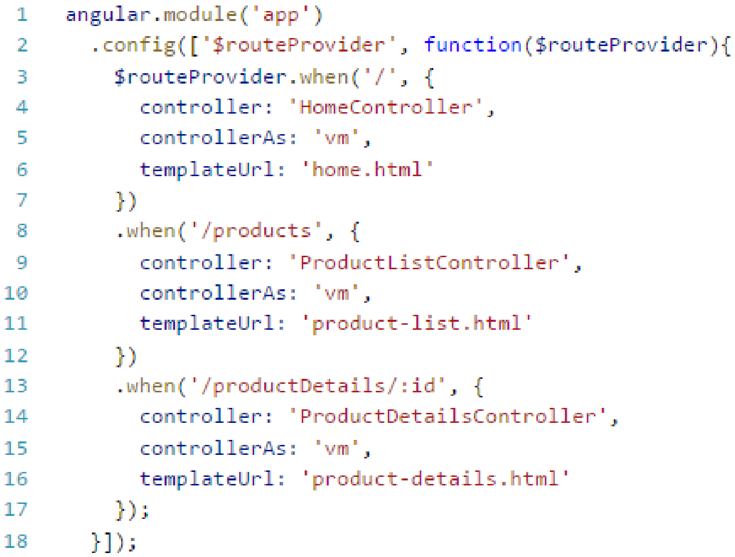
```
http.get('api/todos')
  .subscribe((response: Response) => {
  this.todos = response.json();
}, (error) => {
  this.error = error;
});
```

Routing

CONFIGURING ROUTES

ANGULAR 1

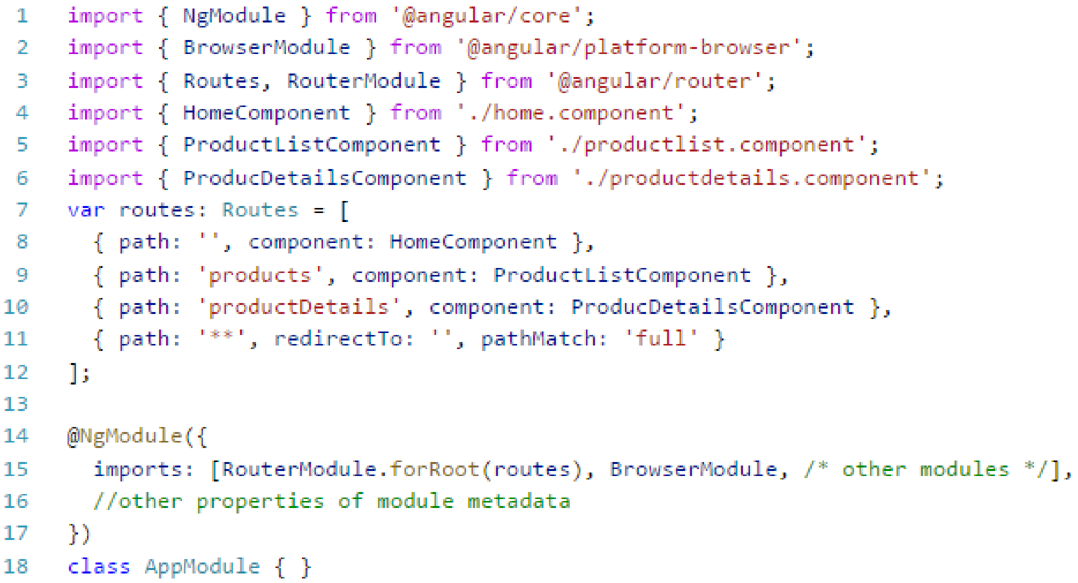
Routes in Angular 1 have to be configured in the *config* method of the module object. Most of the routers targeting Angular 1 have a provider, which must be injected to the config block to configure the routes. Every route consists of a controller and a template.



```
angular.module('app')
  .config(['$routeProvider', function($routeProvider){
    $routeProvider.when('/', {
      controller: 'HomeController',
      controllerAs: 'vm',
      templateUrl: 'home.html'
    })
    .when('/products', {
      controller: 'ProductListController',
      controllerAs: 'vm',
      templateUrl: 'product-list.html'
    })
    .when('/productDetails/:id', {
      controller: 'ProductDetailsController',
      controllerAs: 'vm',
      templateUrl: 'product-details.html'
    });
  }]);
```

ANGULAR 2

Routes in Angular 2 are a set of objects in an Array. At the minimum, every route needs a URL template and a component. A module must be created using the routes array and it has to be imported in the application module.

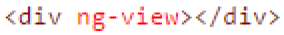


```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home.component';
import { ProductListComponent } from './productlist.component';
import { ProducDetailsComponent } from './productdetails.component';
var routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'products', component: ProductListComponent },
  { path: 'productDetails', component: ProducDetailsComponent },
  { path: '**', redirectTo: '', pathMatch: 'full' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes), BrowserModule, /* other modules */],
  //other properties of module metadata
})
class AppModule { }
```

PLACEHOLDER IN VIEW FOR ROUTES

ANGULAR 1

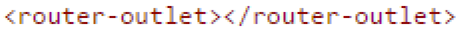
The page has to define a placeholder where the route has to render the content. In Angular 1, the target has to be marked with the directive *ng-view.*



```
<div ng-view></div>
```

ANGULAR 2

Angular 2 provides the component *router-outlet* to define the placeholder on the page.



```
<router-outlet></router-outlet>
```

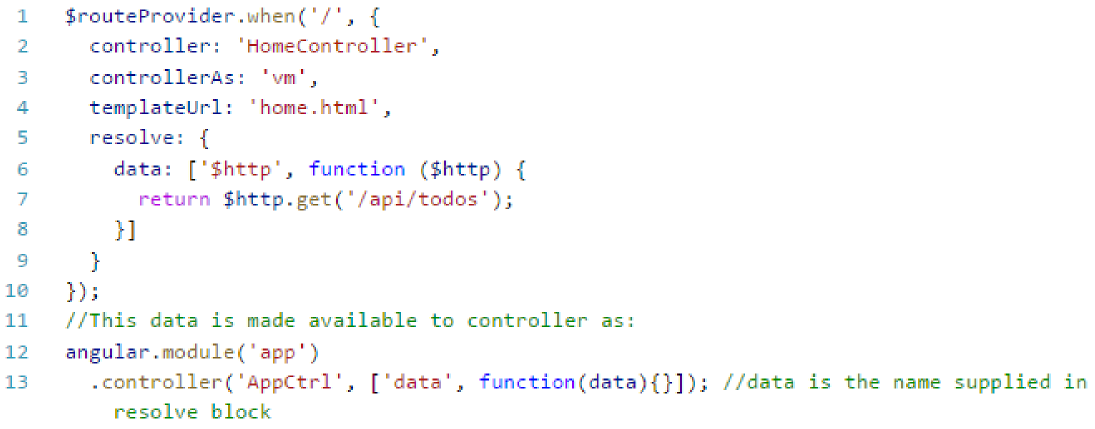
ROUTE RESOLVE BLOCKS

At times, the route to be loaded may need some data to display on the page. This data is provided using the resolve blocks.

ANGULAR 1

The resolve block in Angular 1 is an object with every entry equivalent to a factory. The resolve block has to return a value, which could be a simple object or a promise.

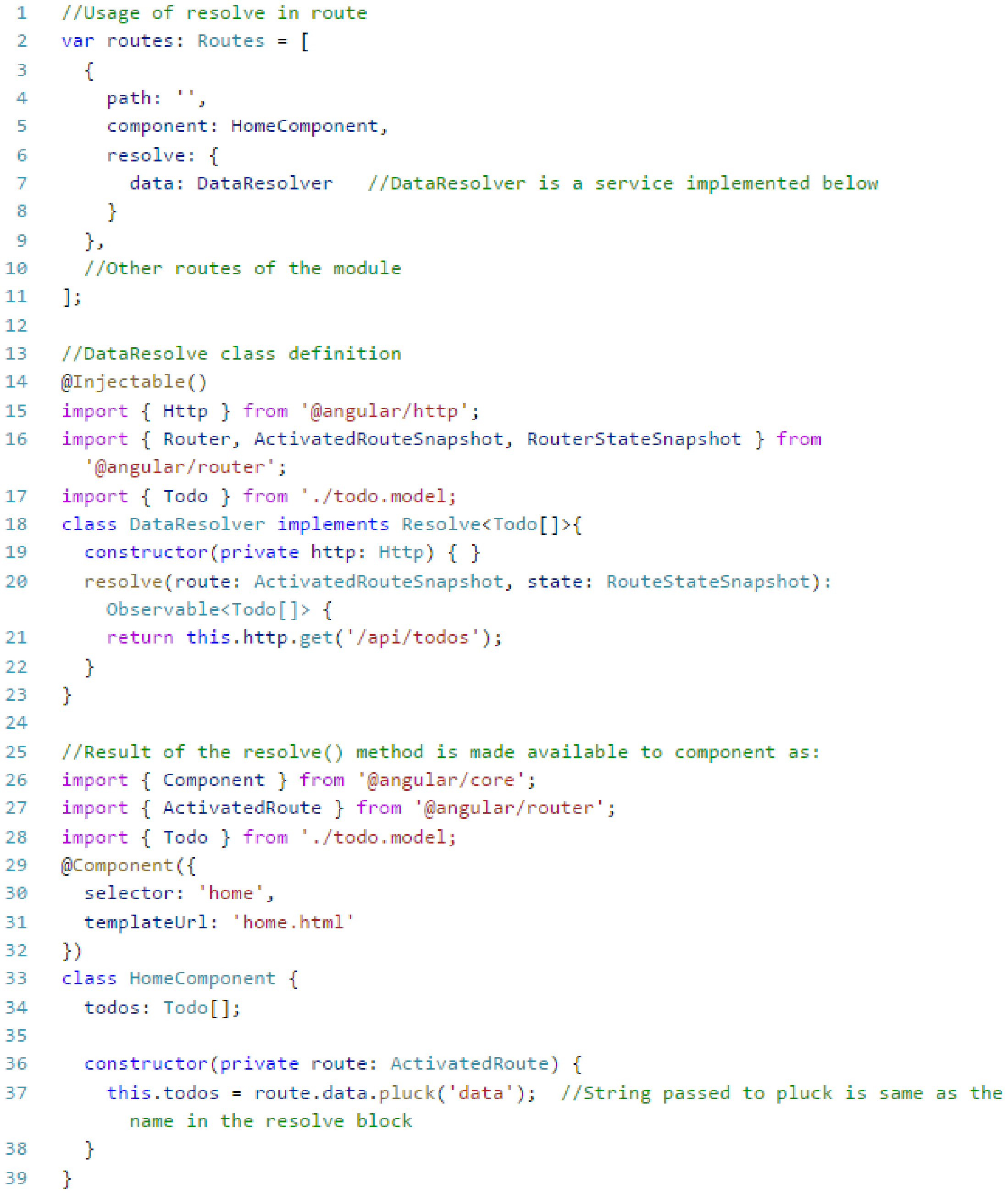
ANGULAR 2



```
$routeProvider.when('/', {
  controller: 'HomeController',
  controllerAs: 'vm',
  templateUrl: 'home.html',
  resolve: {
    data: ['$http', function ($http) {
      return $http.get('/api/todos');
    }]
  }
});
//This data is made available to controller as:
angular.module('app')
  .controller('AppCtrl', ['data', function(data){}]); //data is the name supplied in resolve block
```

The resolve block in Angular 2 is an object with every entry assigned with a service. The service has to implement the method *resolve*. This method is called by the framework before loading the route. The method may return an object, a promise, or an RxJS *Observable*.

Dependency Injection (DI)

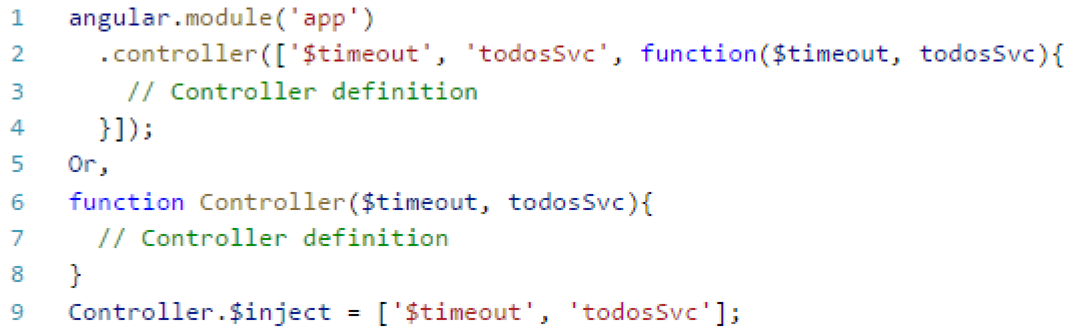


```
//Usage of resolve in route
var routes: Routes = [
  {
    path: '',
    component: HomeComponent,
    resolve: {
      data: DataResolver   //DataResolver is a service implemented below
    }
  },
  //Other routes of the module
];

//DataResolve class definition
@Injectable()
import { Http } from '@angular/http';
import { Router, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
import { Todo } from './todo.model;
class DataResolver implements Resolve<Todo[]>{
  constructor(private http: Http) { }
  resolve(route: ActivatedRouteSnapshot, state: RouteStateSnapshot): Observable<Todo[]> {
    return this.http.get('/api/todos');
  }
}

//Result of the resolve() method is made available to component as:
import { Component } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Todo } from './todo.model;
@Component({
  selector: 'home',
  templateUrl: 'home.html'
})
class HomeComponent {
  todos: Todo[];

  constructor(private route: ActivatedRoute) {
    this.todos = route.data.pluck('data');  //String passed to pluck is same as the name in the resolve block
  }
}
```

DI is a pattern in software development which is used to make the code loosely coupled, extensible, and testable. It is quite popular among typed server side languages like Java and C#. Angular 1 brought this feature to JavaScript, and Angular 2 continues to use this feature, but it implements it in a different way.

ANGULAR 1

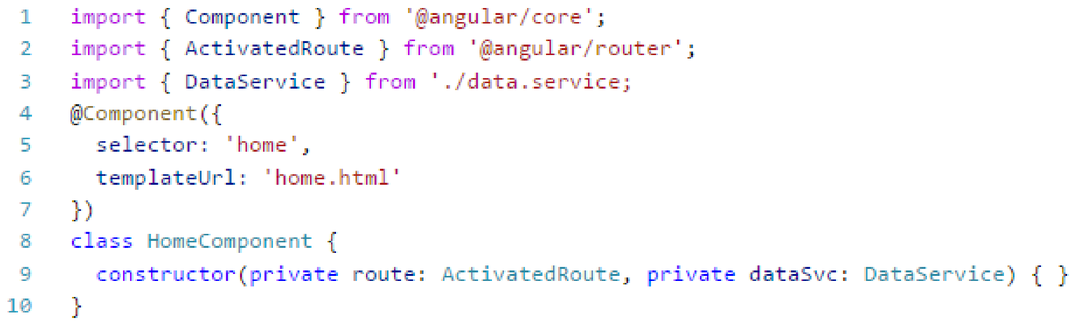
Every service, factory, value, provider and constant in Angular 1 is injectable. There are multiple ways to inject dependencies in Angular 1.



```
angular.module('app')
  .controller(['$timeout', 'todosSvc', function($timeout, todosSvc){
    // Controller definition
  }]);
Or,
function Controller($timeout, todosSvc){
  // Controller definition
}
Controller.$inject = ['$timeout', 'todosSvc'];
```

ANGULAR 2

Every service or token in Angular 2 can be injected in any component, service, directive or pipe. As all of these blocks are TypeScript classes, Angular 2 injects them into the constructors when their objects are created.



```
import { Component } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { DataService } from './data.service;
@Component({
  selector: 'home',
  templateUrl: 'home.html'
})
class HomeComponent {
  constructor(private route: ActivatedRoute, private dataSvc: DataService) { }
}
```

In the above snippet, *ActivatedRoute* and *DataService* are services injected into the component. If they are from a different module, the modules must be imported in the component's module. If they are custom services, they have to be registered in the *providers* property of module metadata.

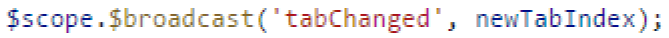
Pub/Sub Events

Events are a great way to communicate among multiple blocks of code without making them interact directly. This pattern helps in sending and receiving data very easily and effectively.

ANGULAR 1

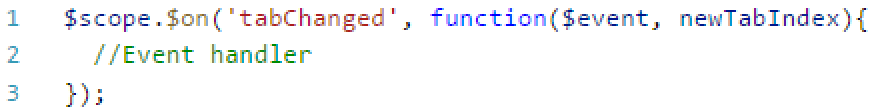
Angular 1's *scope* object has the methods *$broadcast* and *$on* to send and listen respectively. The events can be published from anywhere and listened from anywhere.

Sending event:



```
$scope.$broadcast('tabChanged', newTabIndex);
```

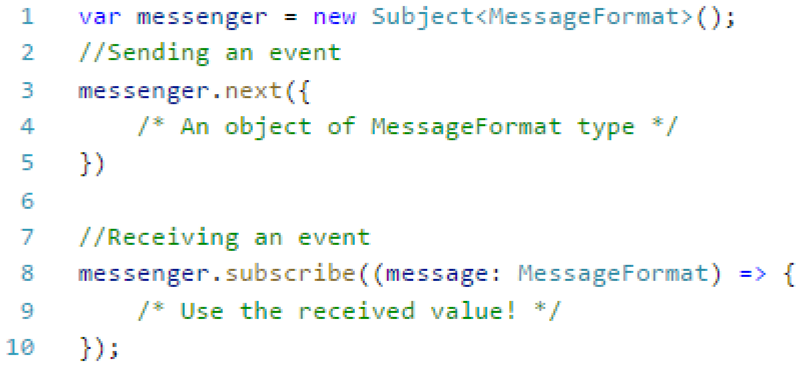
Listening to the event:



```
$scope.$on('tabChanged', function($event, newTabIndex){
  //Event handler
});
```

ANGULAR 2

Subject class from RxJS is used for pub/sub events in Angular 2. It can be used in any of the code blocks of Angular 2.



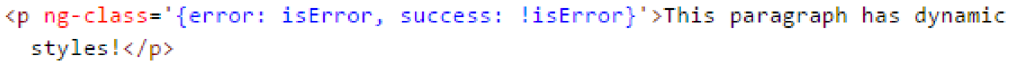
```
var messenger = new Subject<MessageFormat>();
//Sending an event
messenger.next({
    /* An object of MessageFormat type */
})

//Receiving an event
messenger.subscribe((message: MessageFormat) => {
    /* Use the received value! */
});
```

Assigning Dynamic CSS Classes

Effective use of data binding makes the UI more intuitive and responsive. The ability to dynamically modify CSS classes assigned on an element is built into both versions of Angular.

ANGULAR 1

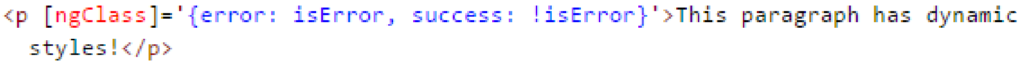
Angular 1 provides the directive *ng-class* to handle the CSS classes. It is extremely powerful and accepts different types of values.



```
<p ng-class='{error: isError, success: !isError}'>This paragraph has dynamic
  styles!</p>
```

ANGULAR 2

Angular 2 provides the directive *ngClass* to handle the CSS classes. Like its counterpart in Angular 1, it accepts different types of values.

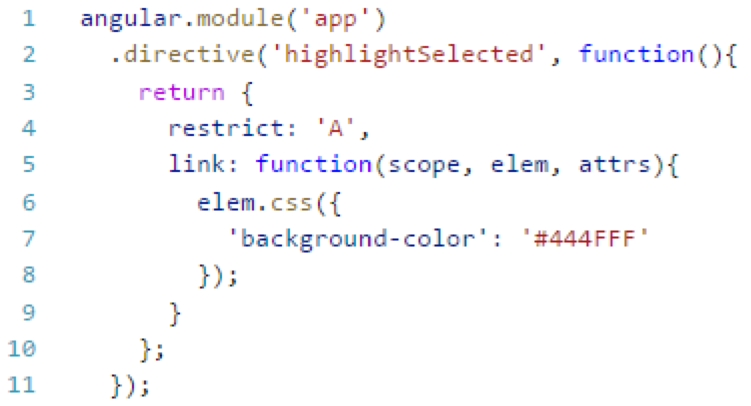


```
<p [ngClass]='{error: isError, success: !isError}'>This paragraph has dynamic
  styles!</p>
```

Manipulating DOM in a Directive

ANGULAR 1

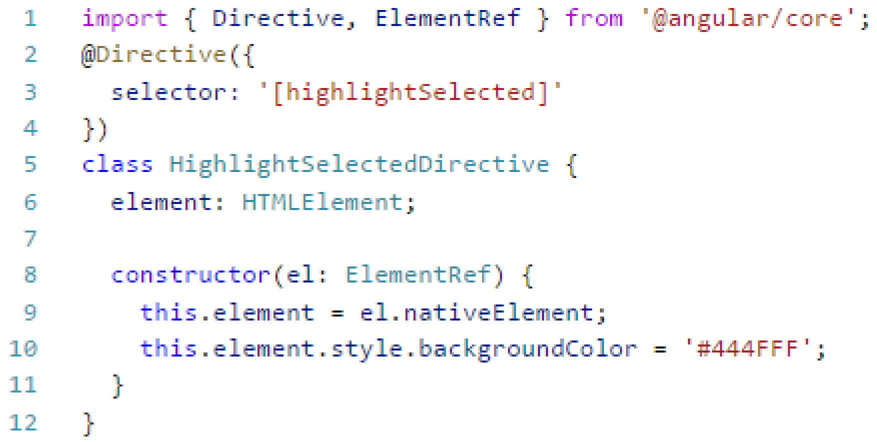
Angular 1 wraps the elements around jqLite, a lighter version of jQuery. Most of the common DOM operations of jQuery can be performed using the jqLite objects. As a best practice, the DOM manipulation has to be done in the directives.



```
angular.module('app')
  .directive('highlightSelected', function(){
    return {
      restrict: 'A',
      link: function(scope, elem, attrs){
        elem.css({
          'background-color': '#444FFF'
        });
      }
    };
  });
```

ANGULAR 2

Angular 2 doesn't use jqLite. The DOM operations in Angular 2 are the same as the operations that are generally performed on browser objects. Directives and Components can be used for DOM manipulation.

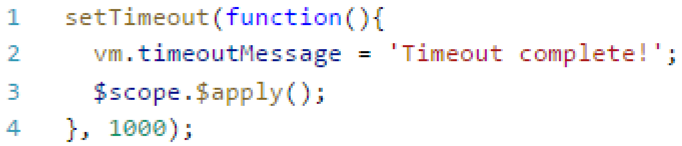


```
import { Directive, ElementRef } from '@angular/core';
@Directive({
  selector: '[highlightSelected]'
})
class HighlightSelectedDirective {
  element: HTMLElement;

  constructor(el: ElementRef) {
    this.element = el.nativeElement;
    this.element.style.backgroundColor = '#444FFF';
  }
}
```

Using Browser APIs

ANGULAR 1

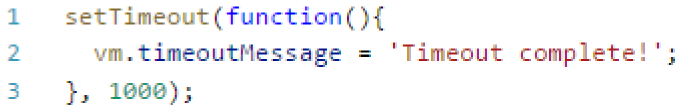
Angular 1 works based on a digest cycle. So any change in the values of any event other than Angular 1's own events, have to be notified to the framework by calling the *$scope.$apply* method.



```
setTimeout(function(){
  vm.timeoutMessage = 'Timeout complete!';
  $scope.$apply();
}, 1000);
```

ANGULAR 2

Angular 2 has zones running behind the scenes to monitor any activity happening in the browser's event loop. All browser events are tracked by zones, and thus the data binding system begins acting immediately without waiting for an explicit signal from the application.



```
setTimeout(function(){
  vm.timeoutMessage = 'Timeout complete!';
}, 1000);
```

Conclusion

Many of the features of Angular 1 can be mapped to their counterparts in Angular 2. This cheat sheet lists both Angular 1 and Angular 2 versions of most of the code blocks to ease the process of transition.

Lesson

1

Write Applications Fast Using Ignite UI Grid

WRITE WEB APPLICATIONS FASTER with Ignite UI. You can use the Ignite UI library to help quickly solve complex LOB requirements in HTML5, jQuery, Angular, React, or ASP.NET MVC. Use the Ignite UI library to add a fast, responsive grid with many features (like pagination, sorting, search, virtualization etc.). It takes just a few minutes using a few lines of code. Ignite UI has many controls, data visualizations charts, and framework elements that are simple to configure and customize. The ease of Ignite UI control configurations and customizations allows you to create a web application quickly.

In addition to seamlessly rendering large sets of data, the Ignite UI Grid features many valuable tools, such as filtering, paging, and sorting. You can learn more about Ignite UI features at *http://www.igniteui.com*; you can also learn more about Angular in *Angular Essentials, a* free eBook published by Infragistics.

Lesson Objectives

1. Add Ignite UI grid
2. Configure grid columns

For more information on the controls used in this lesson, see *http://infragistics.com/products/IgniteUI/grids/data-grid*.

At the end of this lesson, you will have a working grid configured for columns in an Angular application.

You can learn more about Ignite UI Angular 2 here: *https://github.com/IgniteUI/igniteui-angular2*.

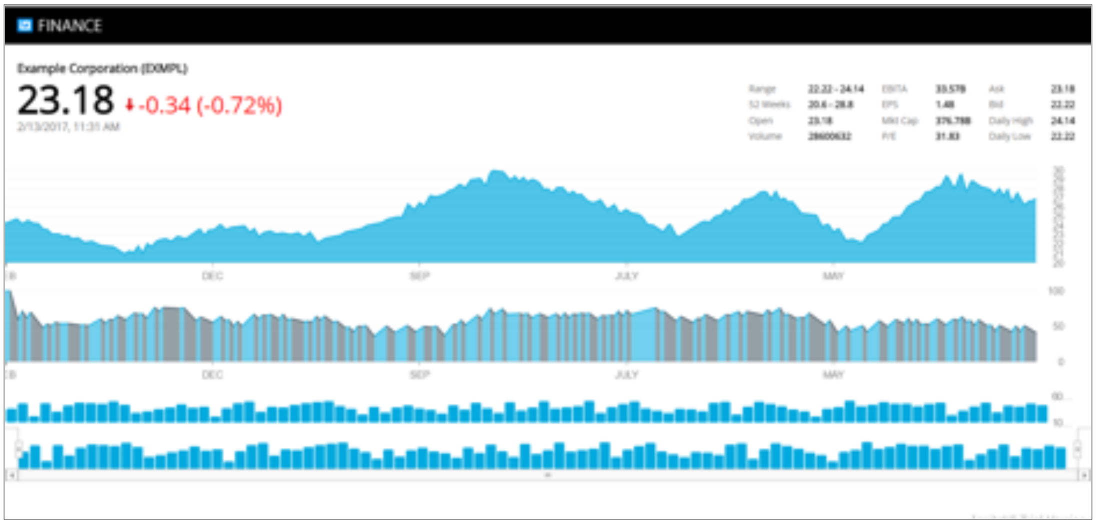
Setting up the Project

You may download the starter project for this lesson *by clicking here*. (You can also download the final project *by clicking here*.)

After downloading the project, navigate to the directory and run the commands below:

```
npm install
npm start
```

You have executed the npm install command to install all dependencies, using the npm start command to run the Angular application. If the project setup is correct, you will have a running Angular application as shown in the image below:



FINANCE

Example Corporation (EXMPL)

23.18 ↓-0.34 (-0.72%)

2/13/2017, 11:31 AM

Range	22.22 - 24.14	EBITA	33.578	Ask	23.18
52 Weeks	20.6 - 28.8	EPS	1.48	Bid	22.22
Open	23.18	Mkt Cap	376.788	Daily High	24.14
Volume	28600632	P/E	31.83	Daily Low	22.22

STEP 1: Import and Declare the Component

To work with Ignite UI Angular components, you must import and declare them in the module. For example, to use the igGrid component in an Angular application, import and declare the IgGridComponent in the application module.

In the project, navigate to the Finance App folder and then to the app folder. Open the file app.module.ts, and add the import statements below, just after the existing import statements.

```
import{IgGridComponent} from 'igniteui-angular2';
import {GridComponent} from './grid.component';
```

After importing the required components, you must declare them in the application module. Add IgGridComponent and GridComponent in the AppModule's declaration array. Modify @NgModule decorator in app.module.ts as shown below:

```
@NgModule({
  imports: [BrowserModule,HttpModule],
  declarations: [AppComponent,
         IgZoombarComponent,
         IgDataChartComponent,
         PriceChartComponent,
         InfoComponent,
         IndicatorChartComponent,
         VolumeChartComponent,
         IgGridComponent, GridComponent],
  providers: [AppService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

You have added IgGridComponent and GridComponent in the declaration array of AppModule module. We will examine other added components and properties (like providers) in subsequent lessons.

STEP 2: Create a Data Source

You need data to bind to the grid. This data can be a JavaScript array or a JSON object array and can be local or provided by a REST service.

Ideally, you should create a function to return data in an Angular service so you can use the data function in multiple components. However, for this lesson, there is already a function called getData in GridComponent class. This function returns a JSON object array.

In the app folder, open the file grid.component.ts and find the getData() function. In later lessons, you will learn how to create a grid that uses data from the REST services.

STEP 3: Get data

To use data returned from the getData() function, call the function inside Angular ngOnInit() life cycle hook and assign a returned value to the GridComponent property.

Learn more about Angular Life Cycle hooks here: *https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html*

In the app folder, open the file grid.component.ts and modify the ngOnInit() function as shown in the listing below:

```
ngOnInit(){
  this.stocks = this.getData();
}
```

STEP 4: Create a Grid

The Ignite UI Grid component can be used like any other component. In the app folder, open the file grid.component.html, and add the code as shown in the below listing:

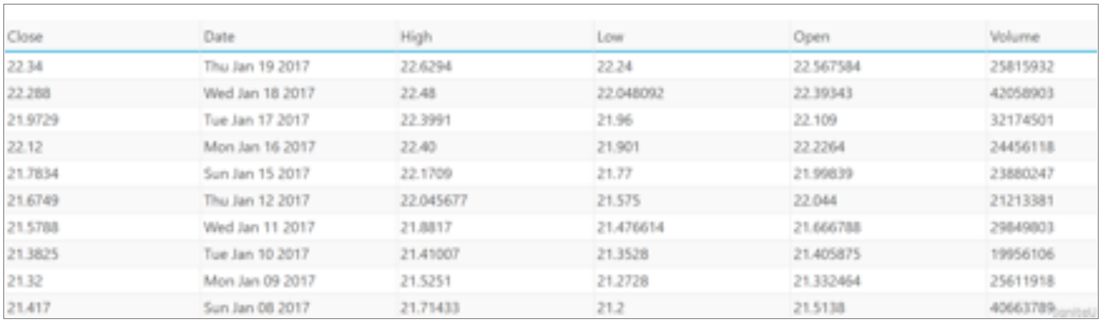
```
<ig-grid widgetId="grid1" [dataSource]="stocks" [autoGenerateColumns]="true">
```

STEP 5: Use in an Application

To use the GridComponent in an application: in the app folder, open the app.component.html file and add the code below just at the end of all of the markup. Add it below the <br/> element.

```
<grid></grid>
```

Navigate to the application, scroll down, and, at the bottom of the page, you will find the grid added as shown in the image below:



Close	Date	High	Low	Open	Volume
22.34	Thu Jan 19 2017	22.6294	22.24	22.567584	25815932
22.288	Wed Jan 18 2017	22.48	22.048092	22.39343	42058903
21.9729	Tue Jan 17 2017	22.3991	21.96	22.109	32174501
22.12	Mon Jan 16 2017	22.40	21.901	22.2264	24456118
21.7834	Sun Jan 15 2017	22.1709	21.77	21.99839	23880247
21.6749	Thu Jan 12 2017	22.045677	21.575	22.044	21213381
21.5788	Wed Jan 11 2017	21.8817	21.476614	21.666788	29849803
21.3825	Tue Jan 10 2017	21.41007	21.3528	21.405875	19956106
21.32	Mon Jan 09 2017	21.5251	21.2728	21.332464	25611918
21.417	Sun Jan 08 2017	21.71433	21.2	21.5138	40663789

STEP 6: Configure Columns of the Grid

In Step 4, you created a grid by setting the autoGenerateColumns property to true. You didn't need to configure the columns of the grid and they were generated automatically.

In many cases you may need to configure columns manually. You can configure columns and other features such as paging, sorting, and filtering of the grid in the component class.

To configure columns: in the app folder, open grid.component.ts file, and update ngOnInit() function in grid.component.ts file with the listing below:

```
ngOnInit() {
   this.stocks = this.getData();
   this.gridId = "Grid1";
```

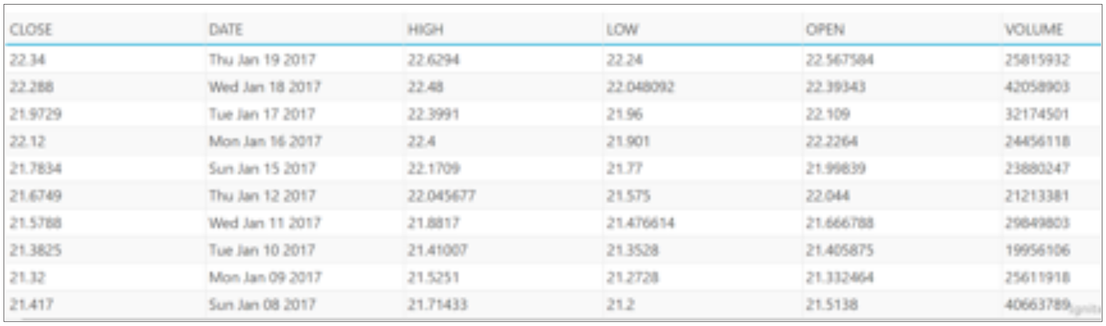
```
    this.gridOptions = {
      dataSource: this.stocks,
      autoGenerateColumns: false,
      columns: [
        { headerText: "CLOSE", key: "Close", dataType: "number" },
        { headerText: "DATE", key: "Date", dataType: "string" },
        { headerText: "HIGH", key: "High", dataType: "number" },
        { headerText: "LOW", key: "Low", dataType: "number" },
        { headerText: "OPEN", key: "Open", dataType: "number" },
        { headerText: "VOLUME", key: "Volume", dataType: "number" }
      ]
    }
  }
```

STEP 7: Modify The Grid With Configured Columns

Ignite UI grid options and widgetId properties are enabled for two-way data binding, so any changes in the source will be reflected on the grid. To set options and widgetId properties: in the app folder, open the file grid.component.html, and modify it as shown in the below listing:

```
<ig-grid [(options)]= "gridOptions" [(widgetId)]="gridId">
```

Navigate to the application and scroll to the bottom of the page to find the grid added as shown below:



CLOSE	DATE	HIGH	LOW	OPEN	VOLUME
22.34	Thu Jan 19 2017	22.6294	22.24	22.567584	25815932
22.288	Wed Jan 18 2017	22.48	22.048092	22.39343	42058903
21.9729	Tue Jan 17 2017	22.3991	21.96	22.109	32174501
22.12	Mon Jan 16 2017	22.4	21.901	22.2264	24456118
21.7834	Sun Jan 15 2017	22.1709	21.77	21.99839	23880247
21.6749	Thu Jan 12 2017	22.045677	21.575	22.044	21213381
21.5788	Wed Jan 11 2017	21.8817	21.476614	21.666788	29849803
21.3825	Tue Jan 10 2017	21.41007	21.3528	21.405875	19956106
21.32	Mon Jan 09 2017	21.5251	21.2728	21.332464	25611918
21.417	Sun Jan 08 2017	21.71433	21.2	21.5138	40663789

Conclusion

Ignite UI can help you write web applications more quickly. In addition to Angular, Ignite UI may be used with React, AngularJS, jQuery, and ASP.NET MVC.

Lesson

2

Write Applications Fast Using Ignite UI Data Charts

Write web applications and solve complex LOB requirements more quickly with Ignite UI. The Ignite UI library (with HTML5, jQuery, Angular, React, or ASP.NET MVC) can add complex and dynamic charts to your web application quickly with a few lines of code.

Different types of charts are available in the Ignite UI:

- **Data Chart:** Display data on x-axis and y-axis as bars, lines, areas etc.
- **Pie Chart:** Display data in a circle, divided into sectors that each represent a proportion of the total data.
- **Doughnut Chart:** Display data in a circle, with more than one data series.

There are approximately 50 types of data charts available in Ignite UI. Learn more about Ignite UI data charts here: *http://www.igniteui.com/data-chart/overview*; you can also learn more about Angular in *Angular Essentials, a* free eBook published by Infragistics.

Lesson Objectives

1. Add Ignite UI DataChart
2. Configure data charts for axes, data sources, and series
3. Configure data charts for various series types.

For more information on the controls used in this lesson, see *http://www.infragistics.com/products/IgniteUI/charts/data-chart.*

At the end of the lesson, you will have a working data chart configured for different types of series in an Angular application.

Learn more about Ignite UI Angular 2 here: *https://github.com/IgniteUI/igniteui-angular2*

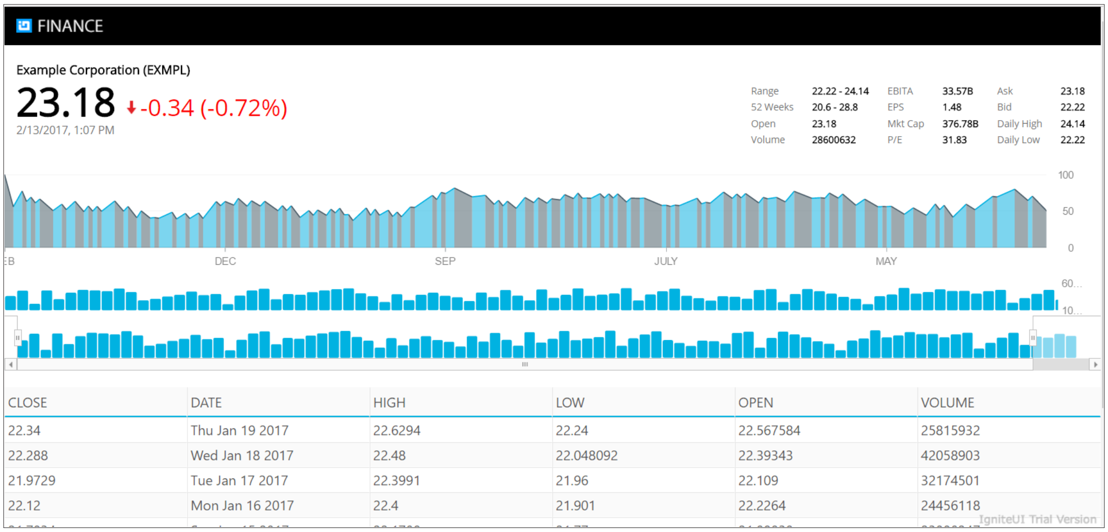
Setting up the Project

Download the starter project for this lesson *by clicking here*. (You can also download the final project *by clicking here*.)

After you download the project, navigate to the Finance App directory and run the commands below:

```
npm install
npm start
```

You have executed the npm install command to install all dependencies and are using the npm start command to run the Angular application. If the project setup is correct, you will have a running Angular application as shown below. If you receive an error while running the application, stop and run the npm start command again.



Example Corporation (EXMPL)

23.18 ↓-0.34 (-0.72%)
2/13/2017, 1:07 PM

Range	22.22 - 24.14	EBITA	33.57B	Ask	23.18
52 Weeks	20.6 - 28.8	EPS	1.48	Bid	22.22
Open	23.18	Mkt Cap	376.78B	Daily High	24.14
Volume	28600632	P/E	31.83	Daily Low	22.22

CLOSE	DATE	HIGH	LOW	OPEN	VOLUME
22.34	Thu Jan 19 2017	22.6294	22.24	22.567584	25815932
22.288	Wed Jan 18 2017	22.48	22.048092	22.39343	42058903
21.9729	Tue Jan 17 2017	22.3991	21.96	22.109	32174501
22.12	Mon Jan 16 2017	22.4	21.901	22.2264	24456118

STEP 1: Import and Declare the Component

To work with Ignite UI Angular components, you must import and declare them in the module. For example, to use the igDataChart component in an Angular application, import and declare IgDataChartComponent in the application module.

In the project, navigate to the Finance App folder, and then the app folder. Open the file app.module.ts, and you will find that igDataChartComponent has been added. Add the import statements below, after the existing import statements.

```
import{PriceChartComponent} from './charts/pricechart.component';
```

After importing the required components, you must declare them in the application module. Add PriceChartComponent in the AppModule's declaration array. Modify @ NgModule decorator in app.module.ts as shown below:

```
@NgModule({
  imports: [BrowserModule, HttpModule],
  declarations: [AppComponent,
    IgZoombarComponent,
    IgDataChartComponent,
    InfoComponent,
    IndicatorChartComponent,
    VolumeChartComponent,
    IgGridComponent,
    GridComponent,
    PriceChartComponent],
  providers: [AppService],
  bootstrap: [AppComponent]
})
```

You've now added PriceChartComponent in the declaration array of AppModule module. Other added components and other properties like providers will be outlined in subsequent lessons.

STEP 2: Create Data Source

The data needed to bind the data chart can be a JavaScript array or a JSON object array and can be local or be may be provided by a REST service.

Ideally, you should create a function to return data in the Angular service so data may function in multiple components. However, for this lesson, there is already a function called getData in PriceChartComponent class, which returns a JSON object array. In the app\charts folder, open the file pricechart.component.ts and find the getData() function. In future lessons, you will learn to create a grid which uses data from the REST services.

STEP 3: Get Data

To use data returned from getData() function, call the function inside the Angular ngOnInit() life cycle hook and assign returned value to PriceChartComponent property.

Learn more about Angular Life Cycle hooks here: *https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html*

In the app\charts folder, open the file pricechart.component.ts and modify the ngOnInit() function as shown in the listing below:

```
ngOnInit() {
  this.stocks = this.getData();
}
```

STEP 4: Configure Axes

To create a data chart, you must configure the chart options. Usually, chart options consist of three main properties:

1. X and Y axis
2. Data source
3. Series

Aside from these properties, other important properties are height, width, title, etc.

To configure axes, open the pricechart.component.ts file and directly after the ngOnInit() function, add the getPriceChartAxes() function as shown listed below :

```
getPriceChartAxes() {
  return [
    {
      name: "xAxis",
      type: "categoryX",
      label: "Date"
    },
    {
      name: "yAxis",
      type: "numericY",
      labelLocation: "outsideRight",
      labelExtent: 40
    }
  ];
}
```

In the above listing:

- X-axis type and Y-axis type are useful to display financial, scatter, or category price series. Other possible values are category, numericAngle, categoryDateTimeX, categoryAngle etc.

You can learn about these values and types of charts here: *http://www.igniteui.com/help/igdatachart-series-types*

- Y-axis labelExtent value is set to 40, which specifies the size of the area dedicated to the labels or how far label would be from the axis.

STEP 5: Configure Series

An Ignite UI data chart can have any number of series, but it must have at least one series. To add a series in a data chart in the app\charts folder, open pricechart.component.ts file and directly after the getPriceChartAxes() function, add the getPriceChartSeries() function as shown in the listing below:

```
getPriceChartSeries() {
  return [
    {
      name: "stockSeries",
      type: "splineArea",
      title: "Price Data",
      isHighlightingEnabled: true,
      isTransitionInEnabled: true,
      xAxis: "xAxis",
      yAxis: "yAxis",
      valueMemberPath: "High",
      showTooltip: true,
      Outline: "#00AADE"
    }
  ];
}
```

In the above listing:

- Series type value is set to splineArea to create Spline Area series. If you want to create a Line series, set the value of type to "line". IgniteUI provides more than 25 possible series types for the data chart including area, bar, and column.
- As a series valueMemberPath, you need to set property from the data array to be displayed in the chart. Here you are setting "Hight" property from the data source will be rendered in the data chart series.
- Series isTransitionInEnabled value is set to true to enable animation when data source is assigned.

STEP 6: Configure Chart Option

You have configured axis and series. Next, configure a chart option. In chart option, you set all other important properties of a data chart.

Learn more about chart properties here: *http://www.igniteui.com/data-chart/overview*

To configure a data chart, in app\charts folder, open pricechart.component.ts file and directly after getPriceChartSeries () function, add getPriceChartOption() function as shown in the listing below:

```
getPriceChartOptions() {
  return {
    axes: this.getPriceChartAxes(),
    series: this.getPriceChartSeries(),
    windowResponse: "deferred",
    horizontalZoomable: true,
    width: "100%",
    height: this.desiredHeight,
    leftMargin: 0,
    rightMargin: 30,
    windowRectMinWidth: 0.05,
    syncChannel: "channel1",
    synchronizeVertically: false,
    syncrhonizeHorizontally: false
  };
}
```

In the above listing:

- Chart's syncChannel property is set so the chart can be synced with other charts of the application to intimate functionalities of other controls such as ZoomBar. Charts synced in same channel can use single zoom bar for zoom in and zoom out functionalities.
- Chart's windowResponse property is set to "deferred" so the chart view update will defer until after the user action is complete. Another possible value is "immediate."

STEP 7: Initialize Chart Option and Data Source

To initialize chart option and data source, in pricechart.component.ts file, modify ngOnInit() function as shown in the listing below:

```
ngOnInit() {
  this.stocks = this.getData();
  this.desiredHeight = 0.22 * (window.screen.height) + "px";
  this.chartOptions = this.getPriceChartOptions();
}
```

STEP 8: Create Chart

To create a chart, open pricechart.component.html file and the add markup given below:

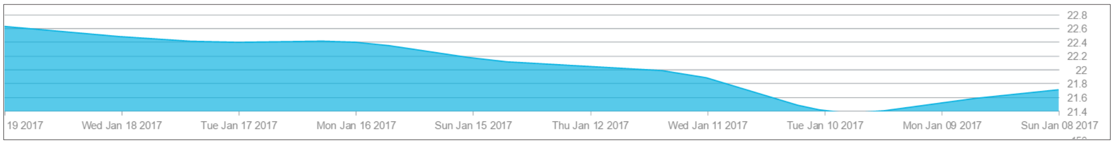
```
<ig-data-chart [(options)]="chartOptions" [(dataSource)]="stocks" widgetId="price-chart"></ig-data-chart>
```

STEP 9: Use in Application

To use PriceChartComponent in an application, in the app folder, open app.component.html file and add below code just after the <info-screen> element and before <indicatorchart> element.

```
<pricechart></pricechart>
```

Navigate to the application, scroll down, and at the bottom of the page, you will find chart added as shown in the image below:



Conclusion

Ignite UI is useful in writing web applications quickly. In addition to Angular, you can use Ignite UI in React, AngularJS, jQuery, and ASP.NET MVC. In this lesson, you learned how to use Ignite UI Data Charts in an Angular application.

Lesson

3

Sort, Filter, and Page Fast With Ignite UI Grid

Ignite UI enables you to write web applications faster. You can use Ignite UI library with HTML5, jQuery, Angular, React, or ASP.NET MVC. It helps you to solve complex LOB requirements faster. The Ignite UI library makes it possible for you to quickly and efficiently add a fast, responsive grid with features like pagination, sorting, search, virtualization, and more.

In addition to seamlessly rendering large sets of data, IgniteUI Grid is loaded with many other features, such as filtering, paging, and sorting. You can learn more about Ignite UI features at *http://www.igniteui.com*; you can also learn more about Angular in *Angular Essentials, a* free eBook published by Infragistics.

In this lesson, you will learn how to configure various important features of Ignite UI Grid.

Lesson Objective

1. Enable sorting on the grid
2. Enable filters on the grid
3. Enable paging on the grid

For more information on the controls used in this lesson, see *http://infragistics.com/products/IgniteUI/grids/data-grid*.

At the end of this lesson, you will have an Ignite UI Grid configured for basic features such as sorting, filtering, and pagination in an Angular application.

You can learn more about Ignite UI Angular 2 here: *https://github.com/IgniteUI/igniteui-angular2* .

Setting up the Project

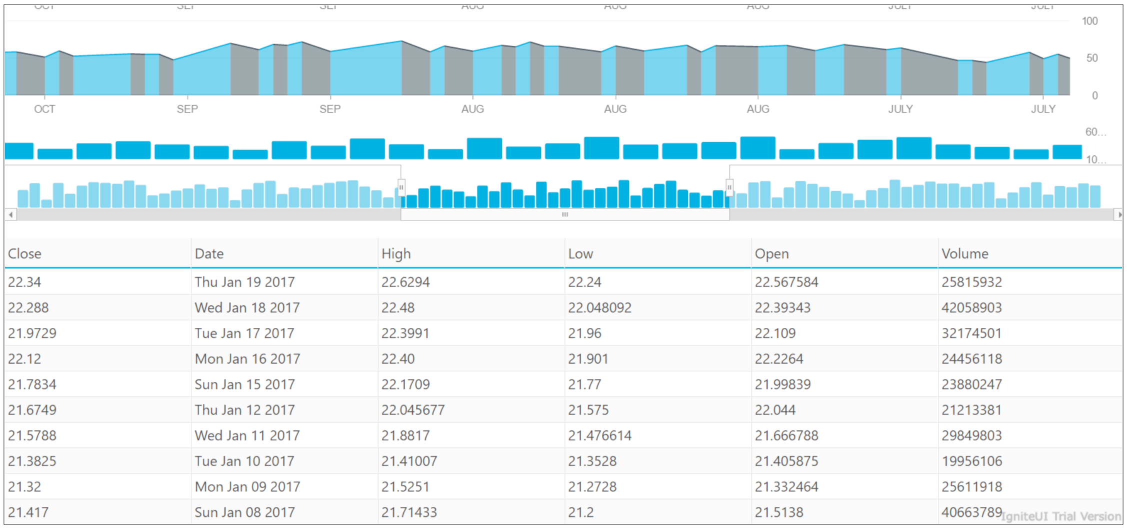
You can download the starting project for this lesson *by clicking here*. (You can also download the final project *by clicking here*.)

After downloading the project, navigate to Finance App directory and run the commands below:

```
npm install
npm start
```

You executed npm install command to install all dependencies, and to use the npm start command to run the Angular application. If the project setup is correct, you will have a running Angular application as shown in the image below.

Scroll to bottom of the application and you will find the Ignite UI Grid. At the end of the lesson, this grid will be configured with sorting, paging, and filter features.



Close	Date	High	Low	Open	Volume
22.34	Thu Jan 19 2017	22.6294	22.24	22.567584	25815932
22.288	Wed Jan 18 2017	22.48	22.048092	22.39343	42058903
21.9729	Tue Jan 17 2017	22.3991	21.96	22.109	32174501
22.12	Mon Jan 16 2017	22.40	21.901	22.2264	24456118
21.7834	Sun Jan 15 2017	22.1709	21.77	21.99839	23880247
21.6749	Thu Jan 12 2017	22.045677	21.575	22.044	21213381
21.5788	Wed Jan 11 2017	21.8817	21.476614	21.666788	29849803
21.3825	Tue Jan 10 2017	21.41007	21.3528	21.405875	19956106
21.32	Mon Jan 09 2017	21.5251	21.2728	21.332464	25611918
21.417	Sun Jan 08 2017	21.71433	21.2	21.5138	40663789

STEP 1: Enable Sorting

You can enable sorting on Ignite UI Grid by adding a feature with the name "Sorting" in the grid. Ignite UI Grid supports local and remote sorting.

To enable local sorting, create an object with the following properties and add it to the features property of the grid option:

- **name :** set to Sorting
- **type :** set to local

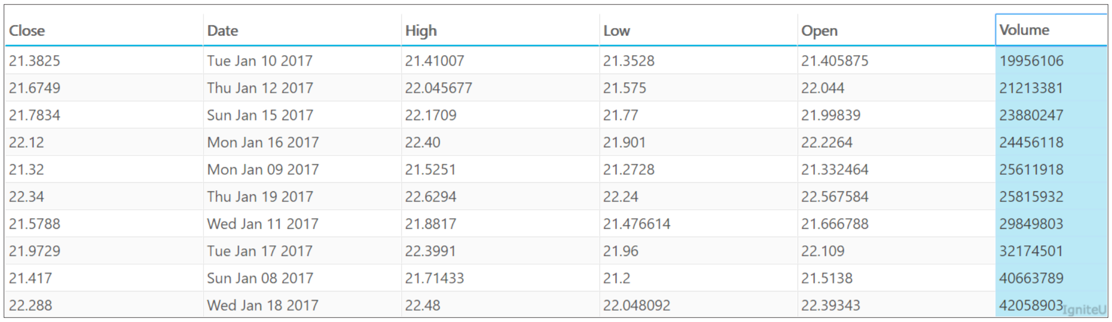
To do this, in the App folder, open the file grid.component.ts and add below getGridFeatures() function just after the getData() function.

```
getGridFeatures()
{
   return [
      {
         name: "Sorting",
         type: "local"
      }
   ];
}
```

Next, add sorting features to the grid options. For that in the this.gridOptions, add a new property called feature and set its value to this.getGridFeatures(). Updated ngOnInit() function in the grid.component.ts file will look like as shown in the code listed below:

```
ngOnInit() {
   this.stocks = this.getData();
   this.gridId = "grid1"
   this.gridOptions = {
      dataSource: this.stocks,
      autoGenerateColumns: true,
      features: this.getGridFeatures()
   }
}
```

Navigate to the application, scroll down, and at the bottom of the page, you will find the grid added as shown below:



Close	Date	High	Low	Open	Volume
21.3825	Tue Jan 10 2017	21.41007	21.3528	21.405875	19956106
21.6749	Thu Jan 12 2017	22.045677	21.575	22.044	21213381
21.7834	Sun Jan 15 2017	22.1709	21.77	21.99839	23880247
22.12	Mon Jan 16 2017	22.40	21.901	22.2264	24456118
21.32	Mon Jan 09 2017	21.5251	21.2728	21.332464	25611918
22.34	Thu Jan 19 2017	22.6294	22.24	22.567584	25815932
21.5788	Wed Jan 11 2017	21.8817	21.476614	21.666788	29849803
21.9729	Tue Jan 17 2017	22.3991	21.96	22.109	32174501
21.417	Sun Jan 08 2017	21.71433	21.2	21.5138	40663789
22.288	Wed Jan 18 2017	22.48	22.048092	22.39343	42058903

Click on any of the columns and you will find that the grid is sorted for that particular column as shown in the image above. In addition, you will notice that sorted column headers have sorting indicators applied, so sortable columns are distinguished visually from the rest of the columns in the grid. Ignite UI also supports sorting on multiple columns.

You have configured sorting locally. Ignite UI also supports remote sorting. Learn more here: *http://www.igniteui.com/grid/sorting-remote*

STEP 2: Enable Paging

You can enable paging on Ignite UI grid by adding a feature named "Paging" in the grid. Ignite UI Grid supports local and remote pagination.

To enable local paging, create an object with following properties and add to the features property of the grid option:

- **name :** set to Paging
- **type :** set to local
- **pageSize :** 5

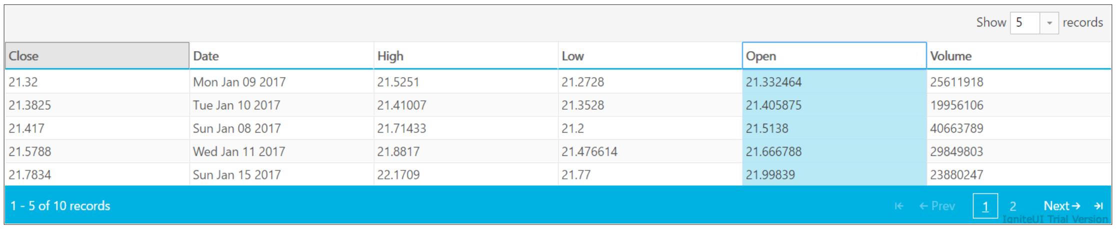
You can also set show/hide size drop-down or show/hide paging buttons, etc. Therefore, Paging feature object would look like below

```
{
    name: "Paging",
    type: "local",
    pageSize: 5
}
```

Add above object in features object array. To do that, open the file grid.component.ts and modify getGridFeatures() function such that it returns both paging and sorting features. After adding paging feature, getGridFeatures() function will look like below,

```
getGridFeatures() {
   return [
      {
         name: "Sorting",
         type: "local"
      },
      {
         name: "Paging",
         type: "local",
         pageSize: 5
      }
   ];
}
```

To test configured grid with paging: navigate to the application, scroll down, and at the bottom of the page you will find the grid added as shown in the image below:



Close	Date	High	Low	Open	Volume
21.32	Mon Jan 09 2017	21.5251	21.2728	21.332464	25611918
21.3825	Tue Jan 10 2017	21.41007	21.3528	21.405875	19956106
21.417	Sun Jan 08 2017	21.71433	21.2	21.5138	40663789
21.5788	Wed Jan 11 2017	21.8817	21.476614	21.666788	29849803
21.7834	Sun Jan 15 2017	22.1709	21.77	21.99839	23880247

1 - 5 of 10 records

Ignite UI also supports remote paging. Learn more here: *http://www.igniteui.com/help/iggrid-paging#remote*

STEP 3: Enable Filtering

You can enable filter on Ignite UI grid by adding a feature named "Filtering" in the grid.

To enable filtering, create an object with following properties and add to the features property of the grid option:

- **name :** set to Filtering
- **allowFiltering :** set to true
- **caseSensitive:** set to false/true

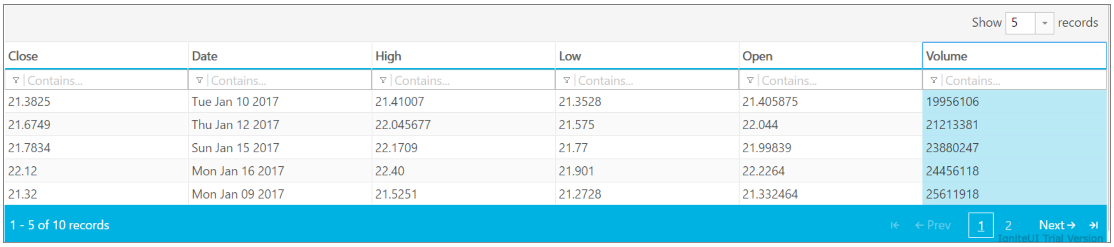
Filtering feature object would look like below

```
{
   name: "Filtering",
   allowFiltering: true,
   caseSensitive: false
}
```

Add above object in features object array. To do that, open the file grid.component.ts and modify getGridFeatures() function such that it returns paging, sorting and filtering features. After adding filtering feature, getGridFeatures() function will look like below,

```
getGridFeatures() {
  return [
    {
       name: "Sorting",
       type: "local"
    },
    {
       name: "Paging",
            type: "local",
            pageSize: 5
        },
        {
            name: "Filtering",
            allowFiltering: true,
            caseSensitive: false
        }
    ];
}
```

To test the configured grid with filtering, navigate to the application, scroll down, and at the bottom of the page, you will find the grid added as shown below:



Show 5 records

Close	Date	High	Low	Open	Volume
Contains...	Contains...	Contains...	Contains...	Contains...	Contains...
21.3825	Tue Jan 10 2017	21.41007	21.3528	21.405875	19956106
21.6749	Thu Jan 12 2017	22.045677	21.575	22.044	21213381
21.7834	Sun Jan 15 2017	22.1709	21.77	21.99839	23880247
22.12	Mon Jan 16 2017	22.40	21.901	22.2264	24456118
21.32	Mon Jan 09 2017	21.5251	21.2728	21.332464	25611918

1 - 5 of 10 records

As you will find the grid configured with filtering, paging, and sorting.

Conclusion

Ignite UI makes it possible to write your web applications faster. In addition to Angular, Ignite UI can be used in React, AngularJS, jQuery, and ASP.NET MVC. In this lesson, you learned to configure a grid for basic features like paging, sorting, and filtering.

Lesson

4

Run Fast Using Virtualization in Ignite UI Grids

Why Virtualization?

Virtualization is a valuable tool when displaying large sets of records to end users. A virtualized grid can bind to and support a data source of thousands of records, while providing a responsive experience to the end user using a rapid scroll of the grid.

The Ignite UI igGrid support two types of virtualization

1. Continuous Virtualization
2. Fixed Virtualization

In fixed virtualization, only the visible rows are rendered in the grid; in continuous virtualization, a pre-defined number of rows are rendered in the grid. The Ignite UI grid can be configured for column virtualization, row virtualization, or both. In the row virtualization, data row will be virtualized; in columns virtualization, columns of data source will be virtualized. You may choose to enable column virtualization when you have large number of columns in a data source.

Lesson Objectives

1. Configure grid for fixed virtualization
2. Configure grid for continuous virtualization

For more information on the controls used in this lesson, see *http://infragistics.com/products/IgniteUI/grids/data-grid.*

At the end of the lesson, you will have a working grid configured for virtualization in an Angular application. You can learn more about Ignite UI Angular 2 here: *https://github.com/IgniteUI/igniteui-angular2*; you can also learn more about Angular in *Angular Essentials, a* free eBook published by Infragistics.

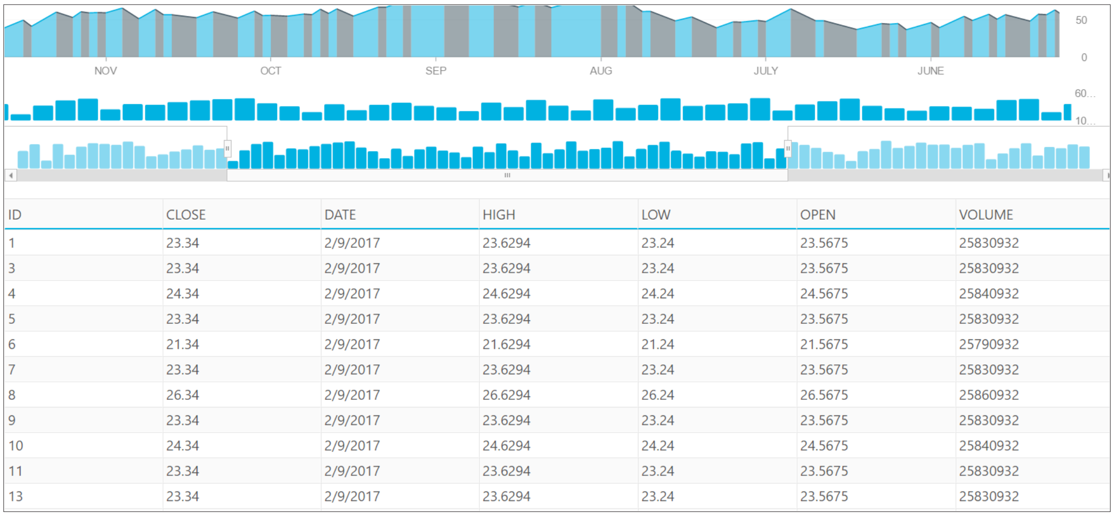
Setting up the Project

You can download the starting project for this lesson *by clicking here*. (You can also download the final project *by clicking here*.)

After downloading the project, navigate to run the commands below:

```
npm install
npm start
```

You have executed the npm install command to install all dependencies and the npm start command to run the Angular application. If the project setup is correct, you will have a running Angular application with a grid as shown below. If you receive an error while running the application, stop and run the npm start command again.



ID	CLOSE	DATE	HIGH	LOW	OPEN	VOLUME
1	23.34	2/9/2017	23.6294	23.24	23.5675	25830932
3	23.34	2/9/2017	23.6294	23.24	23.5675	25830932
4	24.34	2/9/2017	24.6294	24.24	24.5675	25840932
5	23.34	2/9/2017	23.6294	23.24	23.5675	25830932
6	21.34	2/9/2017	21.6294	21.24	21.5675	25790932
7	23.34	2/9/2017	23.6294	23.24	23.5675	25830932
8	26.34	2/9/2017	26.6294	26.24	26.5675	25860932
9	23.34	2/9/2017	23.6294	23.24	23.5675	25830932
10	24.34	2/9/2017	24.6294	24.24	24.5675	25840932
11	23.34	2/9/2017	23.6294	23.24	23.5675	25830932
13	23.34	2/9/2017	23.6294	23.24	23.5675	25830932

The starter project includes a grid that was created with a large set of data. Since virtualization is not yet enabled on the grid, the grid is taking some time to render all of the records. In addition, it is rendering all rows at once. For 5,000 rows, the grid is creating 5,000 row elements on the DOM, which causes the application to run more slowly and less efficiently. To run the application faster, despite the very large set of data, you need to configure virtualization on the grid.

The starter project of this lesson contains code to work with REST API in an Angular application to create a large data set. To work with REST API and server communication, Angular provides an http class.

Learn more about the http class and server communication in Angular here: *https://angular.io/docs/ts/latest/guide/server-communication.html*

STEP 1: Enabling Fixed Virtualization

To enable virtualization, you must set these three properties of the Ignite UI grid.

1. virtualizationMode
2. virtualization
3. height

You must set the height property of the grid to enable virtualization. If the height property is not set, and virtualization is true, Ignite UI will throw an error.

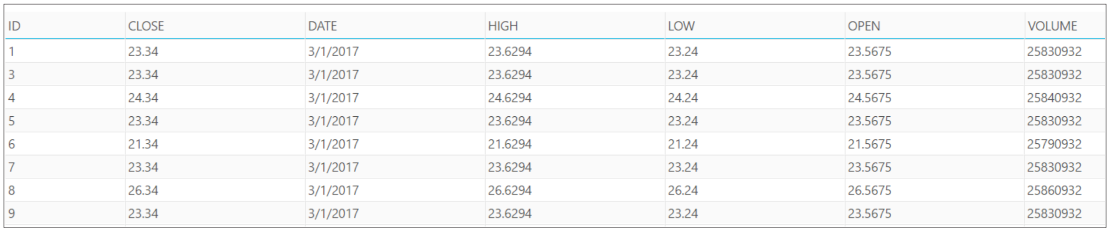
To enable both row and column virtualization, set the value of the virtualization property to true. The virtualization property can also be set to a numeric value so whenever a number of records in the data source is more than the specified number, virtualization will be enabled.

To enable fixed virtualization, set the following properties of the grid:

- Set virtualization property to “true”
- Set virtulizationMode property to “fixed”
- Set height property to some pixel value (ere it will be set to “300px”)
- To configure all of these properties of the grid, in the app folder: open the grid.component.ts file and update the getGridOptions() function as shown in the highlighted listing below. You are adding three more properties to existing grid options.

```
getGridOptions() {
  return {
    width: "100%",
    autoGenerateColumns: false,
    height: "300px",
    virtulization: true,
    virtualizationMode: "fixed",
    columns: [
      { headerText: "ID", key: "Id", dataType: "string",width:"10%" },
      { headerText: "CLOSE", key: "Close", dataType: "number",width:"15%" },
      { headerText: "DATE", key: "Date", dataType: "string",width:"15%" },
      { headerText: "HIGH", key: "High", dataType: "number",width:"15%" },
      { headerText: "LOW", key: "Low", dataType: "number",width:"15%" },
      { headerText: "OPEN", key: "Open", dataType: "number",width:"15%" },
      { headerText: "VOLUME", key: "Volume", dataType: "number",width:"15%"}
    ]
  };
}
```

To test the fixed virtualization: navigate to the application, scroll down, and you will find the grid configured with fixed virtualization added as shown in the image below:



ID	CLOSE	DATE	HIGH	LOW	OPEN	VOLUME
1	23.34	3/1/2017	23.6294	23.24	23.5675	25830932
3	23.34	3/1/2017	23.6294	23.24	23.5675	25830932
4	24.34	3/1/2017	24.6294	24.24	24.5675	25840932
5	23.34	3/1/2017	23.6294	23.24	23.5675	25830932
6	21.34	3/1/2017	21.6294	21.24	21.5675	25790932
7	23.34	3/1/2017	23.6294	23.24	23.5675	25830932
8	26.34	3/1/2017	26.6294	26.24	26.5675	25860932
9	23.34	3/1/2017	23.6294	23.24	23.5675	25830932

STEP 2: Enabling Continuous Virtualization

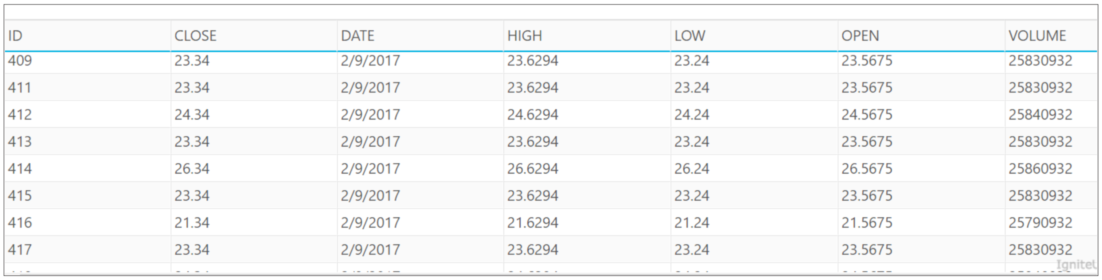
To enable continuous virtualization, set the following properties of the grid:

- Set rowVirtualization property to “true”
- Set virtulizationMode property to “continuous”
- Set height property to some pixel value (ere it will be set to “300px”)

To configure all of these properties of the grid, in the app folder: open the grid.component.ts file and update the getGridOptions() function as shown in the highlighted listing below

```
getGridOptions() {
   return {
      width: "100%",
      autoGenerateColumns: false,
      height: "300px",
      rowVirtualization: true,
      virtualizationMode: "continuous",
      columns: [
         { headerText: "ID", key: "Id", dataType: "string" },
         { headerText: "CLOSE", key: "Close", dataType: "number" },
         { headerText: "DATE", key: "Date", dataType: "string" },
         { headerText: "HIGH", key: "High", dataType: "number" },
         { headerText: "LOW", key: "Low", dataType: "number" },
         { headerText: "OPEN", key: "Open", dataType: "number" },
         { headerText: "VOLUME", key: "Volume", dataType: "number" }
      ]
   };
}
```

To test the continuous virtualization: navigate to the application, scroll down, and you will find the grid configured with continuous virtualization added as shown in the image below:



ID	CLOSE	DATE	HIGH	LOW	OPEN	VOLUME
409	23.34	2/9/2017	23.6294	23.24	23.5675	25830932
411	23.34	2/9/2017	23.6294	23.24	23.5675	25830932
412	24.34	2/9/2017	24.6294	24.24	24.5675	25840932
413	23.34	2/9/2017	23.6294	23.24	23.5675	25830932
414	26.34	2/9/2017	26.6294	26.24	26.5675	25860932
415	23.34	2/9/2017	23.6294	23.24	23.5675	25830932
416	21.34	2/9/2017	21.6294	21.24	21.5675	25790932
417	23.34	2/9/2017	23.6294	23.24	23.5675	25830932

In continuous virtualization, only a portion of the total rows in the data source are rendered in the DOM. As the user scrolls up and down on the grid, the virtualization feature determines if the current rows are sufficient to display the next/previous portion of rows. If new rows are required, the current portion of rows is deleted and the new portion of rows is created.

Conclusion

In any functional LOB application, you must render thousands of records in a grid. When an application is rendering thousands of records, the grid should be responsive during a rapid scroll of the grid. Achieve this by enabling the virtualization feature on the grid. In this lesson, you learned about configuring a grid for fixed and continuous virtualization.

Lesson

5

Run Fast with Large Sets of Data in Ignite UI Data Charts

Ignite UI Data Charts can render thousands of data points very smoothly and are fastest when rendering large sets of data.

Lesson Objectives

1. Configure a data chart to work with REST API
2. Create a data chart with large set of data

For more information on the controls used in this lesson, see *http://www.infragistics.com/products/IgniteUI/charts/data-chart*.

At the end of this lesson, you will have a data chart configured to work with large sets of data in an Angular application. You will see that even with a large amount of data, the chart renders quickly and zooming in and out of the chart is fluid and responsive.

Learn more about Ignite UI Angular 2 here: *https://github.com/IgniteUI/igniteui-angular2*; you can also learn more about Angular in *Angular Essentials, a* free eBook published by Infragistics.

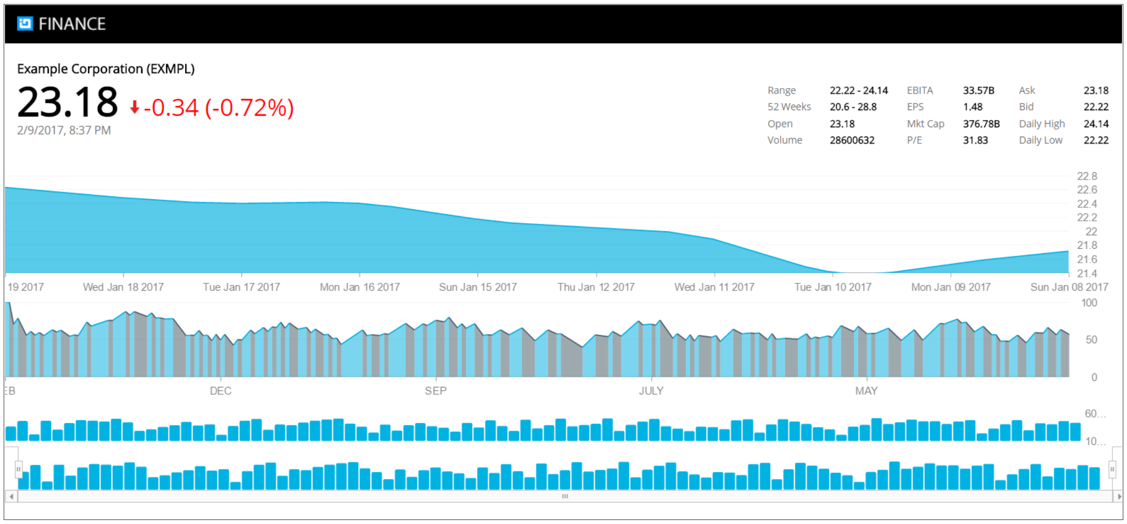
Setting up the Project

You can download the starting project for this lesson *by clicking here*. (You can also download the final project *by clicking here*.)

After you download the project, navigate to the directory and run the commands below:

```
npm install
npm start
```

You have executed the npm install command to install all dependencies and the npm start command to run the Angular application. If the project is setup correctly, you will have a running Angular application as shown in the image below. If you receive an error while running the application, stop and run the npm start command again.



FINANCE

Example Corporation (EXMPL)

23.18 -0.34 (-0.72%)

2/9/2017, 8:37 PM

Range	22.22 - 24.14	EBITA	33.57B	Ask	23.18
52 Weeks	20.6 - 28.8	EPS	1.48	Bid	22.22
Open	23.18	Mkt Cap	376.78B	Daily High	24.14
Volume	28600632	P/E	31.83	Daily Low	22.22

Right now, the first chart of the application is configured to work with a small data set. It is rendering a data source with ten rows and and functioning smoothly so you can zoom in and zoom out on the chart. Ignite UI data charts are created to work with large *and* small data sets. Whether you are rending 10 data points or 1000 data points, Ignite UI data charts will behave in the same smooth, seamless manner to help to run the application faster.

To see it in action, modify the chart to work with a large data set returning from a REST API.

STEP 1: Get Data in a Component

Currently the chart is configured is to work with a small data set, which is configured in the first line of code in the ngOnInit() function. To get a large data set in the Price-ChartComponent, you must use AppService in the component . To do so, in the app folder, open the pricechart.component.ts file, navigate to the ngOnInit() function, and (in the function) delete the first line of code and make a call to appService getStcoks() method. Replace only the first line of code as shown below and leave other codes of ngOnInit() to function as they are.

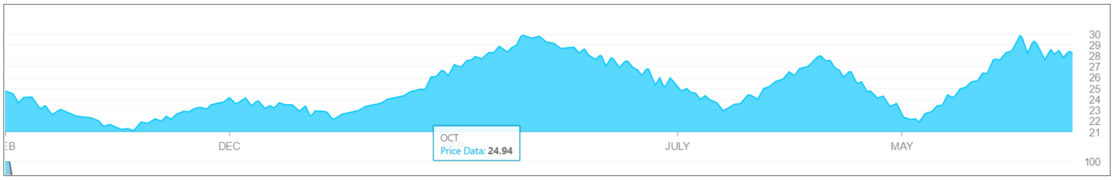
```
ngOnInit() {
  this._appService.getStocks()
     .subscribe(
     stocks => this.stocks = stocks,
        error => this.errorMessage = <any>error);
    this.desiredHeight = 0.22 * (window.screen.height) + "px";
    this.chartOptions = this.getPriceChartOptions();
}
```

Right now, you are fetching data from AppService in the PriceChartComponent. The AppService getStocks() method is fetching data from REST API, which has more than 200 data points. Essentially, you have reconfigured the chart to work with a large data set.

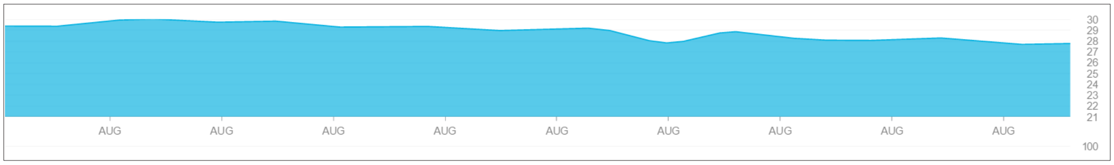
STEP 2: Run The Application

Navigate to the application and you will see that the Ignite UI data chart is rendering a large set of data very quickly and smoothly.

Now there are more than 200 data points rendered in the chart.



You can zoom out to a particular data point and the IgniteUI data chart will render in the same way and help the application run faster.



Conclusion

Ignite UI can be very useful in writing faster web applications. In addition to Angular, you can use Ignite UI in React, AngularJS, jQuery, and ASP.NET MVC. In this lesson, you learned how to use Ignite UI data charts with large sets of data in an Angular application. Various functionalities of Ignite UI data charts, such as zoom in and zoom out, work seamlessly with both small and large data sets.

Lesson

6

Zoom Fast with Ignite UI Zoombar

Ignite UI provides a Zoombar control to zoom range-enabled controls like data charts. Use Zoombar to zoom in on a widget in a resizable zoom-range window. Zoombar includes a horizontal scroll bar that can zoom either the whole range or a particular section of the chart. Zoombar works as a stand-alone control.

Learn more about other Ignite UI features here: *http://www.igniteui.com*

In this lesson, you will learn to configure Ignite UI Zoombar with a data chart.

Lesson Objective

1. Add Zoombar
2. Configure Zoombar with a Ignite UI data chart.

For more information on the controls used in this lesson, see *http://www.infragistics.com/products/IgniteUI /other-charts/zoombar*.

At the end of this lesson, you will have an Ignite UI data chart that is configured with Ignite UI Zoombar in an Angular application.

Learn more about Ignite UI Angular 2 here: *https://github.com/IgniteUI/igniteui-angular2*; you can also learn more about Angular in *Angular Essentials, a* free eBook published by Infragistics.

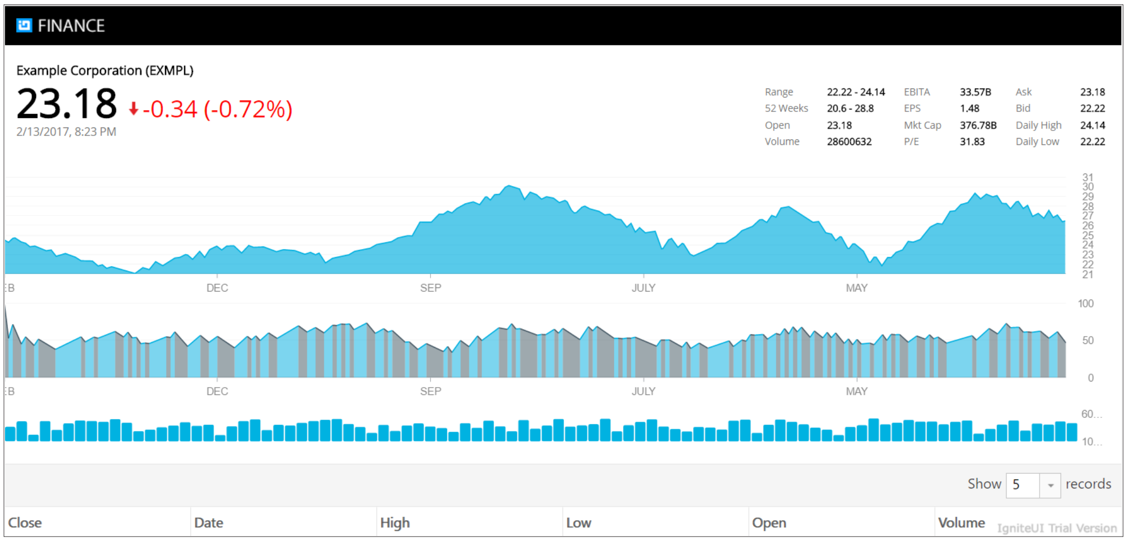
Setting up the Project

You can download the starting project for this lesson *by clicking here*. (You can also download the final project *by clicking here*.)

Next, navigate to the Finance App directory and run the commands below:

```
npm install
npm start
```

You have executed the npm install command to install all dependencies, and the npm start command to run the Angular application. If the project is setup correctly, you will have a running Angular application as shown in the image below. In addition, while working through the lesson, if you receive an error while running the application, stop and run the npm start command again.



FINANCE

Example Corporation (EXMPL)

23.18 ↓-0.34 (-0.72%)

2/13/2017, 8:23 PM

Range	22.22 - 24.14	EBITA	33.57B	Ask	23.18
52 Weeks	20.6 - 28.8	EPS	1.48	Bid	22.22
Open	23.18	Mkt Cap	376.78B	Daily High	24.14
Volume	28600632	P/E	31.83	Daily Low	22.22

Show 5 records

Close	Date	High	Low	Open	Volume

IgniteUI Trial Version

STEP 1: Import and Declare the Component

To work with Ignite UI Angular components, you must import and declare them in the module. For example, to use the igGrid component in an Angular application, import and declare the IgGridComponent in the application module.

Navigate to the Finance App folder and then the app folder. Open the file app.module.ts, and add below import statements, just after all of the existing import statements:

```
import { IgZoombarComponent } from 'igniteui-angular2';
```

After importing the required components, you must declare them in the application module. Add IgZoombarComponent in the AppModule's declaration array. Modify @ NgModule decorator in app.module.ts as shown below:

```
@NgModule({
  imports: [BrowserModule, HttpModule],
  declarations: [AppComponent,
    IgDataChartComponent,
    InfoComponent,
    IndicatorChartComponent,
    VolumeChartComponent,
    IgGridComponent,
    GridComponent,
    PriceChartComponent,
    IgZoombarComponent,
  ],
  providers: [AppService],
  bootstrap: [AppComponent]
})
```

You have added IgZoombarComponent in the declaration array of the AppModule module. Other added components and other properties, like providers, will be reviewed in subsequent lessons.

STEP 2: Add Zoombar

To work with Ignite UI Zoombar, you must first add the Zoombar component. In the app\charts folder, open the volumechart.component.html file and add the ig-zoombar control as shown below, just after the ig-data-chart control:

```
<ig-zoombar [(options)]="zoombarOptions" widgetId="zoombar"></ig-zoombar>
```

STEP 3: Add Zoombar Options Property

In the Zoombar option, you can attach a chart to the Zoombar.To configure the Zoombar option, create a property in the VolumeChartComponent class. In the app\charts folder, open the volumechart.component.ts file and, just above the constructor, add the property listed below:

```
private zoombarOptions: IgZoombar;
```

STEP 4: Attach Chart to Zoombar

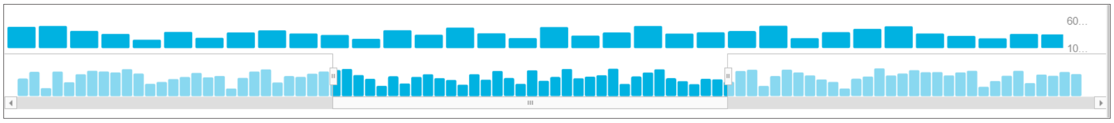
To attach a chart widget to Zoombar, you must set the target property value of Zoombar options to the ID of the chart widget. In the app\charts folder, open the volumechart.component.ts file and the code below just after the this.chartOptions assignment in the ngOnInit() function:

```
this.zoombarOptions = {
  target: "#volumechart"
};
```

In the above listing, volumechart is the ID of the data chart widget.

STEP 5: Run the Application

Navigate to application, scroll down, and at the bottom of the page, you will find Zoombar added as shown in the image below:



The chart has been cloned in the Zoombar and, by using the horizontal scroll bar, you can zoom the chart. You will find animation while zooming in and out is very fast and smooth. Regardless of doeshow many data points are rendered in the chart, Ignite UI Zoombar will zoom in on a particular section very quickly and with smooth animation.

Conclusion

Ignite UI can be very helpful in writing and running web applications more quickly. In addition to Angular, you may use Ignite UI in React, AngularJS, jQuery, and ASP.NET MVC. In this lesson, you learned how to configure Ignite UI Zoombar in an Angular application.

Lesson 7

Ignite UI With Different Package Managers

Ignite UI works with popular package managers to manage the dependencies of the project. The most popular package managers are:

- NPM
- Yarn
- JSPM

STEP 1: Working With NPM

In previous lessons, you have used NPM to work with Ignite UI controls. To see in how NPM works, download the *starter project* (you'll also find the final version of the project *here*), open the terminal, and run the command listed below:

```
npm install
```

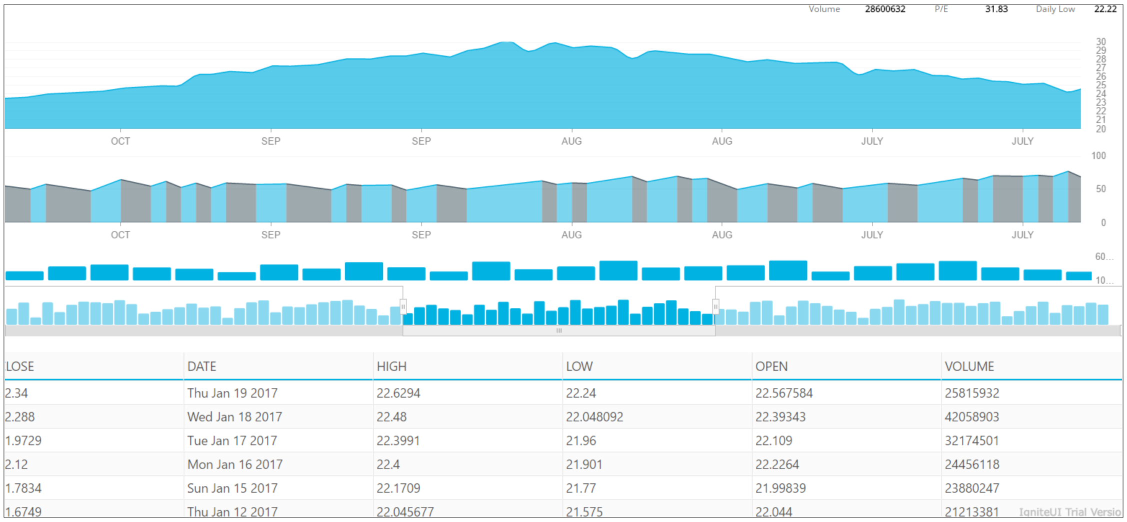
NPM install commands install the dependencies. It reads package.json file to install all the dependencies. to work with NPM, you must have NodeJS installed. If you do not have NodeJS installed, you can install it from *https://nodejs.org/en/*. After running NPM install command, you will find folder node_modules added in your project. This folder contains all the libraries installed using command NPM install.

If you are working on an existing project, then you may install the individual package in the project. To install the Ignite UI package individually, execute the following command:

```
npm install –save-dev igniteui-angular2
```

To run the application, execute the command below:

```
npm start
```



LOSE	DATE	HIGH	LOW	OPEN	VOLUME
2.34	Thu Jan 19 2017	22.6294	22.24	22.567584	25815932
2.288	Wed Jan 18 2017	22.48	22.048092	22.39343	42058903
1.9729	Tue Jan 17 2017	22.3991	21.96	22.109	32174501
2.12	Mon Jan 16 2017	22.4	21.901	22.2264	24456118
1.7834	Sun Jan 15 2017	22.1709	21.77	21.99839	23880247
1.6749	Thu Jan 12 2017	22.045677	21.575	22.044	21213381

STEP 2: Working with Yarn

NPM is one of the most popular package managers, but it has some shortcomings, including:

- Nested dependencies, which causes a long file path on Windows
- NPM does only sequential installation, so one package must be completely installed before moving to the install of the next package
- It can only install from the NPMSJS package and it does not have offline installation.

Yarn solves these problems. Yarn is a fast, reliable, and secure package manager. It takes packages from npmjs or bower registry. Although Yarn has advantages over NPM, NPM is still widely used and is the most popular package manager.

You can learn more about Yarn on their github page here: *https://github.com/Yarnpkg/Yarn*.

Like NPM, Yarn also reads package.json files of your project to install dependencies.

To work with Yarn, download the start project and open it in the terminal.

If you do not have Yarn installed on your machine, you must install it with the command below:

```
npm install –g yarn
```

NOTE: If you are using Apple's Mac OS, you may receive permission errors when you try to install global packages with NPM. If this happens, try:

```
sudo npm install –g yarn
```

After installing Yarn, you can use Yarn to install dependencies in your project. Like npm, Yarn also reads package.json to install dependencies. To install dependencies, run the command below:

```
yarn install
```

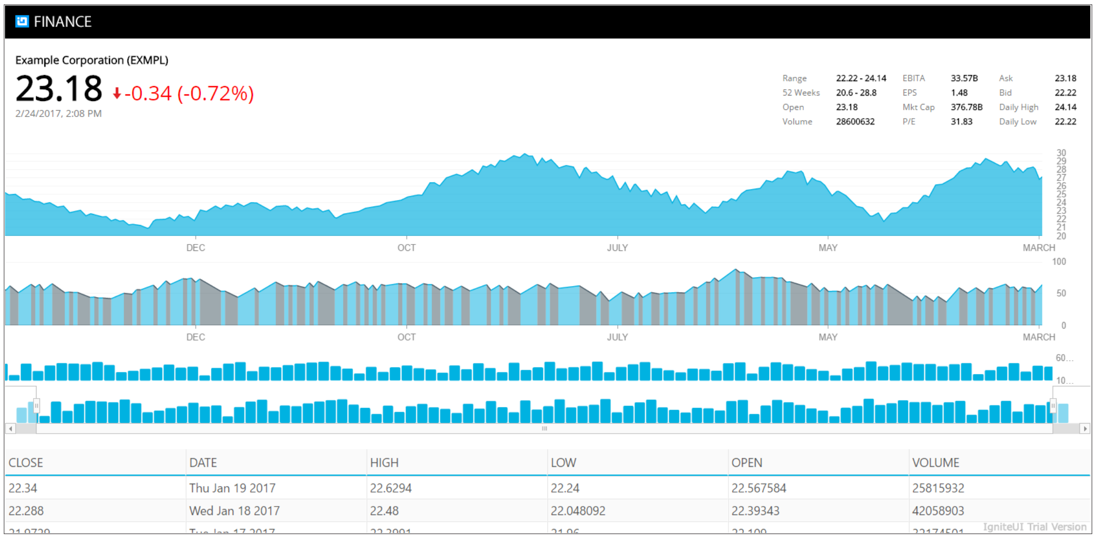
If you are working on an existing project, you may also install an individual package in the project. To install the Ignite UI package, execute the command below:

```
yarn add -dev igniteui-angular2
```

After successful installation, you will find a node_modules folder added in the project. To run the application, execute the command below:

```
yarn start
```

If everything is correct, the above command should start the application and you will have a running application as shown below:



Example Corporation (EXMPL)

23.18 ↓-0.34 (-0.72%)

2/24/2017, 2:08 PM

Range	22.22 - 24.14	EBITA	33.57B	Ask	23.18
52 Weeks	20.6 - 28.8	EPS	1.48	Bid	22.22
Open	23.18	Mkt Cap	376.78B	Daily High	24.14
Volume	28600632	P/E	31.83	Daily Low	22.22

CLOSE	DATE	HIGH	LOW	OPEN	VOLUME
22.34	Thu Jan 19 2017	22.6294	22.24	22.567584	25815932
22.288	Wed Jan 18 2017	22.48	22.048092	22.39343	42058903

Working With Dynamic Module Loaders

SystemJS is a module loader that can import a module at run-time and is built on the top of the ES6 module loader. It can transpile ES6 code or TypeScript. SystemJS can work with many types of modules formats such as AMD and CommonJS. SystemJS module loader can also work with Ignite UI modules and support it.

In the downloaded project, open the System.config.js file and you will find mapping for Ignite UI Angular 2 as shown in the listing below:

```
'igniteui-angular2': 'npm:igniteui-angular2'
```

In addition, you can find the Ignite UI package for loading as shown below:

```
'igniteui-angular2': {
   main: 'index.js',
   defaultExtension: 'js'
}
```

Due to packaging and mapping information in system.config.js, when the Angular application needs Ignite UI modules, it will be dynamically loaded by SystemJS in the application.

Conclusion

In web development, adding references of libraries has come a long way. It began with adding references manually in the project, then using Content Delivery Networks (CDN) to add references, and then various package managers such as bower, NPM, and Yarn came to existence. Ignite UI can be used with previous ways of managing packages like CDN or can be used with modern package managers such as NPM and Yarn. In addition to package managers, Ignite UI can be used with popular module loaders like SystemJS.

Lesson

8

Write React JS Apps with Ignite UI

Ignite UI fully supports modern web development. In addition to Angular, you can use Ignite UI library in React. This lesson will demonstrate how to use Ignite UI grid in a React application.

Lesson Objective

- Add Ignite UI grid in ReactJS
- Configure columns of the grid.

For more information on the controls used in this lesson, see *http://infragistics.com/products/IgniteUI/grids/data-grid*.

Setting up the Project

You can download the starting project for this lesson *by clicking here*. (You can also download the final project *by clicking here*.)

This project is already configured to work with ReactJS and Ignite UI and all references have been added to the project. You can learn more about using Ignite UI in ReactJS project here:

http://www.infragistics.com/community/blogs/igniteui_team/archive/2016/11/04/how-to-use-ignite-ui-components-with-react.aspx

In the *starting project for this lesson*, in addition to the React and Ignite UI libraries, you will find following files.

- **index.html :** contains application markup and references

- **index.js :** contains react code

- **data.js:** contains data to be used as data source of the Ignite UI grid.

In the project, data.js contains data to be rendered in the Ignite UI grid. File Index.js contains the App component. In index.js, you can find component class created as shown in the listing below:

```
 var App = React.createClass(
 {
    getInitialState: function()
    {
       return{
       }
    },
    render: function()
    {
       return(
          <h2>Ignite UI Grid will be rendered here</h2>
       )
    }
 });
ReactDOM.render(
 <App />,
 document.getElementById("app")
);
```

The above App component class contains two functions :

- The getInitialState() function simply returns an Object of initial state

- The render() function returns the description of what you want to render. In the next steps, we will render Ignite UI grid in the render function of the App component.

You can learn more about React.createClass API here: *https://facebook.github.io/react/docs/react-api.html*

On the index.html, as shown in the listing below, you will find that index.js has been referenced as babel script:

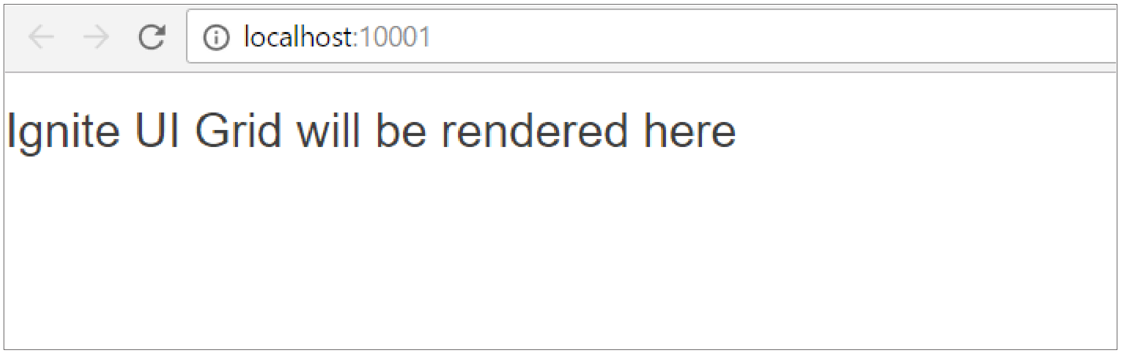
```
<div id="app">
  <script type="text/babel" src="index.js">
  </script>
</div>
```

In index.html, you will also find references of React, jquery, and Ignite UI libraries.

After downloading the project, navigate to the directory and run the commands below:

```
npm install
npm start
```

You have executed the npm install command to install lite server (web server) dependencies, using the npm start command to run the React application. If everything is correct, you will find a React application running in the browser as shown below:



Ignite UI Grid will be rendered here

STEP 1 : Initialize Initial State

To initialize the grid, you may want to set values for various properties of grid such as datasource, width, row styles, etc. You can set these grid properties in the getInitialState() function. Open index.js file, and modify the getInitialState() function with the code below.

```
getInitialState: function()
        {
          return{
            data: stocks,
            gridWidth: "100%",
            alternateRowStyles: true
          }
        },
```

You are creating properties to set the grid's width, row style, and data source. In addition, you will find there is already an array called "stocks" in the application.

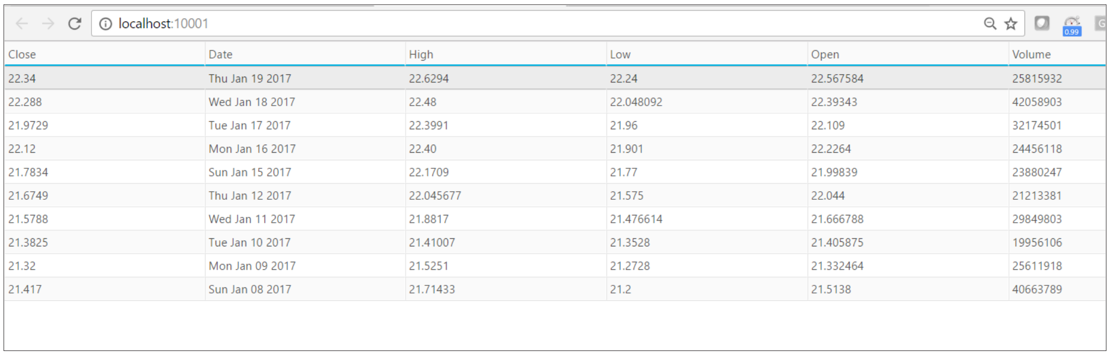
STEP 2: Render the Grid

To render the grid, you must return it from the render() function of the component class. To return IgGrid, open index.js and modify the render function as shown below:

```
render: function()
   {
   return(
   <div>
      <IgGrid id="grid1"
            autoGenerateColumns={true}
            dataSource={this.state.data}
            width={this.state.gridWidth}
            alternateRowStyles={this.state.alternateRowStyles} />
   </div>
   )
 }
```

You are setting dataSource, width, and alternateRowStyles properties with the properties of the object returned from the getInitialState() function.

Navigate to the application to find the RecatJS application running with the Ignite UI grid as shown in the image below:



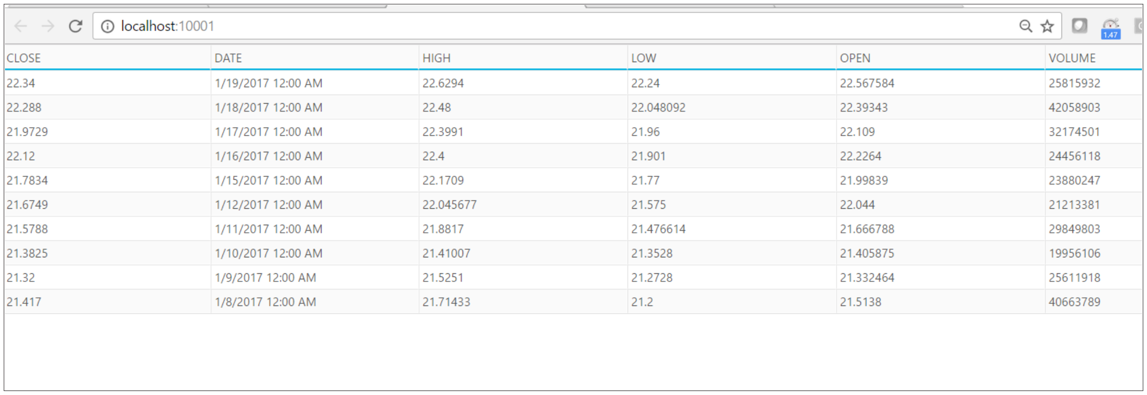
Close	Date	High	Low	Open	Volume
22.34	Thu Jan 19 2017	22.6294	22.24	22.567584	25815932
22.288	Wed Jan 18 2017	22.48	22.048092	22.39343	42058903
21.9729	Tue Jan 17 2017	22.3991	21.96	22.109	32174501
22.12	Mon Jan 16 2017	22.40	21.901	22.2264	24456118
21.7834	Sun Jan 15 2017	22.1709	21.77	21.99839	23880247
21.6749	Thu Jan 12 2017	22.045677	21.575	22.044	21213381
21.5788	Wed Jan 11 2017	21.8817	21.476614	21.666788	29849803
21.3825	Tue Jan 10 2017	21.41007	21.3528	21.405875	19956106
21.32	Mon Jan 09 2017	21.5251	21.2728	21.332464	25611918
21.417	Sun Jan 08 2017	21.71433	21.2	21.5138	40663789

STEP 3 CONFIGURE COLUMNS OF THE GRID

In the previous step, you set autoGenerateColumns to true in order to create a grid. You can also configure selected columns from the data set to display. To do so, you must configure columns for the Ignite UI Grid by setting the autoGenerateColumns property to false and adding the columns property in the grid. Modify the IgGrid in the render() function as shown in the listing below:

```
<IgGrid id="grid1"
       autoGenerateColumns={false}
       dataSource={this.state.data}
       width={this.state.gridWidth}
       alternateRowStyles={this.state.alternateRowStyles}
       columns={[
              { headerText: "CLOSE", key: "Close", dataType: "number" },
              { headerText: "DATE", key: "Date", dataType: "date", format: "dateTime" },
              { headerText: "HIGH", key: "High", dataType: "number" },
              { headerText: "LOW", key: "Low", dataType: "number"},
              { headerText: "OPEN", key: "Open", dataType: "number"},
              { headerText: "VOLUME", key: "Volume", dataType: "number"},
       ]}
  />
```

Navigate to the application and you will find that a grid has been configured with the columns.



CLOSE	DATE	HIGH	LOW	OPEN	VOLUME
22.34	1/19/2017 12:00 AM	22.6294	22.24	22.567584	25815932
22.288	1/18/2017 12:00 AM	22.48	22.048092	22.39343	42058903
21.9729	1/17/2017 12:00 AM	22.3991	21.96	22.109	32174501
22.12	1/16/2017 12:00 AM	22.4	21.901	22.2264	24456118
21.7834	1/15/2017 12:00 AM	22.1709	21.77	21.99839	23880247
21.6749	1/12/2017 12:00 AM	22.045677	21.575	22.044	21213381
21.5788	1/11/2017 12:00 AM	21.8817	21.476614	21.666788	29849803
21.3825	1/10/2017 12:00 AM	21.41007	21.3528	21.405875	19956106
21.32	1/9/2017 12:00 AM	21.5251	21.2728	21.332464	25611918
21.417	1/8/2017 12:00 AM	21.71433	21.2	21.5138	40663789

Conclusion

React is quickly becoming a very popular option for building client-side JavaScript applications. Enterprises are already looking at using React of their Line of Business applications. Ignite UI supports modern web development and its controls can be used with modern web development framework such as React

Lesson

9

Look Great With IgniteUI Themes

As a developer, you want to ensure that your application looks good and works on all types of devices, including desktops, tablets, and mobile devices. Modern web applications should be responsive and touch-enabled, but will require a lot of CSS/ SASS /LESS in your application. As a developer, you may not be skilled in CSS or have the time to learn it for use in your application. IgniteUI can help by providing various themes, which can be used as they are in your application or you can use the IgniteUI Theme Generator to create themes as required by your application.

Provided themes:

- Infragistics theme
- Metro theme
- iOS theme
- Default bootstrap theme
- Superhero bootstrap theme
- Yeti bootstrap theme
- Flatly bootstrap theme

In addition to these themes, you can use the IgniteUI Bootstrap Theme Generator to create your own theme. Learn more about IgniteUI Theme Generator at http://www.igniteui.com/bootstrap-theme-generator/Help; you can also learn more about Angular in *Angular Essentials, a* free eBook published by Infragistics.

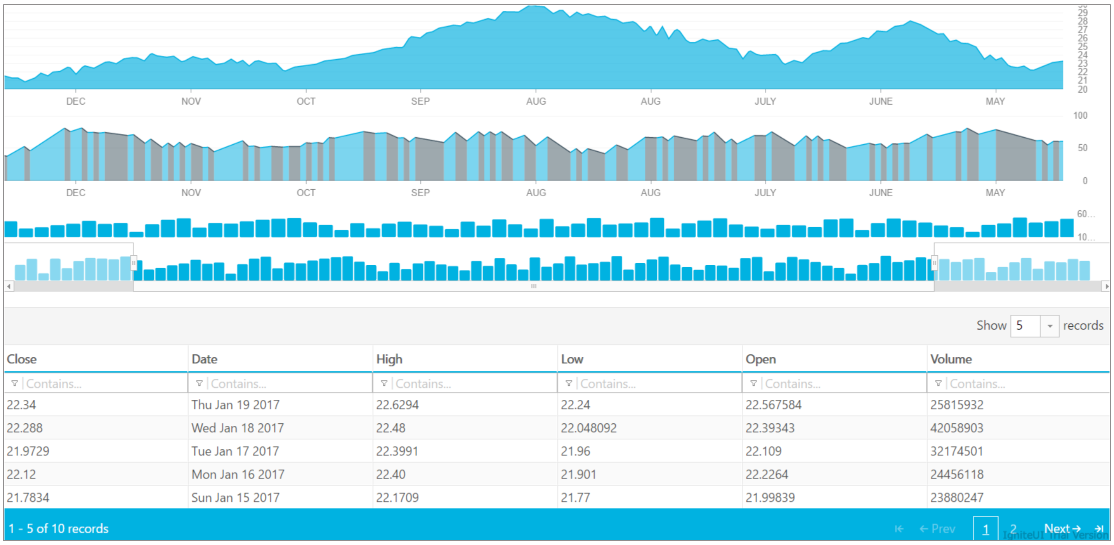
Setting Up The Project

You can download the starting project for this lesson *by clicking here*. (You can also download the final project *by clicking here*.)

After downloading the project, navigate to the directory and run the commands below:

```
npm install
npm start
```

You have executed the npm install command to install all dependencies and have used the npm start command to run the Angular application. If the project setup is correct, you will have a running Angular application as shown in the image below:



Show 5 records

Close	Date	High	Low	Open	Volume
Contains...	Contains...	Contains...	Contains...	Contains...	Contains...
22.34	Thu Jan 19 2017	22.6294	22.24	22.567584	25815932
22.288	Wed Jan 18 2017	22.48	22.048092	22.39343	42058903
21.9729	Tue Jan 17 2017	22.3991	21.96	22.109	32174501
22.12	Mon Jan 16 2017	22.40	21.901	22.2264	24456118
21.7834	Sun Jan 15 2017	22.1709	21.77	21.99839	23880247

1 - 5 of 10 records

The application is currently using the IgniteUI metro theme. In the project, open the index.html file, navigate to line number 9 to 10, or look for the CSS references in the head section. You will find that the application is referring to the metro theme from the IgniteUI CDN as shown in the listing below.

```
<link href="http://cdn-na.infragistics.com/igniteui/latest/css/themes/metro/infrag-
istics.theme.css" rel="stylesheet" />
<link href="http://cdn-na.infragistics.com/igniteui/latest/css/structure/infragistics.
css" rel="stylesheet" />
```

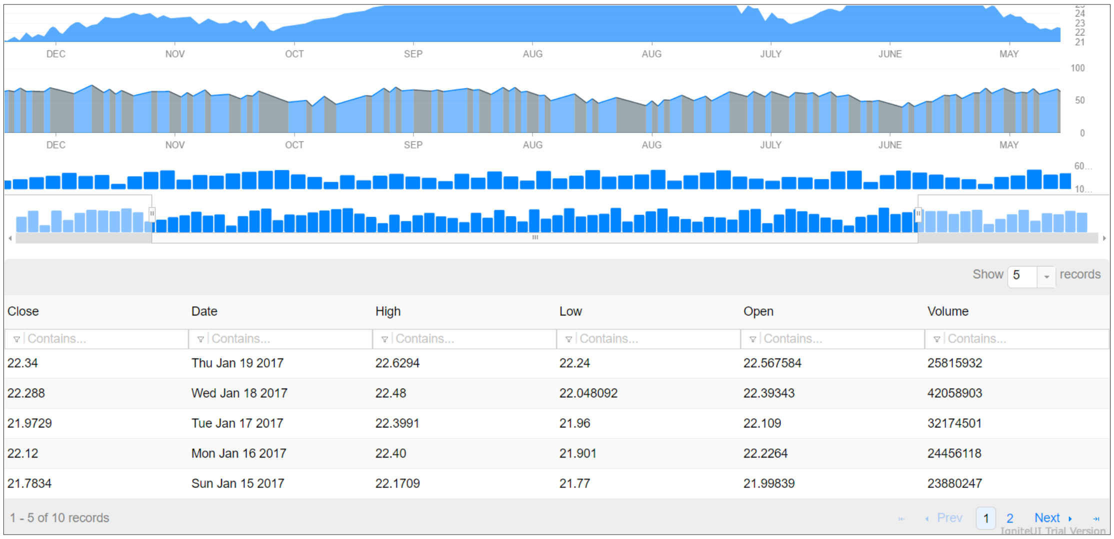
To work with any theme, you need a reference of Infragistics.css besides the theme reference.

STEP 1: Changing to iOS Theme

Change the existing IgniteUI themes easily by switching to the desired theme reference. To change the theme from the metro theme to the iOS theme, leave the reference of infragitics.css as it is and modify the IgniteUI theme reference in the index.html head section as shown in the listing below.

```
<link href="http://cdn-na.infragistics.com/igniteui/latest/css/themes/ios/infragistics.
theme.css" rel="stylesheet" />
```

Navigate to the application and you will find that all of the controls have been changed to the iOS theme. You may notice that the grid's look and the navigation button's design have been changed.



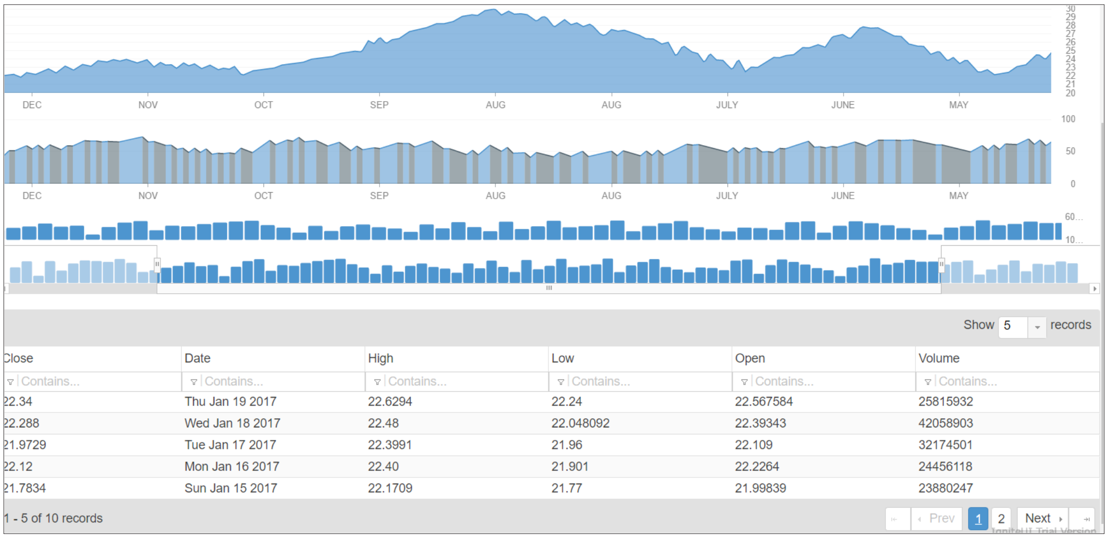
Close	Date	High	Low	Open	Volume
Contains...	Contains...	Contains...	Contains...	Contains...	Contains...
22.34	Thu Jan 19 2017	22.6294	22.24	22.567584	25815932
22.288	Wed Jan 18 2017	22.48	22.048092	22.39343	42058903
21.9729	Tue Jan 17 2017	22.3991	21.96	22.109	32174501
22.12	Mon Jan 16 2017	22.40	21.901	22.2264	24456118
21.7834	Sun Jan 15 2017	22.1709	21.77	21.99839	23880247

STEP 2: Changing to Default Bootstrap Theme

IgniteUI provides a default bootstrap theme. To change the theme to the bootstrap theme, leave the reference of infragitics.css as it is and modify the IgniteUI theme reference in the index.html head section as shown in the listing below.

```
<link href="http://cdn-na.infragistics.com/igniteui/latest/css/themes/bootstrap/
infragistics.theme.css" rel="stylesheet" />
```

Navigate to the application and you will find that all of the controls have been changed to the default basic bootstrap theme. The grid's look and the navigation button's design have been changed to the bootstrap theme.



Close	Date	High	Low	Open	Volume
Contains...	Contains...	Contains...	Contains...	Contains...	Contains...
22.34	Thu Jan 19 2017	22.6294	22.24	22.567584	25815932
22.288	Wed Jan 18 2017	22.48	22.048092	22.39343	42058903
21.9729	Tue Jan 17 2017	22.3991	21.96	22.109	32174501
22.12	Mon Jan 16 2017	22.40	21.901	22.2264	24456118
21.7834	Sun Jan 15 2017	22.1709	21.77	21.99839	23880247

STEP 3: Using Your Own Bootstrap Theme

IgniteUI helps you to create your own bootstrap-based theme. Simply upload a variables.less file in IgniteUI bootstrap theme generator and download the theme (combination of LESS, Complied CSS and images) to use in your application.

Learn more about IgniteUI theme generator here: *http://www.igniteui.com/bootstrap-theme-generator/Help* . IgniteUI theme generator helps you in two possible ways:

1. To customize existing IgniteUI themes
2. To create new bootstrap theme using the variables.less file.

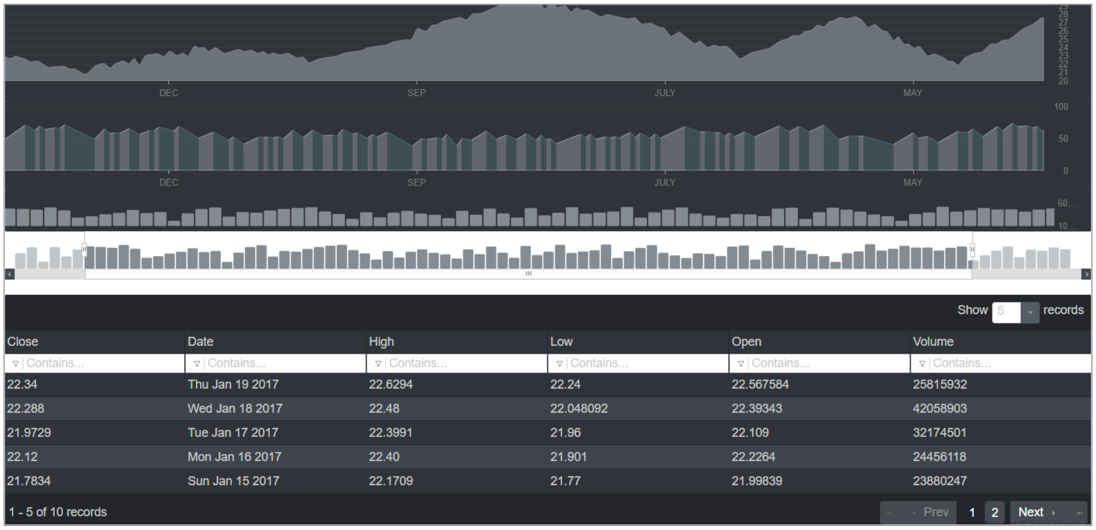
In previous steps you have used themes provided by IgniteUI. To use your own bootstrap theme, follow the steps as below. Note that, for this lesson, you do not have to perform these steps, as a bootstrap-based theme has been added in the project.

1. You can create your own variables.less file or use one of the bootstrap themes from *http://bootswatch.com* . To use a theme from bootswatch, select the theme and download the varibales.less file.
2. Upload variables.less file here: *http://www.igniteui.com/bootstrap-theme-generator/Theme/Upload* .
3. Download the theme and unzip it.
4. Save the downloaded theme in your application project.

The project contains a CSS folder, which contains a theme generated by the IgniteUI theme builder. To use this theme: in the head section of index.html add a reference of the theme and bootstrap as shown in the listing below. Delete reference of metro theme (line number 9) and add the below references just before the ./css/structure/infragistics.css reference.

```
<link href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css"
rel="stylesheet" type="text/css" />
<link href="css/themes/infragistics.theme.css" rel="stylesheet" type="text/css" />
```

Navigate to the application and you will find that application is using new theme.



Close	Date	High	Low	Open	Volume
Contains...	Contains...	Contains...	Contains...	Contains...	Contains...
22.34	Thu Jan 19 2017	22.6294	22.24	22.567584	25815932
22.288	Wed Jan 18 2017	22.48	22.048092	22.39343	42058903
21.9729	Tue Jan 17 2017	22.3991	21.96	22.109	32174501
22.12	Mon Jan 16 2017	22.40	21.901	22.2264	24456118
21.7834	Sun Jan 15 2017	22.1709	21.77	21.99839	23880247

1 - 5 of 10 records

Conclusion

In addition to the themes provided by IgniteUI, you can use your own themes or jQueryUI Theme Roller. Learn more here: *http://www.igniteui.com/help/deployment-guide-styling-and-theming#_Styling_and_Theming_IgniteUI*

IgniteUI supports the latest designs available in the modern web development and allows you to write web applications faster.

About the Authors

Rabi Kiran (a.k.a. Ravi Kiran) is a developer working on Microsoft Technologies at Hyderabad. He spends his time on JavaScript frameworks like AngularJS, latest updates to JavaScript in ES6 and ES7, Web Components, Node.js and also on several Microsoft technologies including ASP.NET 5, SignalR and C#. He is an active blogger, and an author at *DotNetCurry* and *SitePoint*. He is rewarded with the Microsoft MVP (Visual Studio and Dev Tools) and DZone MVB awards for his contribution to the community. Follow him on twitter @*sravi_kiran*

Mahesh Sabnis is a *DotNetCurry* author and Microsoft MVP having over 17 years of experience in IT education and development. He is a Microsoft Certified Trainer (MCT) since 2005 and has conducted various Corporate Training programs for .NET Technologies (all versions). Follow him on twitter @*maheshdotnet*

Suprotim Agarwal is the founder of *DotNetCurry*, *DNC Magazine for Developers*, *SQLServerCurry* and *DevCurry*. He has also authored a couple of books, *51 Chapters using jQuery with ASP.NET Controls* and *The Absolutely Awesome jQuery CookBook*. Suprotim has received the prestigious Microsoft MVP award for nine times in a row now. In a professional capacity, he is the CEO of A2Z Knowledge Visuals Pvt Ltd, a digital group that represents premium web sites and digital publication. Get in touch with him on Twitter @*suprotimagarwal* or at *LinkedIn* .