Angular Essentials

The Essential Guide to Learn Angular

By

Dhananjay Kumar

FIRST EDITION 2019

ISBN: 978-93-88511-24-7

Distributors:

BPB PUBLICATIONS
20, Ansari Road, Darya Ganj
New Delhi-110002
Ph: 23254990/23254991

BPB BOOK CENTRE
376 Old Lajpat Rai Market,
Delhi-110006
Ph: 23861747

MICRO MEDIA
Shop No. 5, Mahendra Chambers,
150 DN Rd. Next to Capital Cinema,
V.T. (C.S.T.) Station, MUMBAI-400 001
Ph: 22078296/22078297

DECCAN AGENCIES
4-3-329, Bank Street,
Hyderabad-500195
Ph: 24756967/24756400

Published by Manish Jain for BPB Publications, 20, Ansari Road, Darya Ganj, New Delhi-110002 and Printed by Repro India Pvt Ltd, Mumbai

I dedicate this book
to my Nani and my late Baba

Foreword

Angular (the modern web framework from Google) seems to have taken the world by storm in the last few years. With the move from AngularJS to Angular 2 to now Angular 7 (soon to be 8), it seems if you aren't building an Angular app you might feel behind the curve! But don't get too stressed out – you're not alone.

Technology is moving at such a fast pace; the web has evolved in amazing ways in the last decade but even more in the last 5 years. There're more JavaScript frameworks than ever to choose from, thousands of debates online about which framework is better, which is cooler, which has the most hype, and which are the most useful. It's never ending. If you didn't have a guide, a trusted advisor, you could end up with analysis-paralysis to move forward with your next project. You might never decide which direction to go. That's why I believe this book is hitting the market at just the right time. Angular has been established as the go-to framework for the Enterprise developer and is exploding in adoption all over the globe. It is helping teams everywhere modernize web experiences as they deliver next-generation apps that will fuel business for the next decade. With this book, you will have the ammunition you need to understand Angular, and build your next great app.

When Dhananjay (DJ) asked me to write the foreword to his new book, I was excited because I knew he is exactly the sort of expert that has helped tens of thousands of developers over the last 10 or more years wade through the confusion and mystery of where to go next, which web platform or JavaScript framework to use, but even more important, how to use it. DJ has worked almost full-time on Angular for 3 years, he's become known as an expert in the details and nuances of the various versions of the Angular framework. Before that, he was a general JavaScript addict, spending a lot of time deep in ASP.NET Web Forms, ASP.NET MVC and whatever the most recent platform was at the time. DJ's expertise and passion for helping others got him noticed from Microsoft – he was awarded the prestigious Microsoft MVP status almost 10 years ago. It is this love for teaching and helping others that brought him to write this book.

So, what will you learn in DJ's book? This book covers everything from the basics of Angular to more advanced features like working with

Services and Dependency Injection. There's even a bonus chapter where you'll learn how to add real business functionality with the Infragistics Angular Data Grid control. Once you make it through all the chapters, you'll be armed with everything you need to build applications in Angular, and then you can evangelize to others the way DJ has evangelized to you!

Will the web slow down? No. Technology continues to move forward, but that doesn't mean you need to race at the same pace. With a framework like Angular, and with the knowledge you'll gain in this book, be secure in the knowledge that you are using a great framework that has over 10 years of history and many, many more years ahead as the choice for development teams building enterprise apps.

Congratulations DJ on your new book, and I hope you (the reader) enjoys this book!

Jason Beres
Senior Vice President, Developer Tools @ Infragistics
February 2019, Cranbury, New Jersey
@jasonberes on Twitter

Contents

CHAPTER 1

Introduction

In this chapter, you will learn about what is Angular, how it is used, and setting up the development environment. Following topics will be covered in this chapter:

- What is Angular
- How is Angular different from AngularJS
- Angular's basic architecture
- Angular CLI
- Creating your First App

What is Angular

Angular is a widely used web application platform and framework created and maintained by Google. It serves as a total rewrite to AngularJS, and the *Angular* name is meant to include all versions of the framework starting from 2 and up.

TypeScript is the core of Angular, being the language upon which Angular is written. As such, Angular implements major and core functionalities as TypeScript libraries while building client applications with additional HTML.

For a variety of reasons, Angular has grown in popularity with developers. It lends itself to maintenance ease with its component and class-based system, modular building, hierarchical structure, and simple, declarative templates. Furthermore, its cross-platform capabilities are advantageous to enterprise and SMB developers, including its speed with server-side rendering.

Angular is a structural framework, which makes it easy to build dynamic web apps. It is a platform that gives developers the power to build applications that reside on web, mobile, or the desktop. In 2009, Angular started as a side-project by Miško Hevery and Adam Abrons to help

developers build applications using simple HTML tags. It was named Angular due to the fact that HTML tags are surrounded by these angle brackets <>. The team of developers at Google then kept on releasing the updated versions of Angular and at the time of writing this, current version being used is 7.0.

This essential book will go over the essential pieces of Angular and the main concepts behind working with the ever-growing platform for web-based applications.

How is Angular Different from AngularJS?

In the past, you might have worked with or learned about AngularJS. There are a few main differences between the two that you need to know about:

- **Modularity:** More of Angular's core functionalities have moved to modules.
- **Hierarchy:** Angular has an architecture built around a hierarchy of components.
- **Syntax:** Angular has a different expression syntax for event and property binding.
- **Dynamic loading:** Angular will load libraries into memory at run-time, retrieve and execute functions, and then unload the library from memory.
- **Iterative callbacks:** Using RxJS, Angular makes it easier to compose asynchronous or callback-based code.
- **Asynchronous template compilation:** Angular, without controllers and the concept of scope, makes it easier to pause template rendering and compile templates to generate the defined code.
- **TypeScript:** Angular includes ES6 and its superset, TypeScript.

Angular's Basic Architecture

Here's a brief overview of the architecture involved and the building blocks that I'll cover in this piece:

- **NgModules:** Declares a compilation context for a set of components that is dedicated to an application domain, a workflow, or a related set of capabilities.
- **Components:** Defines a class that contains application data and logic and works with an HTML template that defines a view.

- **Template:** Combines HTML with Angular markup that can modify HTML elements before they're displayed.
- **Directive:** Attaches custom behavior to elements in the DOM.
- **Two-way data binding:** Coordinates the parts of a template with the parts of a component.
- **Services:** Typically, a class used to increase modularity and reusability with a narrow and well-defined purpose.
- **Dependency injection:** Provides components with needed services and gives access to a service class.
- **Routing:** Defines a navigation path among the different application states lets you view application hierarchies.

Angular CLI

Setting up an Angular project requires many steps, such that:

- Setting up TypeScript compiler
- Setting up Webpack
- Setting up local web development server
- Configuring Unit Test environment

All these tasks are taken care by Angular CLI. Angular Command Line Interface is a command line tool for creating Angular apps. It is recommended to use angular CLI for creating angular apps as you don't need to spend time installing and configuring all the required dependencies and wiring everything together. It provides you with boilerplates and saves your time. It uses Webpack to include all the packaging, importing, BrowserLink etc. all the Webpack configuration is done completely by CLI and the developer needs not worry about it then. It also configures Jasmine and Karma for unit tests and TypeScript complier to transpile TypeScript file to JavaScript. You can install Angular CLI globally on local development machine using npm. To work with npm, make sure to install NodeJS from here : https://nodejs.org/en/ . Once NodeJS is installed, you can use npm to install Angular CLI:

```
npm install -g @angular/cli
```

Here, npm install command is to use npm to install angular CLI, -g is used to install CLI globally in your machine. After the process completes, you can check if it is successfully installed and also the version of the CLI is installed. To check that, we use the following command:

```
ng v or ng version
```

This gives a preview like in *figure 1.1*.

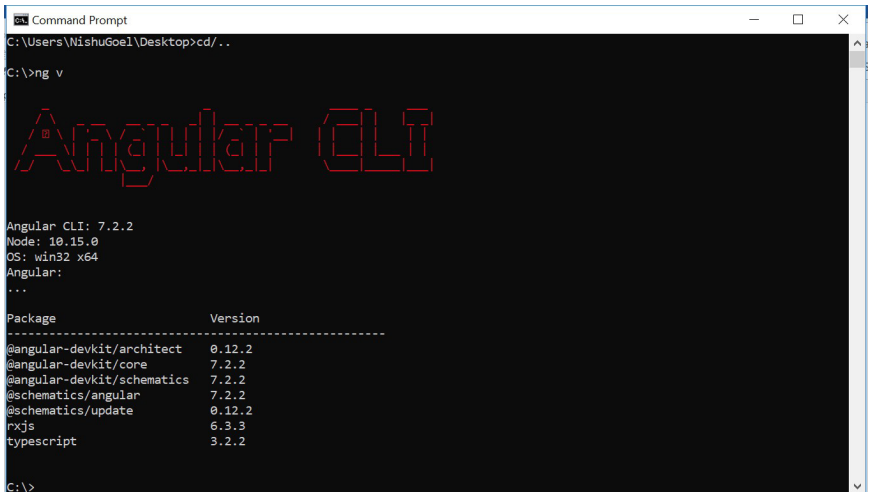


Figure 1.1

In here, we can then check the version of Angular CLI installed. We can also see the node version here with the versions of different Angular packages like core, rxjs, typescript, to name a few.

We can also update the CLI using the following steps:

```
npm uninstall -g @angular/cli
npm cache clean
npm install -g @angular/cli
```

CLI comes to great use when generating components, services, directives, modules etc. There are so many commands that can be used with the help of CLI to generate various Angular features:

Add Component

The command used to *add/generate* a component using CLI is:

```
ng generate component <component-name>
```

or

```
ng g c <component-name>
```

Add Module

The command used to add a module to our Angular application is:

ng generate module *<module-name>*

or

ng g m *<module-name>*

Add Directive

The command which is used to add directives to an angular application is as follows:

ng generate directive *<directive-name>*

or

ng g d *<directive-name>*

Add Service

The command to add service using CLI is:

ng generate service *<service-name>*

or

ng g s *<service-name>*

There are many commands Angular CLI offers, which help you to speed up Angular application development. You can learn about them in details by following the given link: https://cli.angular.io/

Create First Angular App

Now that we have understood the importance of CLI in working with Angular applications, let us create our first angular application using angular CLI.

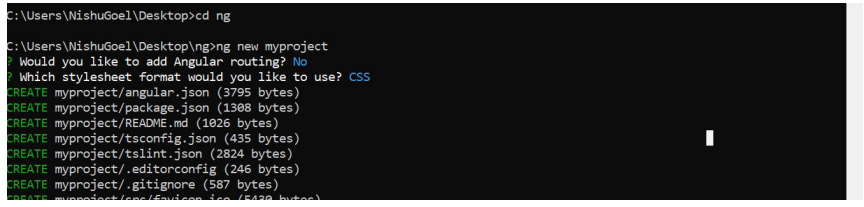


Figure 1.2

To create a new angular app, we use the command:

```
ng new <app-name>
```

ng new allows you to generate a new angular project with all the boilerplate files already generated for you. On the latest version, it asks before proceeding like *figure 1.2*.

Once started, the CLI gives us results like in *Figure 1.3*:

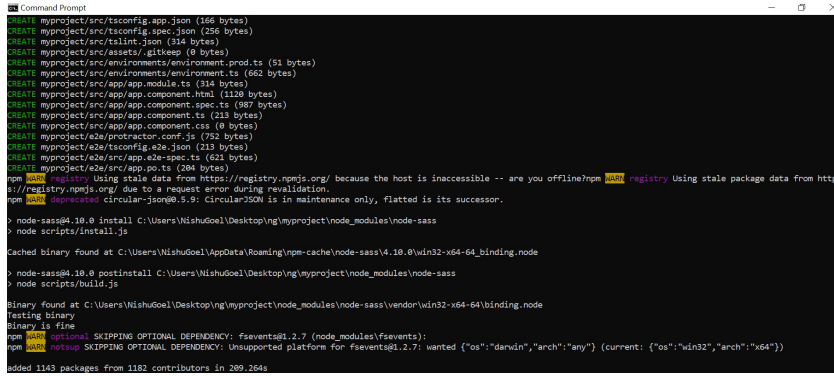


Figure 1.3

To open this newly created app inside Visual Studio code, we use **code followed by a dot.**

Inside VS Code, we can see the structure of the newly created project like in *figure 1.4*.

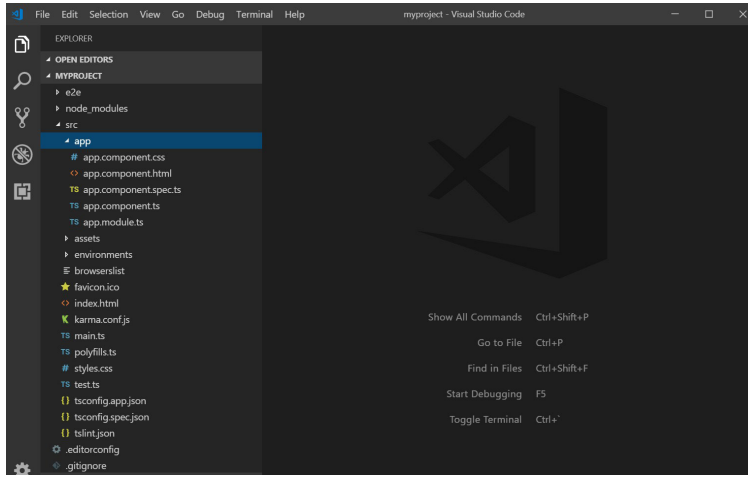


Figure 1.4

Now to compile and run our application, we use the command:

```
ng serve
```

We can also change the port no., in case one is already in use. To do that, we use the command:

```
ng serve -- port 4300
```

This gives us results like in *figure 1.5.*

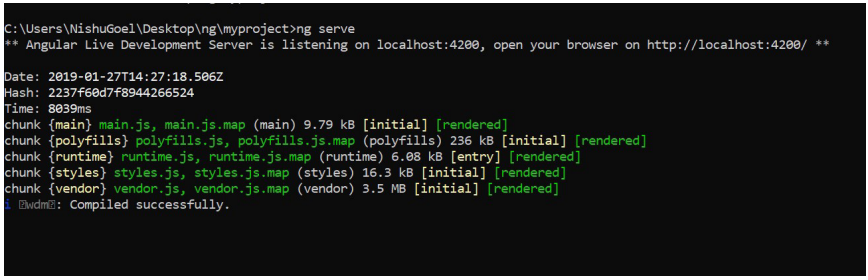


Figure 1.5

The URL is mentioned in the results. In my case, it is **`localhost:4200`**. And now can go to the browser with the URL and check the CLI-created Angular app in *figure 1.6.*

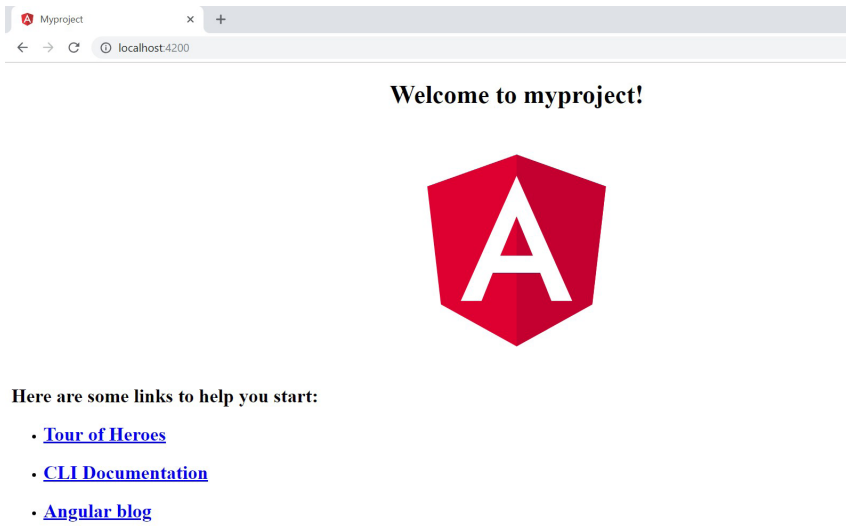


Figure 1.6

Summary

In this chapter, we learnt about theoretical concepts of Angular. A solid understanding of theory of Angular helps you to write Angular application faster and in better way. In this chapter, you learnt about the following topics:

- What is Angular
- How is Angular different from AngularJS
- Angular's basic architecture
- Angular CLI
- Creating your First App

——****——

CHAPTER 2

Component and Data Binding

In this chapter, you will learn about Angular Components and various Data Binding techniques. Following topics will be covered in this chapter:

- Component
- What is Data Binding
- Interpolation
- Property Binding
- Event Binding
- Two way data binding with [(ngModel)]
- Two way data binding without [(ngModel)]

Component

In Angular applications, what you see on the browser (or elsewhere) is a component. A component consists of the following parts:

1. A TypeScript class is called Component class
2. A HTML file is called Template of the component
3. An optional CSS file for the styling of the component

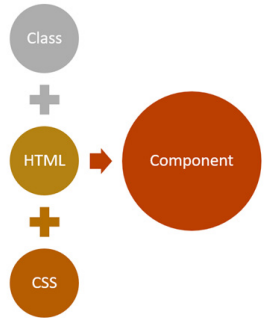


Figure 2.1

Components are type of directives, which has its own template. Whatever you see in an Angular application is a component.

Creating a Component

You can use Angular CLI command to generate a component as:

`ng` Generate Component Product

This command will generate **`ProductComponent`** as shown in *Code Listing 2.1*:

Code Listing 2.1

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

@Component({
  selector: 'app-product',
  templateUrl: './product.component.html',
  styleUrls: ['./product.component.scss']
})
export class ProductComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

A component is a class decorated with **`@Component`** decorator. There are mainly four steps to create a component:

1. Create a class and export it. This class will contain data and the logic.
2. Decorate the class with **`@component`** metadata. Metadata describes the component and sets the value for different properties.
3. Import the required libraries and modules to create the component.
4. Create template of the component and optionally style of the component.

As you can see, generated **`ProductComponent`** consists of:

- A class to hold data and the logic;

- HTML template and styles to display data in the app. It is also called as a view, which is seen by the user on the screen to interact.
- Metadata, which defines the behavior of a component. Component metadata is applied to the class using the @Component decorator. Different behaviors of the component can be passed as properties of the object, which is an input parameter of the @Component decorator.

Component Metadata

@Component decorator decorates a class as a component. It is a function, which takes an object as a parameter. In the @Component decorator, we can set the values of different properties to set the behavior of the component. The most used properties are as follows:

- template
- templateUrl
- Providers
- styles
- styleUrls
- selector
- encapsulation
- changeDetection
- animations
- viewProviders

Apart from the above mentioned properties, there are other properties also. Now let us look into these important properties one by one.

Template and TemplateUrl

A template is the part of the component which gets rendered on the page. We can create a template in two possible ways:

1. Inline template : template property
2. Template in an external file : templateUrl property

To create inline template **tilt** symbol is used to create multiple lines in the template. A single line inline template can be created using either single quotes or double quotes. For inline template set value of template property. Complex template can be created in an external HTML file and can be set using templateUrl property.

Selector

A component can be used using the selector. In the above example, the selector property is set to `<app-product>`. We can use the component on template of other components using its selector.

Styles and StyleUrls

A component can have its own styles or it can refer to various other external style sheets. To work with styles, @Component metadata has styles and styleUrls properties. We can create inline styles by setting the value of the styles property. We can set external style using styleUrls property.

Providers

To inject a service in a component, you pass that to providers array. Component metadata has an array type property called the provider. In the providers, we pass a list of services being injected in the component. We will cover this in detail in further sections.

ChangeDetection

This property determines how change detector will work for the component. We set `ChangeDetectionStrategy` of the component in the property. There are two possible values:

1. Default
2. onPush

We will cover this property in detail in further sections.

Encapsulation

This property determines whether Angular will create shadow DOM for component or not. It determines `ViewEncapsulation` mode of the component. There are four possible values:

1. Emulated this is default
2. Native
3. None
4. ShadowDom

Template

When you generate a component using Angular CLI, by default selector, `templateUrl`, and `styleUrl` properties are set. For `ProductComponent`

template is in external HTML file **`product.component.html`** as shown in *Code Listing 2.2.*

Code Listing 2.2

```
<p>
  product works!
</p>
```

You can pass data and capture events between component class and its template using Data Binding. We will cover this in detail, in further sections.

Using a Component

A component can be used inside an Angular application in various ways:

- As a root component.
- As a child component. We can use a component inside another component.
- Navigate to a component using Routing. In this case, component will be loaded in **RouterOutlet**.
- Dynamically loading component using **`ComponentFactoryResolver.`**

Component must be part of a module and to use a component in a module, first import that and then pass it to declaration array of the module. Refer *Code Listing 2.3*

Code Listing 2.3

```
@NgModule({
    declarations: [
        AppComponent,
        ProductComponent
    ],
```

Data Binding

In Angular, Data Binding determines how data will flow in between Component class and Component Template.

Angular provides us with the three types of data bindings. They are as follows:

- Interpolation
- Property Binding
- Event Binding

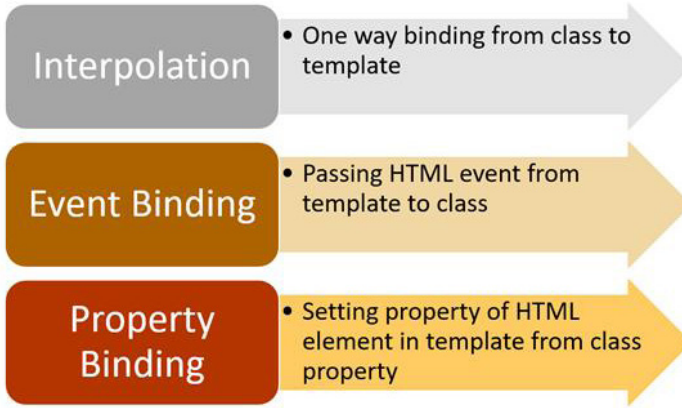


Figure 2.2

Let's see each, one by one.

Interpolation

Angular interpolation is one-way data binding. It is used to pass data from component class to the template. The syntax of interpolation is **`{{propertyname}}`**.

Let's say, we have component class as shown in *code listing 2.4*:

Code Listing 2.4

```
export class AppComponent {

      product = {
          title: 'Cricket Bat',
          price: 500
      };

  }
```

We need to pass the product from the component class to the template. Keep in mind that to keep example simple, I'm hard coding the value of the product object, however, in a real scenario, data could be fetched from

the database using the API. We can display value of the product object using interpolation, as shown in the **code listing 2.5**:

Code Listing 2.5

```
<h1>Product</h1>
<h2>Title : {{product.title}}</h2>
<h2>Price : {{product.price}}</h2>
```

Using interpolation, data is passed from the component class to the template. Ideally, whenever the value of the product object is changed, the template will be updated with the updated value of the product object.

In Angular, there is something called **`ChangeDetector`** Service, which makes sure that value of property in the component class and the template are in sync with each other.

Therefore, if you want to display data in Angular, you should use interpolation data binding.

Property Binding

Angular provides you with a second type of binding called *Property Binding*. The syntax of property binding is the square bracket `[]`. It allows to set the property of HTML elements on a template with the property from the component class.

So, let's say that you have a component class like *Code Listing 2.6.*

Code Listing 2.6

```
export class AppComponent {

        btnHeight = 100;
        btnWidth = 100;
    }
```

Now, you can set height and width properties of a button on template with the properties of the component class using the property binding as shown in *Code Listing 2.7.*

Code Listing 2.7

```
<button
  [style.height.px] = 'btnHeight'
```

```
    [style.width.px] = 'btnWidth' >
        Add Product
</button >
```

Angular Property Binding is used to set the property of HTML Elements with the properties of the component class. You can also set properties of other HTML elements like image, list, table, etc. Whenever the property's value in the component class changes, the HTML element property will be updated in the property binding.

Event Binding

Angular provides you the third type of binding to capture events raised on template in a component class. For instance, there's a button on the component template and, on the click of the button, you want to call a function in component class. You can do this using Event Binding. The syntax behind Event Binding is (**eventname**).

For this example, you might have a component class like this as shown in *Code Listing 2.8*.

Code Listing 2.8

```
export class AppComponent {

     addProduct() {
         console.log('add product');
     }

 }
```

You want to call **`addProduct`** function on the click of the button on the template. You can do this using event binding: like *Code Listing 2.9*.

Code Listing 2.9

```
<h1>Product</h1>
<button (click)='addProduct()'>
    Add Product
</button>
```

Angular provides you these three bindings. In event binding, data flows from template to class and, in property binding and interpolation, data flows from class to template. Refer *figure 2.3*:

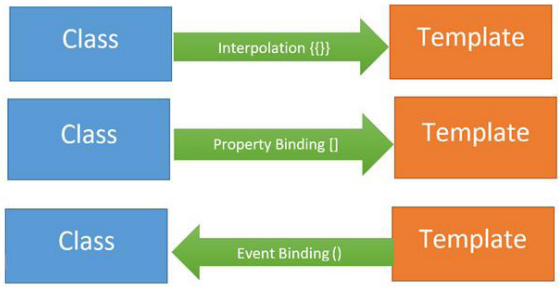


Figure 2.3

Two-Way Data Binding

Angular does not have built-in two-way data binding, however, by combining Property Binding and Event Binding, you can achieve Two-Way Data Binding.

Figure 2.4

Angular provides us a directive, ngModel, to achieve two-way data binding, and It's very easy to use. First, import **Forms** Module and then you can create two-way data binding as shown in *Code Listing 2.10.*

Code Listing 2.10

```
export class AppComponent {

     name = 'foo';
 }
```

We can, in two-way, data bind the name property with an input box as shown in *Code Listing 2.11.*

Code Listing 2.11

```
<input type="text" [(ngModel)]='name' />

<h2>{{name}}</h2>
```

As you see, we are using **`[(ngModel)]`** to create two-way data binding in between input control and name property. Whenever a user changes the value of the input box, the name property will be updated and vice versa.

Two-way Data Binding without `ngModel`

To understand **`ngModel`** directive working, let us see how we can achieve two-way data binding without using **`ngModel`** directive. To do that, we need to use the following:

- Property binding to bind expression to value property of the input element. In this demo, we are binding name variable expression to value property.
- Event binding to emit input event on the input element. Yes, there is an input event which will be fired whenever user will input to the input element. Using event binding, input event would be bind to an expression.

So, using the property binding and the event binding, two-way data binding can be achieved as shown in the *Code Listing 2.12*.

Code Listing 2.12

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

@Component({
    selector: 'app-root',
    template: `
          <div class="container">
           <input [value]="name" (input)="name=$event.
target.value" />
              <br/>
              <h1>Hello {{name}}</h1>
          </div>
          `
})
export class AppComponent {

        name = '';
}
```

Just like **`ngModel`** directive demo, in this demo also, when typing into the input element, the input element's value will be assigned to name variable and also it would be displayed back to the view.

Let us understand few important things here:

- **`[value]="name"`** is the property binding. We are binding value property of the input element with variable (or expression) name.
- **`(input)= "expression"`** is event binding. Whenever input event will be fired expression will be executed.
- **`'name=$event.target.value'`** is an expression which assigns entered value to name variable.
- Name variable can be accessed inside **`AppComponent`** class.

Summary

In this chapter, we learnt about Angular Components and various Data Binding techniques. A good understanding of data bindings is important to leverage the other features of Angular. In this chapter you learnt about following topics:

- Component
- What is Data Binding
- Interpolation
- Property Binding
- Event Binding
- Two way data binding with **`[(ngModel)]`**
- Two way data binding without **`[(ngModel)]`**

——****——

CHAPTER 3

Components Communications

In this chapter, you will learn about Angular Components communication. Following topics will be covered in this chapter:

- Component Communication
- @Input
- @Output and Event Emitter
- Temp Ref Variable

Component Communication

In Angular, components communicate with each other to share data such as object, string, number, array, or html.

To understand component communication, first we need to understand the relationship between components. For example, when two components are not related to each other, they communicate through Angular Service.

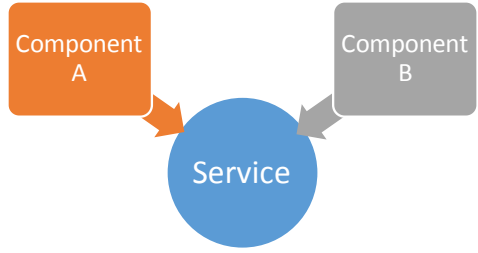


Figure 3.1

When you use a component inside another component, thus creating a component hierarchy, the component being used inside another component is known as the child component and the enclosing component is known as the parent component. As shown in the *figure 3.2*, in context

of **`AppComponent`**, app-child is a child component and **`AppComponent`** is a parent component.

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

@Component({
  selector: 'app-root',
  template: `
  <h1>Hello {{message}}</h1>
  <app-child></app-child>
  `,
})
export class AppComponent {

  message = 'I am Parent';

}
```

Child Component

Parent Component

Figure 3.2

Parent and Child components can communicate with each other in following ways:

- @Input()
- @Output()
- Temp Ref Variable
- ViewChild and ContentChild

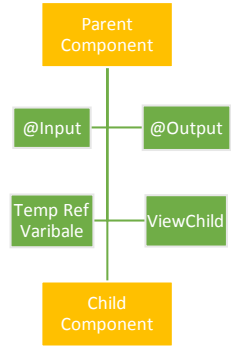


Figure 3.3

When components are not related to each other, they communicate using services. Otherwise, they communicate using one of the various options depending on the communication criteria. Let us explore all options one by one.

@Input

You can pass data from parent component to child component using **@Input** decorator. Data could be of any form such as primitive type's string, number, object, array etc.

Figure 3.4

To understand the use of **@Input**, let us create a component as shown in *code listing 3.1*.

Code Listing 3.1

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

@Component({
    selector: 'app-child',
    template: `<h2>Hi {{greetMessage}}</h2>`
})
export class AppChildComponent {

    greetMessage = 'I am Child';

}
```

Use **`AppChild`** component inside **`AppComponent`** as shown in *code listing 3.2*.

Code Listing 3.2

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

@Component({
  selector: 'app-root',
  template: `
  <h1>Hello {{message}}</h1>
  <app-child></app-child>
  `,
})
```

```
export class AppComponent {

  message = 'I am Parent';

}
```

AppComponent is using **AppChildComponent**, hence **AppComponent** is the parent component and **AppChildComponent** is the child component. To pass data, @**Input** decorator uses the child component properties. To do this, we'll need to modify child **AppChildComponent** as shown in the *code listing 3.3*.

Code Listing 3.3

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

@Component({
    selector: 'app-child',
    template: `<h2>Hi {{greetMessage}}</h2>`
})
export class AppChildComponent implements OnInit {
    @Input() greetMessage: string;
    constructor() {

    }
    ngOnInit() {

    }
}
```

As you notice, we have modified the **greetMessage** property with the **@Input()** decorator. So essentially, in the child component, we have decorated the **greetMessage** property with the **@Input()** decorator so that value of the **greetMessage** property can be set from the parent component. Next, let us modify the parent component **AppComponent** to pass data to the child component as shown in *code listing 3.4*.

Code Listing 3.4

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

@Component({
```

```
  selector: 'app-root',
  template: `
  <h1>Hello {{message}}</h1>
  <appchild [greetMessage]="childmessage"></appchild>
  `,
})
export class AppComponent {

  message = 'I am Parent';
   childmessage = 'I am passed from Parent to child
component';

}
```

From the parent component, we are setting the value of the child component's property **`greetMessage`**. To pass a value to the child component, we need to pass the child component property inside a square bracket and set its value to any property of parent component. We are passing the value of the **`childmessage`** property from the parent component to the **`greetMessage`** property of the child component.

Intercept input from Parent Component in the Child Component

We may have a requirement to intercept data passed from the parent component inside the child component. This can be done:

1. Using @**`Input`** decorator on getter and setter.
2. Using **`ngOnChanges()`** life cycle hook.

We will discuss about **`ngOnChanges`** life cycle hook in further sections. However, let us see how we can use **`@Input`** with setter to intercept passed data to the child component. We have modified **`AppComponent`** as shown in the *code listing 3.5*.

Code Listing 3.5

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

@Component({
  selector: 'app-root',
  template: `
  <h1>Hello {{message}}</h1>
```

```
 <app-child *ngFor="let n of childNameArray" [Name]="n">
  </app-child>
  `,
})
export class AppComponent {

  message = 'I am Parent';
   childmessage = 'I am passed from Parent to child
component';
  childNameArray = ['foo',
                    'koo',
                    ' ',
                    'moo',
                    'too',
                    'hoo',
                    ''
                   ];
}
```

Inside **AppComponent**, we are looping the **AppChildComponent** through all items of **childNameArray** property. A few items of the **childNameArray** are empty strings, these empty strings would be intercepted by the child component setter and set to the default value.

Let us modify **AppChildComponent** to use **@Input** decorator with setter and getter as shown in the *code listing 3.6.*

Code Listing 3.6

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

@Component({
    selector: 'app-child',
    template: `<h2> {{_name}}</h2>`
})
export class AppChildComponent implements OnInit {
    _name: string;
    constructor() {

    }
```

```
    ngOnInit() {

    }

    @Input()
    set Name(name: string) {
        this._name = (name && name.trim()) || 'default
name';
    }
    get Name() {
        return this._name;
    }
}
```

As you notice in the **@Input()** setter, we are intercepting the value passed from the parent, and checking whether it is an empty string. If it is an empty string, we are assigning the default value for the name in the child component.

In this way, **@Input** can be used to pass data to the child component.

@Output

You can emit event from child component to parent component using **@Output** decorator.

Figure 3.5

Angular is based on a one-directional data flow and does not have two-way data binding. So, we use **@Output** in a component to emit an event to another component. Let us modify **AppChildComponent** as shown in the *code listing 3.7*.

Code Listing 3.7

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

@Component({
```

```
    selector: 'app-child',
     template: `<button (click)="handleclick()">Click
me</button> `
})
export class AppChildComponent {

    handleclick() {

        console.log('hey I am  clicked in child');
    }
}
```

There is a button in the **`AppChildComponent`** template which is calling the function **`handleclick`**. Let's use the app-child component inside the **`AppComponent`** as shown in *code listing 3.8*.

Code Listing 3.8

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-root',
    template: `<app-child></app-child>`
})
export class AppComponent implements OnInit {
    ngOnInit() {
    }
}
```

Here we're using **`AppChildCopmponent`** inside **`AppComponent`**, thereby creating a parent-child kind of relationship, in which **`AppComponent`** is the parent and **`AppChildComponent`** is the child. When we run the application with a button click, you'll see this message in the browser console as shown in *figure 3.6*.

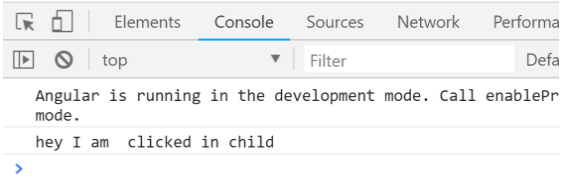


Figure 3.6

So far, it's very simple to use event binding to get the button to call the function in the component. Now, let's tweak the requirement a bit. What if you want to execute a function of **`AppComponent`** on the click event of a button inside **`AppChildComponent`**?

To do this, you will have to emit the button click event from **`AppChildComponent`**. Import **`EventEmitter`** and Output from **@ angular/core**.

Here we are going to emit an event and pass a parameter to the event. Modify **`AppChildComponent`** as shown in *code listing 3.9*.

Code Listing 3.9

```
import { Component, EventEmitter, Output } from '@
angular/core';
@Component({
    selector: 'app-child',
     template: `<button (click)="valueChanged()">Click
me</button> `
})
export class AppChildComponent {
    @Output() valueChange = new EventEmitter();
    counter = 0;
    valueChanged() {

        this.counter = this.counter + 1;
        this.valueChange.emit(this.counter);
    }
}
```

Right now, we are performing the following tasks in the **`AppChildComponent`** class:

- Created a variable called counter, which will be passed as the parameter of the emitted event.
- Created an **`EventEmitter valueChange`**, which will be emitted to the parent component on the click event of the button.
- Created a function named **`valueChanged()`**. This function is called on the click event of the button, and inside the function event **`valueChange`** is emitted.
- While emitting **`valueChange`** event, value of counter is passed as parameter.

In the parent component **AppComponent**, the child component **AppChildComponent** can be used as shown in the *code listing 3.10.*

Code Listing 3.10

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-root',
                              template:       `<app-child
(valueChange)='displayCounter($event)'></app-child>`
})
export class AppComponent implements OnInit {
    ngOnInit() {
    }
    displayCounter(count) {
        console.log(count);
    }
}
```

Right now, we are performing the following tasks in the **AppComponent** class:

- Using **<app-child>** in the template.
- In the **<app-child>** element, using event binding to use the **valueChange** event.
- Calling the **displayCounter** function on the **valueChange** event.
- In the **displayCounter** function, printing the value of the counter passed from the **AppChildComponent**.

As you can see, the function of **AppComponent** is called on the click event of the button placed on the **AppChildComponent**. This is can be done with **@Output** and **EventEmitter**. When you run the application and click the button, you can see the value of the counter in the browser console. Each time you click on the button, the counter value is increased by 1 as shown in the *figure 3.7.*

A Real Time Example using @Input and @Output

Let's take a real time example to find how @**Input**, @**Output**, and **EventEmitter** are more useful. Consider that AppComponent is rendering a list of products in tabular form. To create the product table

above, we have a very simple AppComponent class with only one function: to return a list of products.

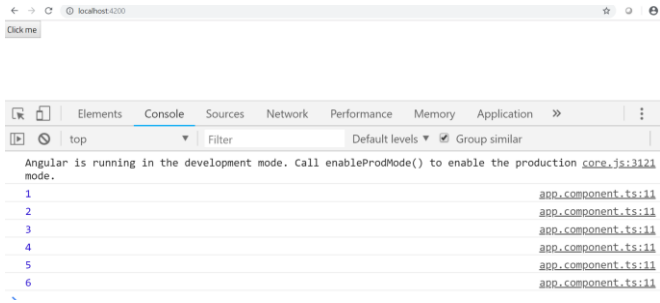


Figure 3.7

Code Listing 3.11

```
export class AppComponent implements OnInit {
  products = [];
  title = 'Products';
  ngOnInit() {
      this.products = this.getProducts();
  }
  getProducts() {
      return [
          { 'id': '1', 'title': 'Screw Driver',
'price': 400, 'stock': 11 },
          { 'id': '2', 'title': 'Nut Volt', 'price':
200, 'stock': 5 },
          { 'id': '3', 'title': 'Resistor', 'price':
78, 'stock': 45 },
          { 'id': '4', 'title': 'Tractor', 'price':
20000, 'stock': 1 },
          { 'id': '5', 'title': 'Roller', 'price': 62,
'stock': 15 },
      ];
  }
}
```

In the **`ngOnInit`** life cycle hook, we are calling the **`getProducts()`** function and assigning the returned data to the products variable so it can

be used on the template. There, we are using the *ngFor directive to iterate through the array and display the products. Template is created as shown in the *code listing 3.12.*

Code Listing 3.12

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-root',
    template: `
    <h1 class="text-center">{{title}}</h1>
    <table>
        <thead>
        <th>Id</th>
        <th>Title</th>
        <th>Price</th>
        <th>Stock</th>
        </thead>
        <tbody>
            <tr *ngFor="let p of products">
                <td>{{p.id}}</td>
                <td>{{p.title}}</td>
                <td>{{p.price}}</td>
                <td>{{p.stock}}</td>
            </tr>
        </tbody>
    </table>
    `
})
export class AppComponent implements OnInit {
  products = [];
  title = 'Products';
  ngOnInit() {
      this.products = this.getProducts();
  }
  getProducts() {
      return [
         { 'id': '1', 'title': 'Screw Driver', 'price':
400, 'stock': 11 },
           { 'id': '2', 'title': 'Nut Volt', 'price':
200, 'stock': 5 },
           { 'id': '3', 'title': 'Resistor', 'price':
```

```
78, 'stock': 45 },
            { 'id': '4', 'title': 'Tractor', 'price':
20000, 'stock': 1 },
          { 'id': '5', 'title': 'Roller', 'price': 62,
'stock': 15 },
      ];
  }
}
```

With this code, products are rendered in a table as shown in the *figure 3.8.*

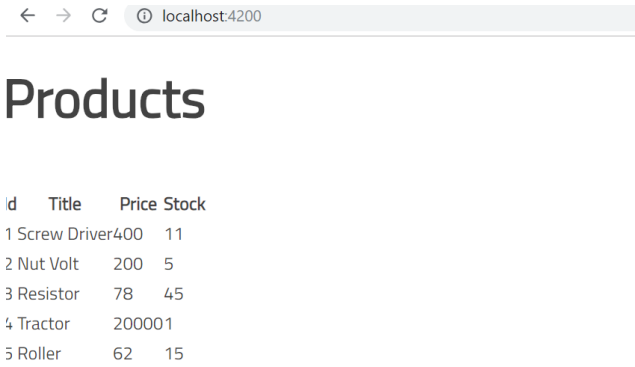


Figure 3.8

Now we have requirements on the above table as follows:

- If the value of stock is more than 10, then the button color should be green.
- If the value of stock is less than 10, then the button color should be red.
- The user can enter a number in the input box, which will be added to that particular stock value.
- The color of the button should be updated on the basis of the changed value of the product stock.

To achieve this task, let us create a new child component called **`StockStatusComponent`**. Essentially, in the template of **`StockStatusComponent`**, there is one button and one numeric input box. In **`StockStatusComponent`**:

- We need to read the value of stock passed from **`AppComponnet`**. For this, we need to use @**`Input`**.
- We need to emit an event so that a function in **`AppComponent`** can be called on the click of the **`StockStatusComponent`**. For this, we need to use **@`Output`** and **`EventEmitter`**.

`StockStatusComponent` is created as shown in the **code listing 3.13**.

Code Listing 3.13

```
import { Component, Input, EventEmitter, Output,
OnChanges } from '@angular/core';
@Component({
    selector: 'app-stock-status',
                template:    `<input    type='number'
[(ngModel)]='updatedstockvalue'/>  <button  class='btn
btn-primary'
     [style.background]='color'
    (click)="stockValueChanged()">Change Stock Value</
button> `
})
export class StockStatusComponent implements OnChanges
{
    @Input() stock: number;
    @Input() productId: number;
    @Output() stockValueChange = new EventEmitter();
    color = '';
    updatedstockvalue: number;
    stockValueChanged() {
      this.stockValueChange.emit({ id: this.productId,
updatdstockvalue: this.updatedstockvalue });
        this.updatedstockvalue = null;
    }
    ngOnChanges() {
        if (this.stock > 10) {
            this.color = 'green';
        } else {
            this.color = 'red';
        }
    }
}
```

Let's explore the above class line by line.

- In the first line we are importing everything required: **@Input**, **@Output** etc.
- In the template, there is one numeric input box which is bound to the **updatedStockValue** property using **[(ngModel)]**. We need to pass this value with an event to the **AppComponent**.
- In the template, there is one button. On the click event of the button, an event is emitted to the **AppComponent**.
- We need to set the color of the button on the basis of the value of product stock. So, we must use property binding to set the background of the button. The value of the color property is updated in the class.
- We are creating two **@Input()** decorated properties - stock and productId - because value of these two properties will be passed from **AppComponent**.
- We are creating an event called **stockValueChange**. This event will be emitted to **AppComponent** on the click of the button.
- In the **stockValueChanged** function, we are emitting the **stockValueChange** event and also passing the product id to be updated and the value to be added in the product stock value.
- We are updating the value of color property in the **ngOnChanges()** life cycle hook because each time the stock value gets updated in the **AppComponent**, the value of the color property should be updated.

Here we are using the **@Input** decorator to read data from **AppComponent** class, which happens to be the parent class in this case. So, to pass data from the parent component class to the child component class, use **@Input** decorator.

In addition, we are using **@Output** with **EventEmitter** to emit an event to **AppComponent**. So to emit an event from the child component class to the parent component class, use **EventEmitter** with **@Output()** decorator.

Therefore, **StockStatusComponent** is using both **@Input** and **@Output** to read data from **AppComponent** and emit an event to **AppComponent**.

Let us first modify the template. In the template, add a new table column. Inside the column, the **<app-stock-status>** component is used.

Code Listing 3.14

```
<h1 class="text-center">{{title}}</h1>
```

```
    <table>
        <thead>
        <th>Id</th>
        <th>Title</th>
        <th>Price</th>
        <th>Stock</th>
        </thead>
        <tbody>
            <tr *ngFor="let p of products">
                <td>{{p.id}}</td>
                <td>{{p.title}}</td>
                <td>{{p.price}}</td>
                <td>{{p.stock}}</td>
              <td><app-stock-status [productId]='p.id'
                    [stock]='p.stock'
        (stockValueChange)='changeStockValue($event)'>
                    </app-stock-status>
                </td>
            </tr>
        </tbody>
    </table>
```

We are passing the value to **`productId`** and stock using property binding (remember, these two properties are decorated with **`@Input()`** in **`StockStatusComponent`**) and using event binding to handle the **`stockValueChange`** event (remember, this event is decorated with **`@Output()`** in **`StockStatusComponent`**).

Next, we need to add **`changeStockValue`** function in the **`AppComponent`**. Add the code as shown in listing 3.15 in the **`AppComponent`** class:

Code Listing 3.15

```
productToUpdate: any;
  changeStockValue(p) {
       this.productToUpdate = this.products.find(this.
findProducts, [p.id]);
    this.productToUpdate.stock = this.productToUpdate.
stock + p.updatdstockvalue;
  }
```

```
findProducts(p) {
  return p.id === this[0];
}
```

In the function, we are using the JavaScript **`Array.prototype.find`** method to find a product with a matched **`productId`** and then updating the stock count of the matched product. When you run the application, you'll get the following output as shown in the *figure 3.9*.

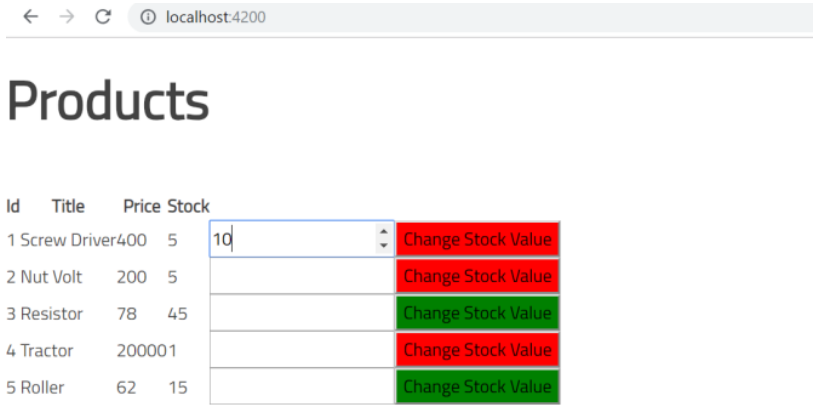


Figure 3.9

When you enter a number in the numeric box and click on the button, you perform a task in the child component that updates the operation value in the parent component. Also, on the basis of the parent component value, the style is being changed in the child component. All this is possible using Angular **`@Input`**, **`@Output`**, and **`EventEmitter`**.

Temp Ref Variable

Temp Ref Variable is used to read properties or call methods of child component in the template of parent component.

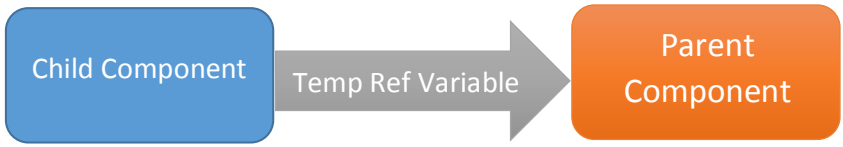


Figure 3.10

It is used to access properties and methods of Child Component inside the template of Parent Component. Let us consider child component as shown in the *code listing 3.16.*

Code Listing 3.16

```
export class AppChildComponent {
    message = 'I am Child';
    sayChildHello() {
        console.log('I am clicked in the child');
    }
}
```

You want to use message property and **`sayChildHello`** function of **`AppChildComponent`** on the template of parent component; you can do that using Tem Ref Variable as shown in the *code listing 3.17.*

Code Listing 3.17

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
     <app-child #childtemp></app-child>
     <h2>{{childtemp.message}}</h2>
      <button (click)='childtemp.sayChildHello()'>call
child function</button>
    `
})
export class AppComponent implements OnInit {
  ngOnInit() {

  }
}
```

To use **Temp Ref** Variable give a name to child component using #. Here we gave name **`childtemp`**. Using this name you can use any properties or methods using dot in the template of parent component.

`ViewChild` and `ContentChild`

ViewChild or Content Child is used to read properties or call methods of child component in the class of parent component.

Figure 3.11

In further chapter, we will cover these topics with content projection in detail.

Summary

Understanding of communication between components is essential. In real time applications, you always send data between components. In this chapter, we learnt about following topics:

- @Input
- @Output
- Temp Ref Variable

——****——

CHAPTER 4

Angular Directives

In this chapter, you learn will about Angular Directives. Following topics will be covered in this chapter:

- What is Directives
- Structural Directives
- Custom Attribute Directives
- @HostListener in Attribute Directives
- @HostBinding in Attribute Directives

What is Directives

Directives creates DOM elements, change their structure, or behavior in an Angular application. There are three types of directives in Angular:

- Components : Directives with template
- Attribute Directives : Change appearance and behavior of an element, component, or another directives
- Structural Directives : Change DOM layout by adding or removing elements

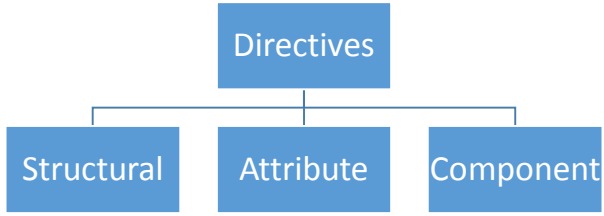


Figure 4.1

The basic difference between a component and a directive is that a component has a template, whereas an attribute or structural directive does not have a template. Angular has provided us many inbuilt structural and attribute directives. Inbuilt structural directives are **`*ngFor`**, **`*ngIf`** and attribute directives as **`NgStyle`** and **`NgModel`**.

Structural Directives

Structural directive changes the structure of DOM elements. For example, **`*ngIf`** is a structure directive which is used to provide an 'if' condition to the statements to be

executed. If the expression evaluates to a False value, the elements are removed from the DOM whereas if it evaluates to True, then element is added to the DOM.

Consider *code listing 4.1*, in this **`*ngIf`** directive will add div in DOM is value of **`showMessage`** property is true.

Code Listing 4.1

```
@Component({
  selector: 'app-message',
  template:`
  <div *ngIf = 'showMessage'>
    Show Message
  </div>
  `
})
export class AppMessageComponent {

        showMessage  = true;
  }
```

Keep in mind that **`*ngIf`** will does not hide or show DOM element. Rather, it adds or removes depending on the condition.

`*ngFor` structure directive creates DOM elements in a loop. Consider *code listing 4.2*, in this **`*ngFor`** directive will add rows in a table depending on number of items in data array.

Code Listing 4.2

```
@Component({
  selector: 'app-message',
  template:`
  <table>
   <tr *ngFor='let f of data'>
   <td>{{f.name}}</td>
   </tr>
  </table>
  `
})
export class AppMessageComponent {

        data = [
```

```
            {name : 'foo'},
            {name: 'koo'}
        ];
  }
```

In most of the cases, you will not have to create custom structural directive and built-in directives should be enough.

Custom Attribute Directive

Attribute Directives change appearance and behavior of an element, component, or another directives. Angular provides many attribute directives such as **`NgStyle`** and **`NgModel`**. We have seen use of **`NgModel`** in previous sections. In this section, let us create a custom Attribute Directive. To do this, we need to create a class and decorate it with **`@directive`** decorators. A simple attribute directive to change the color of an element can be created as shown in the *code listing 4.3*:

Code Listing 4.3

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


@Directive({
    selector: '[appChbgcolor]'
})
export class ChangeBgColorDirective {


constructor(private el: ElementRef, private renderer:
Renderer) {
        this.ChangeBgColor('red');
    }


    ChangeBgColor(color: string) {


      this.renderer.setElementStyle(this.l.nativeEleme
nt,'color', color);
    }
}
```

To create a custom attribute directive, you need to create a class and decorate it with **`@Directive`**. In the constructor of the directive class, inject the services **`ElementRef`** and **`Renderer`**. Instances of these two classes are needed to get the reference of the host element and of the renderer.

While we are creating an attribute directive to change the color of the element, keep in mind that we do not need to create a new attribute directive to just change the color; simple colors can be changed by using property binding. However, the attribute directive we created will change color of an element in this example. There are few important points to remember:

- Import required modules like Directive, ElementRef, and Renderer from Angular core library.
- Create a TypeScript class.
- Decorate the class with @directive.
- Set the value of the selector property in @directive decorator function. The directive would be used, using the selector value on the elements.
- In the constructor of the class, inject ElementRef and Renderer object.

We are injecting **`ElementRef`** in the directive's constructor to access the DOM element. We are also injecting Renderer in the directive's constructor to work with DOM's element style. We are calling the renderer's **`setElementStyle`** function. In the function, we pass the current DOM element by using the object of **`ElementRef`** and setting the color style property of the current element. We can use this attribute directive by its selector in **`AppComponent`** as shown in the *code listing 4.4*.

Code Listing 4.4

```
@Component(
    {
        selector: 'app-container',
        template: `<p appChbgcolor >{{message}}</p>`
    })
export class AppComponent {
```

We used an attribute directive on a paragraph element. It will change the color of the paragraph text to red. In addition to use a directive, we need to import and declare the attribute directive at the **`app.module.ts`**. Right now, the **`appChbgcolor`** directive will change the color of the host element.

`@HostListener()` *in Attribute Directives*

In Angular, the **`@HostListener()`** function decorator allows you to handle events of the host element in the directive class.

Let us take the following requirement: when you hover mouse on the host element, only the color of the host element should change. In addition, when the mouse is gone, the color of the host element should change to its default color. To do this, you need to handle events raised on the host element in the directive class. In Angular, you do this using **`@HostListener()`**.

To understand **`@HostListener()`** in a better way, consider another simple scenario: on the click of the host element, you want to show an alert window. To do this in the directive class, add **`@HostListener()`** and pass the event 'click' to it. Also, associate a function to raise an alert as shown in the *code listing 4.5*:

Code Listing 4.5

```
@HostListener('click') onClick() {
    window.alert('Host Element Clicked');
}
```

In Angular, the **`@HostListener()`** function decorator makes it super easy to handle events raised in the host element inside the directive class. Let us go back to our requirement that says you must change the color to red only when the mouse is hovering, and when it is gone, the color of the host element should change to black. To do this, you need to handle the **`mouseenter`** and **`mouseexit`** events of the host element in the directive class. To achieve this, modify the **`appChbgcolor`** directive class as shown in *code listing 4.6*:

Code Listing 4.6

```
import { Directive, ElementRef, Renderer, HostListener
} from '@angular/core';


@Directive({
    selector: '[appChbgcolor]'
})
export class ChangeBgColorDirective {
```

```
    constructor(private el: ElementRef, private
renderer: Renderer) {
        // this.ChangeBgColor('red');
    }

    @HostListener('mouseover') onMouseOver() {
        this.ChangeBgColor('red');
    }
     @HostListener('click') onClick() {
        window.alert('Host Element Clicked');
    }
    @HostListener('mouseleave') onMouseLeave() {
        this.ChangeBgColor('black');
    }

    ChangeBgColor(color: string) {

          this.renderer.setElementStyle(this.
el.nativeElement, 'color', color);
    }
}
```

In the directive class, we are handling the **mouseenter** and **mouseexit** events. As you see, we are using **@HostListener()** to handle these host element events and assigning a function to it. So, let's use **@HostListener()** function decorator to handle events of the host element in the directive class.

@HostBinding() *in Attribute Directives*

In Angular, the **@HostBinding()** function decorator allows you to set the properties of the host element from the directive class.

Let's say you want to change the style properties such as height, width, color, margin, border etc. or any other internal properties of the host element in the directive class. Here, you'd need to use the **@HostBinding()** decorator function to access these properties on the host element and assign a value to it in directive class.

The **@HostBinding()** decorator takes one parameter, the name of the host element property which value we want to assign in the directive.

In our example, our host element is a HTML **`div`** element. If you want to set border properties of the host element, you can do that using **@`HostBinding()`** decorator as shown in *code listing 4.7*:

Code Listing 4.7

```
@HostBinding('style.border') border: string;

@HostListener('mouseover') onMouseOver() {
    this.border = '5px solid green';
}
```

Using this code, on a mouse hover, the host element border will be set to a green, solid 5-pixel width. Therefore, using **`@HostBinding`** decorator, you can set the properties of the host element in the directive class.

Summary

Directives are essential of an Angular application. You use structural directives to manipulate structure of DOM and use attribute directives to change attribute of DOM Elements. In this chapter, you learnt following topics:

- What is Directives
- Structural Directives
- Custom Attribute Directives
- `@HostListener` in Attribute Directives
- `@HostBinding` in Attribute Directives

——****——

CHAPTER 5

ViewEncapsulation in Angular

In this chapter, you will learn **`ViewEncapsualtion`** in Angular. Following topics will be covered in this chapter:

- Shadow DOM
- None Mode
- Native Mode
- Emulated Mode

Shadow DOM

To understand **`ViewEncapsulation`** in Angular, first we should understand about Shadow DOM. Putting it in simple words; Shadow DOM brings Encapsulation in HTML Elements. Using the Shadow DOM, markup, styles, and behaviors are scoped to the element and do not clash with other nodes of the DOM. Shadow DOM is part of Web Components, which encapsulates styles and login of element.

Angular Components are made up of three things:

- Component class
- Template
- Style

Combination of these three makes an Angular component reusable across application. Theoretically, when you create a component, in some way you create a web component (however, Angular Components are not web components) to take advantage of Shadow DOM. You can also use Angular with browsers, which does not support Shadow DOM because Angular has its own emulation and it can emulate Shadow DOM.

To emulate Shadow DOM and encapsulate styles, Angular provides four types of **`ViewEncapsulation`**. They are as follows:

- Emulated
- None

- ShadowDom
- Native (Deprecated in Angular 6.1)

These four types have different characteristics, as shown in the **figure 5.1**.

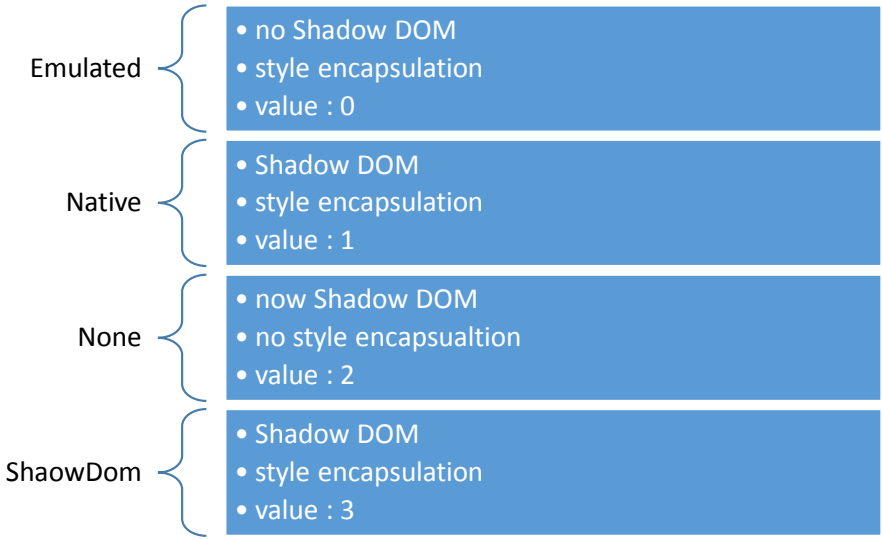


Figure 5.1

Let us try to understand it by using an example.

None Mode

To understand all **`viewEnacapsulation`** modes, let us create a component as shown in the *code listing 5.1*.

Code Listing 5.1

```
import { Component, ViewEncapsulation } from '@angular/
core';
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css'],
    encapsulation: ViewEncapsulation.None
})
export class AppComponent {
```

```
    title = 'parent component';
}
```

Template of **AppComponent** is created as shown in the *code listing 5.2*.

Code Listing 5.2

```
<div>
    <h1>
        Welcome to {{ title }}!
    </h1>
</div>
<app-child></app-child>
```

Since **viewEnacapsualtion** deals with styling, let us create style for **AppComponent**. We have put some style for h1 element in CSS file of **AppComponent** as shown in *code listing 5.3*.

Code Listing 5.3

```
  h1 {
    background: red;
    color: white;
    text-transform: uppercase;
    text-align: center;
}
```

As you noticed **AppChild** component is being used in the **AppComponent**. **AppChild** component is created as shown in the *code listing 5.4*.

Code Listing 5.4

```
import { Component } from '@angular/core';
@Component({
    selector: 'app-child',
    template: `
         <h1>{{title}}</h1>
  `
})
export class AppChildComponent {
    title = 'child app';
}
```

ViewEncapsulation deals with encapsulation of styling and creation of Shadow DOM and to see different options available with

ViewEncapsulation, we are using same element h1 in **AppChild** component also.

In **ViewEncapsulation.None** option,

- There is no shadow DOM
- Style is not scoped to component

As you run the application, you will find h1 style has applied to both components, even though we set style only in **AppComponent**. It happened because in **AppComponent** we have set encapsulation property to **ViewEncapsulation.None**.

Code Listing 5.5

```
import { Component, ViewEncapsulation } from '@angular/
core';
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css'],
    encapsulation: ViewEncapsulation.None
})
export class AppComponent {
    title = 'parent component';
}
```

In the browser when you examine source code, you will find h1 style has been declared in the head section of the DOM as shown in the *figure 5.2*.

Figure 5.2

Therefore, in **`ViewEncapsulation.None`**, style gets moved to the DOM head section and is not scoped to the component. There is no Shadow DOM for the component and component style can affect all nodes of the DOM.

`ShadowDom` Mode

Characteristics of Native and **`ShadowDom`** mode are almost same. However, starting Angular 6.1, Native mode has been deprecated. In **`ViewEncapsulation.ShadowDom`** option:

- Angular will create Shadow DOM for the component
- Style is scoped to component

As you run the application, you will find h1 style has applied to both components, even though we set style only in **`AppComponent`**. It happened because in **`AppComponent`**, we have set encapsulation property to **`ViewEncapsulation.ShadowDom`**, and we are using **`AppChildComponnet`** as child inside template of **`AppComponent`**. You can set encapsulation to **`ShadowDom`** as shown in *code listing 5.6*.

Code Listing 5.6

```
import { Component, ViewEncapsulation } from '@angular/
core';
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css'],
    encapsulation: ViewEncapsulation.ShadowDom
})
export class AppComponent {
    title = 'parent component';
}
```

In the browser, when you examine source code, you will Shadow DOM has created for the **`AppComponent`** and style is scoped to that as shown in *figure 5.3*.

Therefore, in **`ViewEncapsulation.ShadowDom`** Angular creates a Shadow DOM and style is scoped to that Shadow DOM.

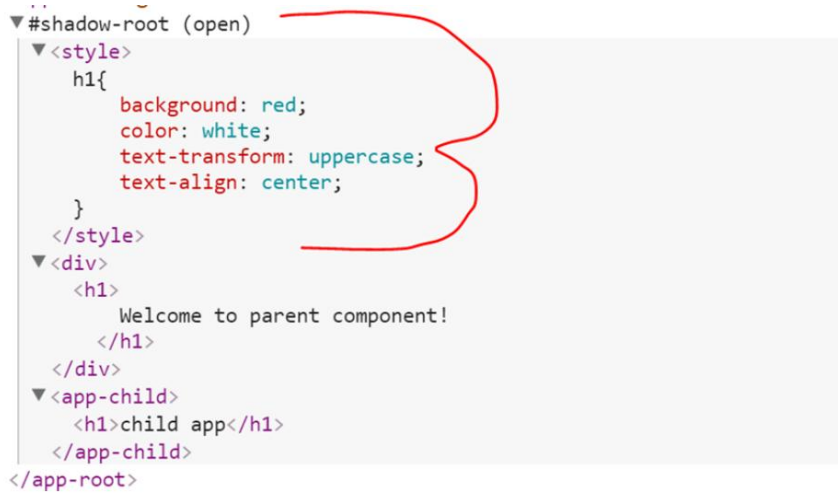


Figure 5.3

Emulated Mode

In Angular default mode is emulated mode. In **`ViewEncapsulation.Emulated`**, in this option:

- Angular will not create Shadow DOM for the component
- Style will be scoped to the component
- This is default value for encapsulation

You can enable emulated mode as shown in *code listing 5.7*.

Code Listing 5.7

```
import { Component, ViewEncapsulation } from '@angular/
core';
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css'],
    encapsulation: ViewEncapsulation.Emulated
})
export class AppComponent {
    title = 'parent component';
}
```

As you run the application, you will find that h1 style from **`AppComponent`** is not applied to h1 of **`AppChildComponent`**. It is due to emulated scoping. In this, style is scoped only to the component. In this option, Angular only emulates to Shadow DOM and does not create a real shadow DOM. Hence, the application that runs in browsers does not support Shadow DOM also and styles are scoped to the component as well.

Let us see how Angular achieves this? In the browser, when you examine source code, you will find answer, consider *figure 5.4*.

Figure 5.4

Angular has created style in the head section of the DOM and given an arbitrary id to the component. On basis of ID, selector style is scoped to the component.

Summary

It is very common misconception that Angular always creates Shadow DOM for components, however after reading this chapter, you know that it depends on **`ViewEncapsulation`** mode. In this chapter, you learnt about:

- Shadow DOM
- None Mode
- Native Mode
- Emulated Mode

———****———

CHAPTER 6

Pipes

In this chapter, we will learn about Pipes in Angular. Following topics will be covered:

- Pipes
- Built-in pipes
- Custom pipes
- Types of pipes

Pipes

Angular pipes take data as input and transform it to your desired output. Angular pipes are functions, which takes input and transform it to the desired output. Keep in mind that, it does not change the underlying data structure of input parameter. You can visualize, Angular pipes as shown in *figure 6.1*.

Figure 6.1

You may use pipes in various scenarios such as:

- Displaying data in lowercase or uppercase
- Displaying currency in a particular format such as USD, INR
- Filtering input data array to display desired rows, and so on.

Built-in Pipes

Angular provides many built-in pipes for various transformations such as:

- UpperCasePipe

- LowerCasePipe
- CurrencyPipe
- PercentPipe
- DatePipe

Let us see how you can use built-in pipes. For example, using interpolation you are displaying name of the product, however, you want to transform the product name output in uppercase. You can do this using Angular pipe uppercase as shown *code listing 6.1*.

Code Listing 6.1

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

@Component({
    selector: 'app-root',
    template: `{{productName | uppercase}}`
})
export class AppComponent {
    productName = 'Cricket Bat';
}
```

As an output, **`productName`** will be displayed in uppercase. However, keep in mind that the underlying data is not changed. Essentially pipes take input and transform it to the desired output. So, pipe works as shown in *figure 6.2*.

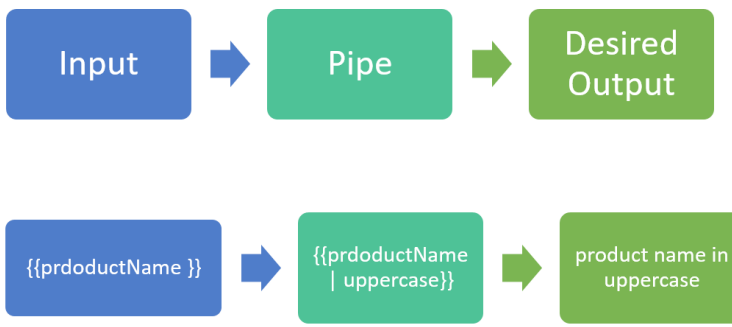


Figure 6.2

Let us see how we could use the built-in currency pipe. Currency pipe can be used as shown in the *code listing 6.2*.

Code Listing 6.2

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-root',
  template: `{{productName | uppercase}} = {{productPrice
| currency}}`
})
export class AppComponent {
    productName = 'Cricket Bat';
    productPrice = 990;
}
```

You can also pass parameters to a pipe using a colon. You can pass input to currency pipe as shown in *code listing 6.3*.

Code Listing 6.3

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

@Component({
    selector: 'app-root',
  template: `{{productName | uppercase}} = {{productPrice
| currency:'CAD':'symbol-narrow':'4.2-2'}}`
})
export class AppComponent {
    productName = 'Cricket Bat';
    productPrice = 990;
}
```

Custom Pipes

Even though Angular provides many default pipes, there could be requirements where you would need custom pipes. Creating a custom pipe is as simple as creating a function. Let us say that we want to create a pipe, which will capitalize first letter of each words in a string.

Consider component as shown *code listing 6.4*.

Code Listing 6.4

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-root',
```

```
    template: `
  <ul *ngFor='let n of names'>
   <li>{{n.name}}</li>
  </ul>
  `
})
export class AppComponent {
    names = [];
    constructor() {
        this.names = this.getNames();
    }
    getNames() {
        return [
            { 'name': 'dhananjay Kumar' },
            { 'name': 'jason beres' },
            { 'name': 'adam jafe' }
        ];
    }
}
```

As output, you will get names printed. Now we have requirement that in output, capitalize first character of each names. For that, we must create a custom pipe.

To create a custom pipe, you need to follow these steps:

1. Create a class.
2. Implement **`PipeTransform`** in the class.
3. Implement transform function.

Therefore, you can create a pipe to capitalize first character as shown in the *code listing 6.5*.

Code Listing 6.5

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({ name: 'firstcharcateruppercase' })
export class FirstCharacterUpperCase implements
PipeTransform {
    transform(value: string, args: string[]): any {
        if (!value) {
            return value;
        }
```

```
        return value.replace(/\w\S*/g, function (str) {
             return str.charAt(0).toUpperCase() + str.
substr(1).toLowerCase();
         });
    }
}
```

As you see, custom pipes are nothing but functions which take input parameters, and return some value. You need to write all logic of the pipe inside transform method. To use **`firstcharacteruppercase`** pipe, first you need to declare it in the module, after that you can use that in the component as shown in *code listing 6.6.*

Code Listing 6.6

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-root',
    template: `
  <ul *ngFor='let n of names'>
   <li>{{n.name | firstcharcateruppercase}}</li>
  </ul>
  `
})
export class AppComponent {
    names = [];
    constructor() {
        this.names = this.getNames();
    }
    getNames() {
        return [
            { 'name': 'dhananjay Kumar' },
            { 'name': 'jason beres' },
            { 'name': 'adam jafe' }
        ];
    }
}
```

Now you will get in output, the first character of each name in the uppercase. You can create custom pipe for other purposes also like filtering data table etc.

To summarize:

- Custom pipes are class, which is decorated with @Pipe
- Name property of @Pipe decorator defines name of the pipe
- Pipe class should implement PipeTransform interface
- Implement pipe business logic inside transform method

Types of Pipe

There are two types of pipes:

- Stateless pipes
- Stateful pipes

What we used and created above are stateless pipes. They are pure functions, which take an input and return transformed values.

Stateful pipes are complex to implement and they remember state of the data they transform. Usually they create an HTTP request, store the response, and display the output. Angular inbuilt async pipe is example of a stateful pipe.

Summary

In this chapter, we learned about pipes in Angular. Pipes transform an input data to the desired output. Angular provides many built-in pipes; however, there could be requirements to write custom pipes. There are two types of pipes: stateless pipes and stateful pipes. We covered following topics:

- Pipes
- Built-in pipes
- Custom pipes
- Types of pipes

——————****——————

CHAPTER 7

Template Driven Forms

In this chapter, you will learn about Forms in Angular. Following topics will be covered in this chapter:

- Template-Driven Forms
- ngModel, [ngModel],[(ngModel)]
- Binding form to ngForm
- Submitting the form
- Handling validation
- Resetting the form

Template- Driven Forms

You create forms to accept user inputs. A form is created for various purposes, such as:

- To login a user
- To sign up a user
- To submit a help request
- To place an order
- To schedule a meeting

A form in combination of other HTML controls, a form may contains,

1. Input elements
2. Check boxes
3. Radio buttons
4. Buttons

To work with user inputs, Angular provides us different types of forms. They are as follows:

- Template Driven Form
- Reactive Form

Purpose of both these forms is same, with few basic differences.

In Template-driven forms, you create all validation logics, form controls in the template. To create a template-driven form, you almost do not need any code in the template class. Template-driven forms are normal forms on which you use Angular directives to enable Angular features such as two-way data binding, change notification, validations, and so on.

Let us create a Login Form as template-driven form. First step required is to import **`FormsModule`** in the **`AppModule`** as shown in the *code listing 7.1*. I am importing **`FormsModule`** in **`AppModule`** and on purpose showing you the entire code of **`AppModule`**.

Code Listing 7.1

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

import { AppComponent } from './app.component';

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

In the **`AppModule`**, we have imported **`FormModule`** and passed it in the imports array.

We are going to create a template-driven form for Login operation. To model login operation, let us create a model class, which will hold information about login operation. The login model class is a plain class with no behavior (methods) and contains only properties.

Code Listing 7.2

```
export class Login {
```

```
    constructor(
      public email: string,
      public password: string,
      public rememberpassword?: boolean
    ) {  }
}
```

Login class is anemic model that has only properties but not the behavior. There are two required properties and one optional property in the **Login** class.

Next, let us create a vanilla HTML form to accept user input for login functionality. You can create a form as shown in the *code listing 7.3*. I have used bootstrap classes to make form little bit more immersive.

Code Listing 7.3

```
<div class="container">
  <h1>Login Form</h1>
  <form>
    <div class="form-group">
      <label for="email">Email</label>
    <input type="text" class="form-control" id="email"
required>
    </div>
    <div class="form-group">
      <label for="password">Password</label>
          <input type="password" class="form-control"
id="password" required>
    </div>
    <div class="form-check">
       <input type="checkbox" class="form-check-input"
id="rpassword">
                    <label   class="form-check-label"
for="rpassword">remember password</label>
    </div>
   <button type="submit" class="btn btn-success">Login</
button>
  </form>
</div>
```

You will get a HTML form rendered as shown in the *figure 7.1*.

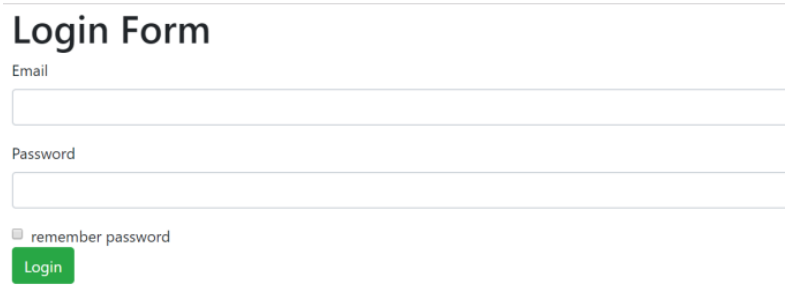


Figure 7.1

We have set up the form, next we need to convert above HTML form to Angular template-driven form. For that, very first create model object in the component class.

We already have a **`Login`** model class; now let us create an object of **`Login`** class, which we will bind to the form. The object of Login class will act as model of the form. You can create Login model object instance as shown in *code listing 7.4*.

Code Listing 7.4

```
model: Login;
   constructor() {
    this.model = new Login('a@abc.com', 'password123',
true );
   }
```

We have model object in place, now bind it to the input elements of the form to enable them as Angular template-driven form elements. We can bind email property of model object to email input at the form as shown in the *code listing 7.5*.

Code Listing 7.5

```
  <input type="text"
              [(ngModel)]='email'
              name='email'
              class="form-control" required>
```

There are two important properties set:

1. name
2. ngModel

To work with template-driven form, you must set name property of input element. The name attribute is used as the key in **`FormGroup`** array for a particular element.

`ngModel, (ngModel),((ngModel))`

ngModel is used to bind input element with model object. There are three options to bind input element with model object.

`ngModel`

You can use **`ngModel`** as shown in *code listing 7.6*. In this case, **`ngModel`** will use name attribute to create key in **`ngForm`** object. In **`ngModel`**, input element is not set with any initial value.

Code Listing 7.6

```
<input  type="text"
         ngModel
         name='email'
         class="form-control" required>
```

`[ngModel]`

You can use **`[ngModel]`** to create one-way binding. When **`ngModel`** is set as property binding, it will set initial value of input element with model object.

Code Listing 7.7

```
<input  type="text"
         [ngModel]='model.email'
         name='email'
         class="form-control" required>
```

In the form, initial value of email input element will be set to model object's email property.

`[(ngModel)]`

You can use **`[(ngModel)]`** to create two-way data binding in between model object and input element. This option will set initial value of element and whenever element is updated, it will update model object also.

Code Listing 7.8

```
<input  type="text"
           [(ngModel)]='model.email'
           name='email'
           class="form-control" required>
```

I would recommend using **ngModel** as it will set initial value of input element and will not update model object.

Binding Form to ngForm

Next, we have to bind the HTML form with **ngForm** directive such that it would work as Angular template-driven form.

Angular **ngForm** directive enables form elements with additional features. It mainly performs following tasks on form elements, which has **ngModel** and name attribute set:

- It monitors properties of form elements
- It monitors validity of form elements
- It raises notification if value changes of form elements

ngForm also has **valid** property which is set to **true**, on if all the form elements validity is true.

You can bind form to **ngForm** directive using template reference variable, as shown in *code listing 7.9.*

Code Listing 7.9

```
<form novalidate #loginform='ngForm'>
</form>
```

By using **ngForm** directive, **ngModel** property binding, and name attribute a HTML Form is converted to Angular template-driven form.

Putting everything together, template-driven login form should look like *code listing 7.10*.

Code Listing 7.10

```
<div class="container">
  <h1>Login Form</h1>
  <form novalidate #loginform='ngForm'>
    <div class="form-group">
      <label for="email">Email</label>
         <input  type="text"  [(ngModel)]='model.email'
```

```
name='email' class="form-control" required>
    </div>
    <div class="form-group">
      <label for="password">Password</label>
          <input  type="password"  [(ngModel)]='model.
password' class="form-control" name="password" required
minlength="4">
    </div>
    <div class="form-check">
       <input type="checkbox" class="form-check-input"
[(ngModel)]='model.rememberpassword' name="rpassword">
                    <label    class="form-check-label"
for="rpassword">remember password</label>
    </div>
   <button type="submit" class="btn btn-success">Login</
button>
  </form>
</div>
```

We have used **`ngForm`**, **`ngModel`** on DOM as shown in the *figure 7.2*.

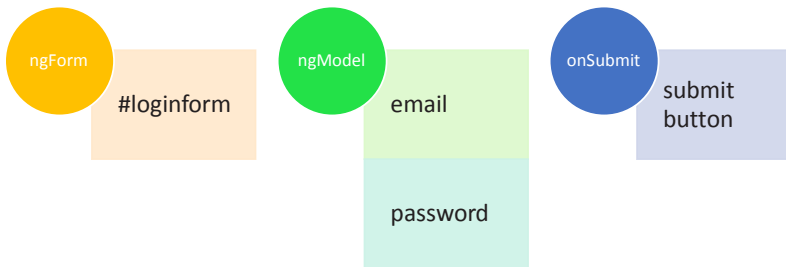


Figure 7.2

Submitting the Form

To submit template driven form, you need to use **`ngSubmit`** event on the form. You can bind **`ngSubmit`** event to a function in the component class. To submit form modify form as shown in *code listing 7.11*.

Code Listing 7.11

```
<form          novalidate          #loginform='ngForm'
(ngSubmit)='onSubmit(loginform)'>
```

```
  <!-- other elements  -->
  </form>
```

On the component class **`onSubmit`** function will look like *code listing 7.12*.

Code Listing 7.12

```
onSubmit(loginform) {
    console.log(loginform.value);
    console.log(loginform.status);
   }
```

Template-driven form has value and status properties. The value is an object, which contains all form elements as properties and status is Boolean, which will be true if form is valid.

You can also disable **`Submit`** button, if form is invalid by setting [disabled] property of button to **`form.invalid`**.

Putting everything together **`template-driven`** Login form will be as shown in *code listing 7.13*.

Code Listing 7.13

```
<form          novalidate          #loginform='ngForm'
(ngSubmit)='onSubmit(loginform)'>
    <div class="form-group">
      <label for="email">Email</label>
         <input  type="text"  [(ngModel)]='model.email'
name='email' class="form-control" required>
    </div>
    <div class="form-group">
      <label for="password">Password</label>
          <input  type="password"  [(ngModel)]='model.
password' class="form-control" name="password" required
minlength="4">
    </div>
    <div class="form-check">
       <input type="checkbox" class="form-check-input"
[(ngModel)]='model.rememberpassword' name="rpassword">
                    <label    class="form-check-label"
for="rpassword">remember password</label>
    </div>
```

```
   <button [disabled]='loginform.invalid' type="submit"
class="btn btn-success">Login</button>
  </form>
```

Component class with Login model and **`onSubmit`** function will be as shown in *code listing 7.14*.

Code Listing 7.14

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
   model: Login;
   constructor() {
    this.model = new Login('a@abc.com', 'password123',
true );
   }
   onSubmit(loginform) {
    console.log(loginform.value);
    console.log(loginform.status);
   }
}
```

Handling Validation in Template Driven Form

You need to validate form controls, and show error messages on validation. Angular template-driven form tracks state of a control and updates CSS class associated with it. When you use **`ngModel`** in the form, it performs following tasks:

- Two- way data binding
- Find whether user touched the control
- Track state of the controls
- Update validation CSS associated with the controls

Angular associates various CSS classes to controls basis on the state as shown in *figure 7.3*.

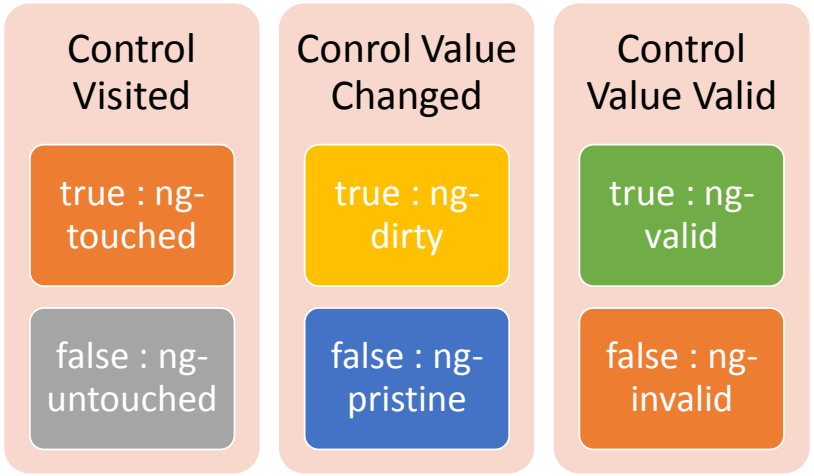


Figure 7.3

To show error messages using these classes, let us add some custom CSS to component. You can add CSS of *code listing 7.15* either in global CSS file or component's style.

Code Listing 7.15

```
.ng-valid[required], .ng-valid.required  {
    border-left: 5px solid #42A948; /* green */
  }

  .ng-invalid:not(form)  {
    border-left: 5px solid #a94442; /* red */
  }
```

Using combination of Angular template-driven form validations and CSS classes, you can show the validation error message on form as shown in the *code listing 7.16*.

Code Listing 7.16

```
<div class="form-group">
      <label for="email">Email</label>
         <input type="text" [(ngModel)]='model.email'
#email="ngModel"   name='email'   class="form-control"
required>
```

```
    </div>
       <div  [hidden]="email.valid  ||  email.pristine"
class="alert alert-danger">
      Email is required
    </div>
```

We are assigning **`ngModel`** to temp reference variable to track the changes. Also, using Bootstrap classes to make error message **`div`** more immersive, error message **`div`** displays if form control is invalid and touched as shown in the *figure 7.4.*

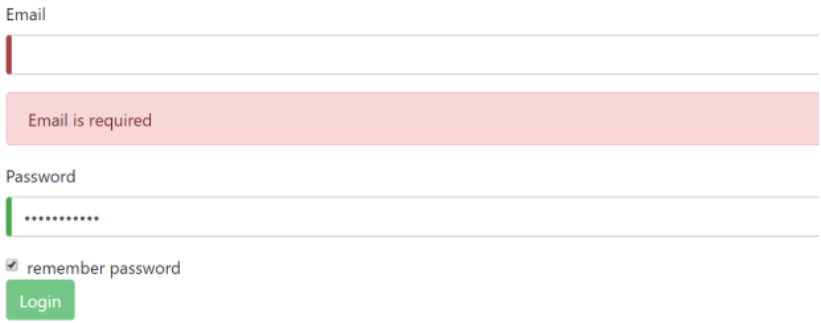


Figure 7.4

Reset the Form

You may have the requirement to reset the form and bind form with new model object. Angular template-driven form provides API for this as well. You can use **`reset()`** method **`ngForm`** to reset the form as shown in *code listing 7.17.*

Code Listing 7.17

```
<button          (click)='newLogin();loginform.reset()'
class="btn btn-warning">Reset</button>
```

On the click event of the button, you are calling two functions:

- newLogin function
- reset method of the form

Before resetting the form, in **`newLogin`** function, you create new model object as shown the *code listing 7.18*. I have put null for all three properties; you can choose other values depending on your requirement.

Code Listing 7.18

```
newLogin() {
     this.model = new Login(null, null, null);
   }
```

Putting Everything Together

Putting everything together, you can create **`template-driven`** login form in Angular with validations, model and track changes as shown in *code listing 7.19*.

Code Listing 7.19

```
          <form     novalidate     #loginform='ngForm'
(ngSubmit)='onSubmit(loginform)'>
    <div class="form-group">
      <label for="email">Email</label>
         <input  type="text"  [(ngModel)]='model.email'
#email="ngModel"   name='email'   class="form-control"
required>
    </div>
       <div  [hidden]="email.valid  ||  email.pristine"
class="alert alert-danger">
      Email is required
    </div>
    <div class="form-group">
      <label for="password">Password</label>
          <input  type="password"  #password="ngModel"
[(ngModel)]='model.password'      class="form-control"
name="password"
        required minlength="4">
    </div>
   <div [hidden]="password.valid || password.pristine"
class="alert alert-danger">
      Password is invalid
    </div>
    <div class="form-check">
       <input type="checkbox" class="form-check-input"
[(ngModel)]='model.rememberpassword' name="rpassword">
                    <label    class="form-check-label"
for="rpassword">remember password</label>
```

```
    </div>
   <button [disabled]='loginform.invalid' type="submit"
class="btn btn-success">Login</button>
       <button  (click)='newLogin();loginform.reset()'
class="btn btn-warning">Reset</button>
  </form>
```

In the component class, you can have model object and function to handle submit as shown in *code listing 7.20.*

Code Listing 7.20

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
   model: Login;
   constructor() {
    this.model = new Login('a@abc.com', 'password123',
true );
   }
   onSubmit(loginform) {
    console.log(loginform.value);
    console.log(loginform.status);
   }

   newLogin() {
     this.model = new Login(null, null, null);
   }
}
```

Summary

In this chapter you learnt about template-driven form, validations etc. You create forms to accept user input. In this chapter you learnt about following topics:

- Template-Driven Forms
- `ngModel`, [`ngModel`],[(`ngModel`)]
- Binding form to ngForm
- Submitting the form
- Handling validation
- Resetting the form

——****——

CHAPTER 8

Reactive Forms

In this chapter, you will learn about Angular Reactive Forms. Following topics will be covered in this chapter:

- Creating Reactive Forms
- Adding Validations
- Using FormBuilder
- Custom Validators
- Passing parameters to Custom Validators
- setValue and patchValue
- Conditional Validation

Creating Reactive Form

Reactive forms work on model-driven approach. Validation logic and initial state of controls are defined by model object. Each change in the form state returns a new state of the model. Every control of reactive forms emits an observable, which gives status and value of the form controls. Since validation logic is part of the component class, writing tests for reactive forms is easier.

To start working with reactive forms, first add **ReactiveFormsModule** in the App Module as shown in *code listing 8.1*.

Code Listing 8.1

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

@NgModule({
  declarations: [
```

```
    AppComponent
  ],
  imports: [
    BrowserModule, ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Once module is imported, you need to import following classes in the component:

- FormGroup
- FormControl
- FormArray

The **`FormControl`** class corresponds to one individual form control, tracking its value and validity. While creating your reactive form, you will create an object of the **`FormControl`** class to add a control in the form. The **`FormControl`** constructor takes three parameters:

- Initial data value, which can be null.
- Array of synchronous validators. This is an optional parameter.
- Array of asynchronous validators. This is an optional parameter.

In the component class, you can create a **`FormControl`** as shown in *code listing 8.2*.

Code Listing 8.2

```
export class AppComponent {
    email = new FormControl('');
}
```

We are not passing any optional parameters like sync validations or async validations, but we will explore these parameters while adding validation to a **`FormControl`**.

On the View, you can use email **`FormControl`** as shown in *code listing 8.3*.

Code Listing 8.3

```
<input [formControl]='email'
       type="text"
```

```
        placeholder="Enter Email" />
{{email.value | json}}
```

As you see, we are using property binding to bind the **formControl** email to the input element on the view. In a form, there will be more than one controls, to work with multiple controls you need **ForGroup** class. **FormGroup** is a group of **FormControls**. You can encapsulate various **FormControls** inside a **FormGroup**, which offers an API for:

- Tracking the validation of group of controls or form
- Tracking the value of group of controls or form

It contains child controls as its property and it corresponds to the top lever form on the view. You can think of a **FormGroup** as a single object, which aggregates the values of child **FormControl**. Each individual form control is the property of the **FormGroup** object.

You can create an object of **FormGroup** class as shown in *code listing 8.4*.

Code Listing 8.4

```
loginForm = new FormGroup({
        email: new FormControl(' '),
        password: new FormControl(' ')
    });
```

Here we have created a login form, which is a **FormGroup**. It consists of two form controls for email and password. It is very easy to use a **FormGroup** on the template as shown in *code listing 8.5*.

Code Listing 8.5

```
<form [formGroup]='loginForm' novalidate class="form">
  <input formControlName='email'
         type="text"
         class="form-control"
         placeholder="Enter Email" />
  <input formControlName='password'
         type="password"
         class="form-control"
         placeholder="Enter Password" />
</form>
{{loginForm.value | json}}
{{loginForm.status | json }}
```

Here we're using property binding to bind your **FormGroup** with the form

and **formControlName** directive to attach **FormControl** to a particular element on the template.

From last chapter, you have used a template driven form, you will notice that the HTML code on template is much leaner now: there is no **ngModel** or name attached with elements. You can find value and status of the form by using value and status property. Now, you no longer need to use template reference variable to find status and value of the form.

To submit the form, let us add a submit button on the form and a function to be called. We will modify the form as shown in *code listing 8.6.*

Code Listing 8.6

```
<form (ngSubmit)='loginUser()' [formGroup]='loginForm'
novalidate class="form">
 <input formControlName='email' type="text" class="form-
control" placeholder="Enter Email" />
    <input  formControlName='password'  type="password"
class="form-control" placeholder="Enter Password" />
  <button class="btn btn-default">Login</button>
</form>
```

In the component class, you can add a function to submit the form as shown in *code listing 8.7.*

Code Listing 8.7

```
export class AppComponent implements OnInit {
    loginForm: FormGroup;
    ngOnInit() {
        this.loginForm = new FormGroup({
             email: new FormControl(null, Validators.
required),
            password: new FormControl()
        });
    }
    loginUser() {
        console.log(this.loginForm.status);
        console.log(this.loginForm.value);
    }
}
```

Here we have just added a function called **`loginUser`** to handle the form submit event. Inside this function, you can read the value and status of **`FormGroup`** object **`loginForm`** using the status and value properties. As you can see, this gives you an object which aggregates the values of individual form controls.

Adding Validation

To add validation to **`FormControls`**, first import Validators from **@ angular/forms**, then you can use Validators while creating controls as shown in the *code listing 8.8.*

Code Listing 8.8

```
ngOnInit() {
        this.loginForm = new FormGroup({
             email: new FormControl(null, Validators.
required),
            password: new FormControl()
        });
    }
```

On the template, you can use the **`FormGroup`** get method to find an error in a particular form control and use it. In the *code listing 8.9*, we are checking the validation error for an email and displaying the error div.

Code Listing 8.9

```
   <div class="alert  alert-danger" *ngIf="loginForm.
get('email').hasError('required')    &&    loginForm.
get('email').touched">
    Email is required
  </div>
```

You can also disable your submit button by default, and enable it when the form is valid to allow submission. This can be done as shown in *code listing 8.10*:

Code Listing 8.10

```
    <button  [disabled]='loginForm.invalid'  class="btn
btn-default">Login</button>
```

Putting everything together, the template with reactive forms should look like *code listing 8.11*.

Code Listing 8.11

```
<form (ngSubmit)='loginUser()' [formGroup]='loginForm'
novalidate class="form">
          <input  formControlName='email'  type="text"
class="form-control" placeholder="Enter Email" />
     <div class="alert  alert-danger" *ngIf="loginForm.
get('email').hasError('required')     &&     loginForm.
get('email').touched">
          Email is required
      </div>
     <input formControlName='password' type="password"
class="form-control" placeholder="Enter Password" />
    <div class="alert  alert-danger" *ngIf=" !loginForm.
get('password').valid    &&    loginForm.get('email').
touched">
Password is required and should less than 10 characters
      </div>
     <button [disabled]='loginForm.invalid' class="btn
btn-default">Login</button>
  </form>
```

In addition, the component class will be as shown in *code listing 8.12.*

Code Listing 8.12

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl, FormArray, Validators
} from '@angular/forms';
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
    loginForm: FormGroup;
    ngOnInit() {

        this.loginForm = new FormGroup({
            email: new FormControl(null, [Validators.
required, Validators.minLength(4)]),
          password: new FormControl(null, [Validators.
```

```
required, Validators.maxLength(8)])
        });
    }
    loginUser() {
        console.log(this.loginForm.status);
        console.log(this.loginForm.value);
    }
}
```

Using FormBuilder

FormBuilder is used to simplify the syntax for **FormGroup** and **FormControl**. This is very useful when your form gets lengthy. Let us refactor **loginForm** to use **FormBuilder**. To do so, first import **FormBuilder** from **@angular/forms** then inject it to the component as shown in *code listing 8.13.*

Code Listing 8.13

```
constructor(private fb: FormBuilder) {

    }
```

You can use **FormBuilder** to create a reactive form as shown in the following listing. As you see, it has simplified the syntax as shown in *code listing 8.14.*

Code Listing 8.14

```
  this.loginForm = this.fb.group({
                  email: [null, [Validators.required,
Validators.minLength(4)]],
               password: [null, [Validators.required,
Validators.maxLength(8)]]
        });
```

The template will be the same for both **FormBuilder** and **FormControl** classes. Putting everything together, Reactive form with the **FormBuilder** will look like, as shown in *code listing 8.15*.

Code Listing 8.15

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl, FormArray, Validators,
```

```
FormBuilder } from '@angular/forms';
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
    loginForm: FormGroup;
    constructor(private fb: FormBuilder) {
    }
    ngOnInit() {

        this.loginForm = this.fb.group({
                  email: [null, [Validators.required,
Validators.minLength(4)]],
               password: [null, [Validators.required,
Validators.maxLength(8)]]
        });
    }
    loginUser() {
        console.log(this.loginForm.status);
        console.log(this.loginForm.value);
    }
}
```

Custom Validators

Angular provides us many useful validators, including required, **`minLength`**, **`maxLength`**, and pattern. These validators are part of the Validators class, which comes with the **`@angular/forms`** package. Let us assume you want to add a required validation to the email control and a **`maxLength`** validation to the password control. You do that as shown in the *code listing 8.16*.

Code Listing 8.16

```
    this.loginForm = new FormGroup({
            email: new FormControl(null, [Validators.
required]),
          password: new FormControl(null, [Validators.
```

```
required, Validators.maxLength(8)]),
            age: new FormControl(null)
        });
```

On the template, you can use validators to show or hide an error message as shown the *code listing 8.17*. Essentially, you are reading the **`formControl`** using the **`get()`** method and checking whether it has an error or not using the **`hasError()`** method. You are also checking whether the **`formControl`** is touched or not using the touched property.

Code Listing 8.17

```
<input formControlName='email' type="text" class="form-
control" placeholder="Enter Email" />
     <div class="alert  alert-danger" *ngIf="loginForm.
get('email').hasError('required')     &&     loginForm.
get('email').touched">
          Email is required
      </div>
```

Let us say you want the age range to be from 18 to 45. Angular does not provide us range validation; therefore, we will have to write a custom validator for this.

In Angular, creating a custom validator is as simple as creating another function. The only thing you need to keep in mind is that it takes one input parameter of type **`AbstractControl`** and it returns an object of key-value pair if the validation fails. Let us create a custom validator called **`ageRangeValidator`**, where the user should be able to enter an age only if it is in the given range.

A custom validator should look like as shown in *figure 8.1*.

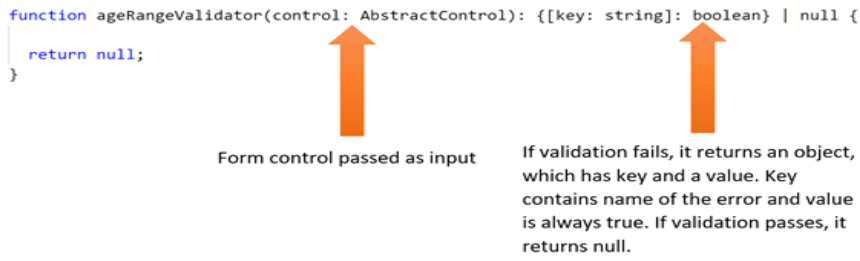


Figure 8.1

The type of the first parameter is **`AbstractControl`** because it is a base class of **`FormControl`**, **`FormArray`**, and **`FormGroup`**, and it allows you to read the value of the control passed to the custom validator function. The custom validator returns either of the following:

- If the validation fails, it returns an object, which contains a key value pair. Key is the name of the error and the value is always Boolean true.
- If the validation does not fail, it returns null.

Now, we can implement the **`ageRangeValidator`** custom validator as shown in *code listing 8.18*.

Code Listing 8.18

```
function ageRangeValidator(control: AbstractControl):
{ [key: string]: boolean } | null {
    if (control.value !== undefined && (isNaN(control.
value) || control.value < 18 || control.value > 45)) {
        return { 'ageRange': true };
    }
    return null;
}
```

Here, we are hardcoding the maximum and minimum range in the validator. In the next section, we will see how to pass these parameters. Now, you can use **`ageRangeValidator`** with the age control as shown in *code listing 8.19*. As you see, you need to add the name of the custom validator function in the array:

Code Listing 8.19

```
this.loginForm = new FormGroup({
            email: new FormControl(null, [Validators.
required]),
          password: new FormControl(null, [Validators.
required, Validators.maxLength(8)]),
        age: new FormControl(null, [ageRangeValidator])
        });
```

On the template, the custom validator can be used like any other validator. We are using the **`ageRange`** validation to show or hide the error message. Refer *code listing 8.20*.

Code Listing 8.20

```
<input formControlName='age' type="number" class="form-
control" placeholder="Enter Age" />
    <div class="alert  alert-danger" *ngIf="loginForm.
get('age').dirty  &&  loginForm.get('age').errors  &&
loginForm.get('age').errors.ageRange ">
      Age should be in between 18 to 45 years
    </div>
```

If the user does not enter an age between 18 to 45 then the reactive form will show an error as shown in *figure 8.2*.

Figure 8.2

Now, age control is working with the custom validator. The only problem with **`ageRangeValidator`** is that hardcoded age range that only validates numbers between 18 and 45. To avoid a fixed range, we need to pass the maximum and minimum age to **`ageRangeValidator`**.

Passing Parameters to a Custom Validator

An Angular custom validator does not directly take extra input parameters aside from the reference of the control. To pass extra parameters, you need to add a custom validator inside a factory function. The factory function will then return a custom validator. You heard it right; in JavaScript, a function can return another function. Essentially, to pass parameters to a custom validator, you need to follow these steps:

- Create a factory function and pass parameters that will be passed to the custom validator to this function.
- The return type of the factory function should be ValidatorFn which is part of @angular/forms.
- Return the custom validator from the factory function.

The factory function syntax will be as shown in *figure 8.3*.

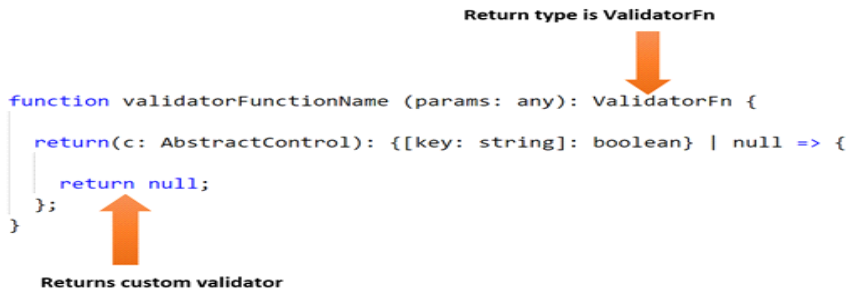


Figure 8.3

Now you can refactor the **`ageRangeValidator`** to accept input parameters as shown in *code listing 8.21*.

Code Listing 8.21

```
function ageRangeValidator(min: number, max: number):
ValidatorFn {
    return (control: AbstractControl): { [key: string]:
boolean } | null => {
       if (control.value !== undefined && (isNaN(control.
value) || control.value < min || control.value > max))
{
            return { 'ageRange': true };
        }
        return null;
    };
}
```

We are using the input parameters max and min to validate age control. Now, you can use **`ageRangeValidator`** with age control and pass the values for max and min as shown in *code listing 8.22*.

Code Listing 8.22

```
    min = 10;
    max = 20;
    ngOnInit() {
        this.loginForm = new FormGroup({
             email: new FormControl(null, [Validators.
required]),
```

```
          password: new FormControl(null, [Validators.
required, Validators.maxLength(8)]),
     age: new FormControl(null, [ageRangeValidator(this.
min, this.max)])
        });
    }
```

On the template, the custom validator can be used like any other validator. We are using **`ageRange`** validation to show or hide an error message as shown in *code listing 8.23.*

Code Listing 8.23

```
 <input formControlName='age' type="number" class="form-
control" placeholder="Enter Age" />
    <div class="alert  alert-danger" *ngIf="loginForm.
get('age').dirty  &&  loginForm.get('age').errors  &&
loginForm.get('age').errors.ageRange ">
      Age should be in between {{min}} to {{max}} years
    </div>
```

In this case, if the user does not enter an age between 10 and 20, the error message will be displayed.

setValue and patchValue

`setValue` and **`patchValue`** methods are used to set controls' values. These methods exist on both **`formArray`** and **`formControl`**.

Purpose of both **`setValue`** and **`patchValue`** methods is to set control's values with one major difference, **setValue** sets values of all controls inside a **`FormGroup`**, whereas **`patchValue`** can set values of a specific control.

setValue

FormGroup's class **`setValue`** method sets values of all controls inside **`FormGroup`**. If you want to set control value of **`loginForm`** created in previous section, you can do that as shown in *code listing 8.24.*

Code Listing 8.24

```
   this.loginForm.setValue({email: 'debugmode@outlook.
com', password: 'abc', age : '30'});
```

As you see that, we are updating value of all controls. If you try to partially update control values, Angular will throw error. Consider *code listing 8.25*.

Code Listing 8.25

```
this.loginForm.setValue({password: 'abc', age : '30'});
```

We are not setting value for email control; hence, **`setValue`** method will throw exception as shown in *image 8.4*.

Figure 8.4

`patchValue`

The **`patchValue`** allows you to set value of a particular control in form group. Using **`patchValue`**, you can opt out some controls and can update value of controls you desire. You can update password and age control of **`loginForm`** as shown in *code listing 8.26*. Keep in mind that we are not updating value of email control and still Angular is not complaining about that.

Code Listing 8.26

```
    this.loginForm.patchValue({password: 'abc', age :
'30'});
```

Both **`patchValue`** and **`setValue`** method has two more nullable parameters:

- onlySelf
- emitEvent

For **`setValue`**, when **`onlySelf`** is set to true, each change only affects this control and not its parent. Default value is set to **false**.

For **`setValue`**, when **`emitEvent`** is true or not supplied, both **`statusChanges`** and **`valueChanges`** observables emit events with latest status and value for updated control. When false, no events are emitted.

For **`patchValue`**, when **`onlySelf`** is set to true, each change only affects this control and not its parent. Default value is set to **`true`**.

For **`patchValue`**, when **`emitEvent`** is true or not supplied, both **`statusChanges`** and **`valueChanges`** observables emit events with latest

status and value for updated control. When false, no events are emitted.

So other major difference you keep in mind about **`FormGroup`** **`setValue`** and **`patchValue`** methods is that by default **`setValue`** update all parent controls whereas **`patchValue`** only updates itself.

Note: **`FormControl`** class also has **`setValue`** and **`patchValue`** methods. Their behavior is little different from **`FormGroup`** class methods.

Conditional Validation

To understand conditional validation, let us modify login form created in previous section as shown in *code listing 8.27*.

Code Listing 8.27

```
this.loginForm = this.fb.group({
            email: [null, Validators.required],
               password: [null, [Validators.required,
Validators.maxLength(8)]],
            phonenumber: [null],
            notification: ['email']
        });
```

On the template, we will add radio button group to handle Send Notification option. Consider *code listing 8.28*.

Code Listing 8.28

```
<form (ngSubmit)='loginUser()' [formGroup]='loginForm'
novalidate class="form">
         <input  formControlName='email'  type="text"
class="form-control" placeholder="Enter Email" />
    <div class="alert  alert-danger" *ngIf="loginForm.
get('email').hasError('required')    &&    loginForm.
get('email').touched">
      Email is required
    </div>
     <input formControlName='password' type="password"
class="form-control" placeholder="Enter Password" />
     <input formControlName='phonenumber' type="text"
class="form-control" placeholder="Enter Phone Number"
/>
    <div class="alert  alert-danger" *ngIf="loginForm.
```

```
get('phonenumber').hasError('required') && loginForm.
get('phonenumber').touched">
      Phone Number is required
    </div>
    <br />
      <label class='control-label'>Send Notification</
label>
    <br />
    <label class="radio-inline">
                <input type="radio" value="email"
formControlName="notification">Email
    </label>
    <label class="radio-inline">
                <input type="radio" value="phone"
formControlName="notification">Phone
    </label>
    <br />
     <button [disabled]='loginForm.invalid' class="btn
btn-default">Login</button>
  </form>
```

In Reactive forms both **`FormControls`** and **`FormGroups`** have a **`valueChanges`** method. It returns an observable type, so you can subscribe to it, to work with real-time value changing of **`FormControls`** or **`FormGroups`**. In our example, we need to subscribe to **`valueChanges`** of notification **`FormControl`** as shown in *code listing 8.29*.

Code Listing 8.29

```
  formControlValueChanged() {
       this.loginForm.get('notification').valueChanges.
subscribe(
            (mode: string) => {
                console.log(mode);
            });
    }
```

You need to call above function on **`ngOnInit`** life cycle hook. Now when you change the selection for notification on the form in the browser console you can see, you have the most recent value. Keep in mind that, we are not handling any event on the radio button to get the latest value. Angular has a **`valueChanges`** method which returns recent value as observable on the

`FormControl` and **`FormGroup`**, and we are subscribed to that for recent value on notification **`FormControl`**.

Our requirement is that when the notification is set to phone, then **`phonenumber FormControl`** should be a required field and if it is set to email, then **`phonenumber FormControl`** should not have any validation.

Let us modify **`formControlValueChnaged()`** function as shown in *code listing 8.30* to enable conditional validation on **`phonenumber FormControl`**.

Code Listing 8.30

```
formControlValueChanged() {
                const phoneControl = this.loginForm.
get('phonenumber');
       this.loginForm.get('notification').valueChanges.
subscribe(
            (mode: string) => {
                console.log(mode);
                if (mode === 'phone') {
               phoneControl.setValidators([Validators.
required]);
                } else if (mode === 'email') {
                    phoneControl.clearValidators();
                }
                phoneControl.updateValueAndValidity();
            });

    }
```

There are a lot of codes above, so let us talk through line by line.

- Using get method of FormBuilder getting an instance of phone number FormControl
- Subscribing to the valueChanges method on notification FormControl
- Checking the current value of notification FormControl
- If the current value is phone, using setValidators method of FormControl to set required validator on phonenumber control
- If the current value is email, using clearValidators method of FormControl to clear all validation on phonenumber control
- In last calling updateValueAndValidity method to update validation rules of phonecontrol

Run the application and you will see that as you change notification value, validation of **`phonenumber`** is getting changed. By using the power of Angular Reactive Form's **`valueChanges`** method, you can achieve conditional validations and many other functionalities such as reacting to changes in the underlying data model of the reactive form.

Summary

In this chapter, we learnt about Reactive Forms in Angular. You use reactive forms to keep all validation logic and model in the component class. In this chapter you learnt about following topics:

- Create Reactive Forms
- Adding Validations
- Using FormBuilder
- Custom Validators
- Passing parameters to Custom Validators
- setValue and patchValue
- Conditional Validation

——****——

CHAPTER 9

Angular Routing

In this chapter, you will learn about different ways of routing and navigation using strategies, query parameters. The topics that will be covered under this chapter are as follows:

- Create Route
- RouterModule.forChild()
- Routing Strategies
- Dynamic Route Parameters
- Navigate using Code
- Query Parameter
- Child Route
- Route Guards

In Angular Routing, you load a component or more than one component dynamically in Router Outlet. You can have more than one Router Outlet and according to Route configuration, components will be loaded dynamically. You can configure application level route and child routes for feature modules. Angular supports Auxiliary Routes using named Router Outlet. Auxiliary Routes means more than one components can be loaded on DOM on the same URL. In Angular application, when you change URL, a particular component will be loaded depending on the route configuration. Angular Routing also supports either hash based or HTML 5 based URL strategy. In a nutshell, you use routing to load component dynamically from the component tree.

Create Route

Starting Angular 7, while creating a new project using Angular CLI, it will ask whether you want to add Angular Routing or not, as shown in the *figure 9.1*.

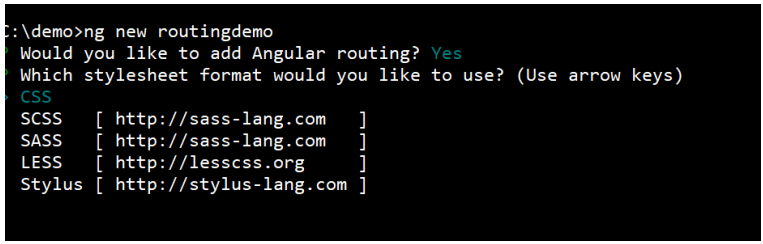


Figure 9.1

If you are working on earlier version of Angular, you can add routing using:

- CLI command
- Manually by creating a routing module

Angular 7 adds a routing module in your project as shown in the *code listing 9.1*:

Code Listing 9.1

app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

This is application level routing module, which is imported in **AppModule** as shown in the *code listing 9.2*:

Code Listing 9.2:

app.module.ts

```
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
```

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

To work with routing, let us first add few components to the application. Using Angular CLI, add following components to the application.

- ng g component login
- ng g component home
- ng g component pagenotfound
- ng g c welcome

Now you should have a project structure as shown in the *figure 9.2*:

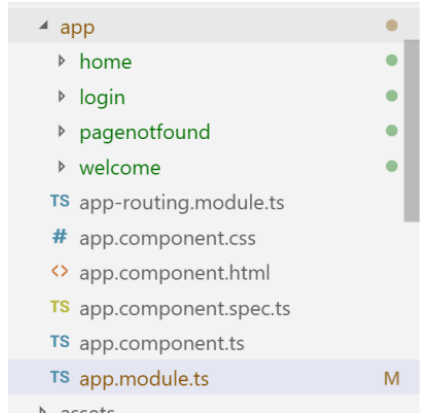


Figure 9.2

In Angular, we navigate from one component to another. You can create a very basic route for **`LoginComponent`**, **`HomeComponent`** as by modifying routes in **`app-routing.module.ts`** as shown in the *code listing 9.3*:

Code Listing 9.3

```
const routes: Routes = [
  {path: 'home', component: HomeComponent},
  {path: '', redirectTo: 'home', pathMatch: 'full'},
  {path: 'login', component: LoginComponent},
  {path: '**', component: PagenotfoundComponent}
];
```

Let us walk through the code:

1. The path property defines part of the URL. Therefore, whenever user enters **baseurl/path**, cross ponding component will be loaded.
2. The component property defines component to load for that particular path.
3. The **pathMatch** property set to **full** means that the whole URL path needs to be matched.
4. The **pathMatch** property set to **prefix** means first route where the path matches the start of the URL is chosen, but then the route matching algorithm is continuing to search for matching child routes where the rest of the URL matches.
5. The **pathMatch** property set to ** means that if nothing matches, go here.

Besides, above properties, there are other properties also which we will discuss in subsequent section. Next, we need to find where to load routes. For that, you need to use:

<router-outlet></router-outlet>

This should be used on the root component. We can create a top-level menu to navigate between the components or rather load components directly as shown in the *code listing 9.4*:

Code Listing 9.4

```
<div>
  <nav class="navbar navbar-default">
      <div class="container-fluid">
          <a class="navbar-brand">{{pageTitle}}</a>
          <ul class="nav navbar-nav">
```

```
                <li>
                    <a [routerLink]="['/home']">Home</a>
                </li>
                <li>
                  <a>Show Messages</a>
              </li>
              <li>
                 <a [routerLink]="['/login']">Log In</a>
              </li>
            </ul>
            <ul class="nav navbar-nav navbar-right">

            </ul>
        </div>
    </nav>
    <div class="container">
      <router-outlet></router-outlet>
    </div>
</div>
```

As you see to navigate, we are using **`[routerLink]`** property binding and binding it to the route name from the routing configuration. See *figure 9.3 & 9.4.*

URL : **baseurl/home**
AppComponent

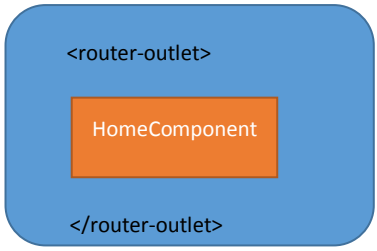


Figure 9.3

URL : **baseurl/Login**
AppComponent

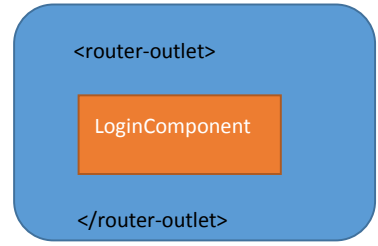


Figure 9.4

So far if you change URL to **baseurl/home** then dynamically **`HomeComponent`** will be loaded in the **`<router-outlet>`** and if you change it to **`baseurl/login`** then dynamically **`LoginComponent`** will be loaded.

RouterModule.forChild()

Let us start adding another feature module called Product. What we are going to do is to configure its own route for the **ProductModule**. To configure routes for feature modules, you need to follow same approach, however instead of **RouterModule.forRoot()** use **RouterModule.forChild().** Add code in feature module **ProductModule** as shown in the *code listing 9.5*:

Code Listing 9.5

Product.module.ts (Excerpts)

```
const productRoutes: Routes = [
  {path: 'products', component: ProductsComponent},
  {path: 'addproduct', component: AddproductComponent},
          {path:     'editproduct/:id',     component:
EditproductComponent},
        {path:    'productdetails/:id',    component:
ProductdetailsComponent},
];
@NgModule({
  declarations: [
                ProductsComponent,
                 AddproductComponent,
                 EditproductComponent,
                 ProductdetailsComponent],
  imports: [
    CommonModule,
    RouterModule.forChild(productRoutes)
  ]
})
export class ProductModule { }
```

Let us talk through code, we have configured routing array, and then using **RouterModule.forChild()** to create route. We are importing **ProductModule** in **AppModule** as shown in the *code listing 9.6*:

Code Listing 9.6

app.module.ts (Excerpts)

```
@NgModule({
```

```
  declarations: [
    AppComponent,
    PagenotfoundComponent,
    HomeComponent,
    WelcomeComponent,
    LoginComponent
  ],
  imports: [
    BrowserModule,
    ProductModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Essentially **`RouterModule`** has two methods to create routes:

1. **forRoot()** method should be used only once, as it creates instance of Router Service. You need only one instance per application to work with this.
2. **`forRoot()`** is used to configure application level route.
3. **`forRoot()`** is used to declare the router directives.
4. For feature module, you should use **`forChild()`** method.
5. **`forChild()`** method does not register Router Service.
6. Both **`forRoot()`** and **`forChild()`** methods accepts array of routes.

In our example above, we are using **`forRoot()`** method with **`AppModule`** because that is application level module. For **`ProductModule`** which is a feature module, we are using **`forChild()`** method.

Also, one important thing you need to keep in mind that always load routes configured using **`forChild()`** before routes configured using **`forRoot()`**.

At this point of time on running application, you should have routing and navigation enabled as shown in the *figures 9.5, 9.6 & 9.7*:

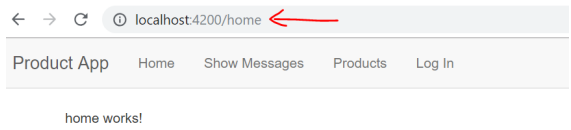


Figure 9.5

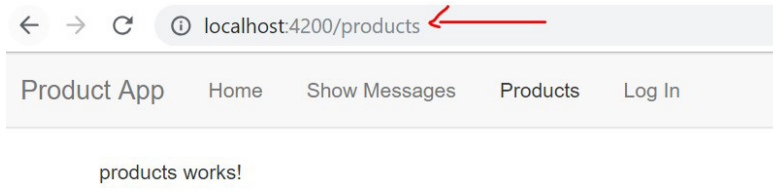


Figure 9.6

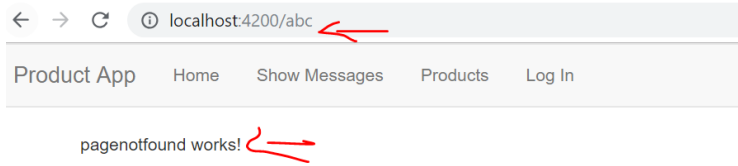


Figure 9.7

As we change URL, cross ponding component is loading dynamically inside **`<router-outlet>`** which is placed on the **`AppComponent`**. At this point of time, template of **`AppComponent`** should look like *code listing 9.7*:

Code Listing 9.7

`app.component.html`

```
<div>
    <nav class="navbar navbar-default">
        <div class="container-fluid">
            <a class="navbar-brand">{{pageTitle}}</a>
            <ul class="nav navbar-nav">
                <li>
                  <a [routerLink]="['/home']">Home</a>
                </li>
                <li>
                    <a>Show Messages</a>
                </li>
                <li>
             <a [routerLink]="['/products']">Products</
a>
```

```
                </li>
                <li>
                      <a [routerLink]="['/login']">Log
In</a>
                </li>
            </ul>
            <ul class="nav navbar-nav navbar-right">

            </ul>
        </div>
    </nav>
    <div class="container">
        <router-outlet></router-outlet>
    </div>
</div>
```

Routing Strategies

Angular Routing allows us to choose either of URL styling:

- HTML 5 style URL or PathLocationStrategy
- Hash-based URL or HashLocationStrategy

To enable hash-based URL or **`HashLocationStrategy`**, you need to configure app routing to use hash that can be done as shown in the *code listing 9.8*:

Code Listing 9.8

`app-routing.module.ts`

```
@NgModule({
   imports: [RouterModule.forRoot(routes, { useHash:
true})],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Now when run the application, you will find hash-based URL as shown in the *figure 9.8*:

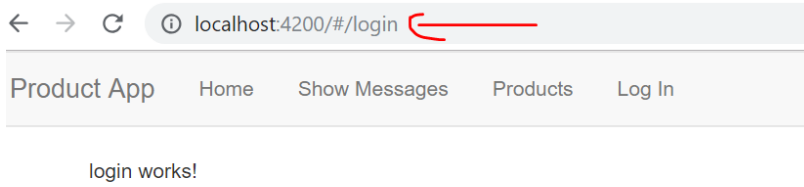


Figure 9.8

Main advantages of using that it works in old browsers also. The # part of the URL is called hash fragment. Biggest advantage of hash # is that anything after hash never gets sent to server. So, for example, if you have URL

Basurl:/Product/#/Home

On the server, only **baseurl:/Product** will be sent and browser will ignore anything after #. This is very useful to maintain some data at the client side and for client side navigation.

Since hash fragment is never sent to the server, client side state can be saved in hash. It is ideal for Single Page Application with baseurl always same for the server. Another advantage is hash fragment can be changed using JavaScript at the client side.

The default Angular routing strategy is `PathLocationStrategy`. It takes advtange of HTML 5 pushstate api to maintain the state of the URL. To work with this, you have to set the base URL as shown in the next listing:

<base href="/">

Since, it is default strategy you don't have to configure anything to enable it. By using this URL can be changed without requesting the server and without using the hash fragment. This is very useful for Single Page Application with one major challenge that if you hard hit a URL:

Basurl:/Product/home

Server should be able to return code for whole URL instead of only root URL. For this you got to write cross ponding code at the server. If you are doing development using Angular CLI that takes care of that.

Dynamic Route Parameters

We may have requirement to create route dynamically. To create route dynamically, you need to pass route parameters. You can pass route parameters using single colon in route configuration. We did that in Product Route Configuration as shown in the *code listing 9.9*:

Code Listing 9.9:

```
const productRoutes: Routes = [
  {path: 'products', component: ProductsComponent},
  {path: 'addproduct', component: AddproductComponent},
          {path:     'editproduct/:id',     component:
EditproductComponent},
        {path:    'productdetails/:id',    component:
ProductdetailsComponent},
];
```

As you see that when you navigate to **basurl/edit/1**, **EditProductComponent** will be loaded with dynamic data id passed to the component. See *figure 9.9*.

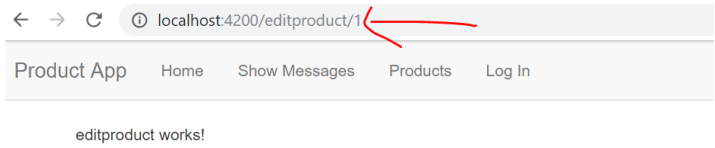


Figure 9.9

You can read passed data in the component. To read that you need to use **`ActivatedRoute`** service. First, import it and inject it in the component class. You need to import **`ActivatedRoute`** service from **`@angular/router`** and then inject it as shown in the *code listing 9.10*.

Code Listing 9.10

```
export class EditproductComponent implements OnInit {

  productidtoedit: any;
  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
     this.route.params.subscribe(
```

```
        (p) => {
              this.productidtoedit = p.id;
        });
  }

}
```

We are subscribing to params, which is a property of **`ActivatedRoute`** service. Since it is an observable, any change in route parameter will be notified.

There is one more way to read the route parameters. Instead of observable approach, you can use snapshot approach. See *code listing 9.11*:

Code Listing 9.11

```
  ngOnInit() {

        this.productidtoedit = this.route.snapshot.
params['id'];
  }
```

Snapshot code is easier to implement and used to read parameter only once. Whereas observable code is complex but it watches for the parameter changes.

Navigate Using Code

To navigate using code we need to use imperative API that router provides. You may need to navigate using the code in various scenarios such as Master-Details etc. Let us say that on **`ProductsComponent`** is rendered as shown in the *figure 9.10*:

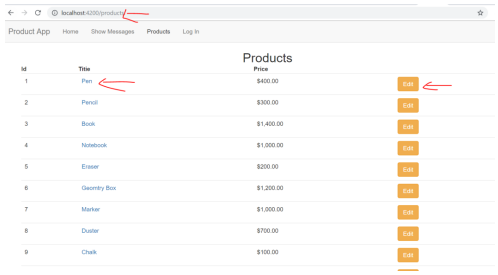


Figure 9.10

On clicking of Product Title, **`ProductdetailsComponent`** Component should be loaded and on click of **Edit** button, **`EditproductComponent`** should be loaded in the **`<router-outlet>`**. In addition, we are passing route parameters.

In template, we can use **`[routerLink]`** to navigate as shown in the *code listing 9.12*:

Code Listing 9.12

```
<a    [routerLink]="['/productdetails',    p.Id]">{{p.
Title}}</a>
```

However, for **Edit** button, we need to write code in component class. For your reference template of **`ProductsComponent`** is as shown in the *code listing 9.13*:

Code Listing 9.13

`Products.component.html`

```
<div class="row">
  <h2 class="text-center">Products</h2>
</div>
<div class="row">
  <table class="table">
     <thead>
       <th>Id</th>
       <th>Titie</th>
       <th>Price</th>
     </thead>
     <tbody>
       <tr *ngFor="let p of products">
         <td>{{p.Id}} </td>
              <td><a  [routerLink]="['/productdetails',
p.Id]">{{p.Title}}</a></td>
         <td>{{p.Price | currency }}</td>
                 <td><button  class="btn  btn-warning"
(click)='editProduct(p.Id)'>Edit</button></td>
       </tr>
     </tbody>
  </table>
```

```
</div>
```

Now to navigate on the **Edit** button to **`EditproductComponent`**, we need to follow the following steps:

1. Import Router from @angular/router
2. Inject it in constructor of component class
3. Use navigate() method to navigate using code

Therefore, you can navigate to **`editproduct`** route as shown in the *code listing 9.14*:

Code Listing 9.14

```
  constructor(private router: Router) { }

  editProduct(id) {
    this.router.navigate(['editproduct', id]);
  }
```

We are using router navigate method, in which passing route name and query parameter. For your reference whole source code for **`ProductsComponent`** is shown in the *code listing 9.15*:

Code Listing 9.15

`Products.component.ts`

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

@Component({
  selector: 'app-products',
  templateUrl: './products.component.html',
  styleUrls: ['./products.component.css']
})
export class ProductsComponent implements OnInit {

  products: any;

  constructor(private router: Router) { }

  editProduct(id) {
```

```
  this.router.navigate(['editproduct', id]);
}

ngOnInit() {

  // in real app data will be fetched from API
  this.products = this.getProducts();
}
getProducts() {
  return [
    {
      Id: '1', Title: 'Pen', Price: '400'

    },
    {
      Id: '2', Title: 'Pencil', Price: '300'

    },
    {
      Id: '3', Title: 'Book', Price: '1400'

    },
    {
      Id: '4', Title: 'Notebook', Price: '1000'

    },
    {
      Id: '5', Title: 'Eraser', Price: '200'

    },
    {
      Id: '6', Title: 'Geomtry Box', Price: '1200'

    },
    {
      Id: '7', Title: 'Marker', Price: '1000'

    },
    {
```

```
        Id: '8', Title: 'Duster', Price: '700'

      },
      {
        Id: '9', Title: 'Chalk', Price: '100'

      },
      {
        Id: '10', Title: 'Stapler', Price: '1300'

      }
    ];
  }
}
```

On the **`EditproductComponent`**, you can read route parameter as shown in the *code listing 9.16*:

Code Listing 9.16

`editproduct.component.ts`

```
export class EditproductComponent implements OnInit {

  productidtoedit: any;
  constructor(private route: ActivatedRoute) { }
  ngOnInit() {
     this.route.params.subscribe(
       (p) => {
             this.productidtoedit = p.id;
       });
  }

}
```

On the template, you can implement back button as shown in the *code listing 9.17*:

Code Listing 9.17

`editproduct.component.html`

```
<h2>
```

```
  Product to Edit : {{productidtoedit}}
</h2>
<button class="btn btn-default" [routerLink]="['/
products']">Back</button>
```

Query Parameter

Route Parameters are required to navigate to a route. Query parameters allow you to pass optional parameters to a route such as pagination information.

You can pass query parameter as in the *code listing 9.18*:

Code Listing 9.18

```
<a [routerLink]="['/productdetails', p.Id]"
[queryParams]="{page:1}">{{p.Title}}</a>
```

You can read query parameter as shown in *code listing 9.19*:

Code Listing 9.19

```
page: any;
  constructor(
    private route: ActivatedRoute) {}

  ngOnInit() {
    this.route.queryParams.subscribe(
      p => {
        this.page = +p['page'] || 0;
      });
  }
```

Query Parameter should be used to pass optional parameters.

Child Route

Angular allows us to create child route. Every Angular route can support a child route inside it. Let us consider that for **`editproduct`** route, we need two child routes.

- Edit by sales representative
- Edit by manager of the product

We can achieve that using **Child Routes** inside editproduct route. We can create a child route as shown in the *code listing 9.20*:

Code Listing 9.20

```
{path: 'editproduct/:id',
   component: EditproductComponent,
   children: [
     { path: '', redirectTo: 'bysalesrep', pathMatch:
'full' },
                { path: 'bymanager', component:
EditproductbymanagerComponent },
               { path: 'bysalesrep', component:
EditproductbyesalesrepComponent },
  ],
  }
```

We have added child routes to **`editproduct`** route using the children property. After adding child routes, whole routing configuration for product feature module will look like as shown in *code listing 9.21*:

Code Listing 9.21:

`product.module.ts`

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ProductsComponent } from './products/products.
component';
import { AddproductComponent } from './addproduct/
addproduct.component';
import { EditproductComponent } from './editproduct/
editproduct.component';
import { RouterModule, Routes } from '@angular/router';
import { ProductdetailsComponent } from './
productdetails/productdetails.component';
import { EditproductbyesalesrepComponent }
from './editproduct/editproductbyesalesrep/
editproductbyesalesrep.component';
```

```
import { EditproductbymanagerComponent } from './
editproduct/editproductbymanager/editproductbymanager.
component';

const productRoutes: Routes = [
  {path: 'products', component: ProductsComponent},
  {path: 'addproduct', component: AddproductComponent},
  {path: 'editproduct/:id',
   component: EditproductComponent,
   children: [
     { path: '', redirectTo: 'bysalesrep', pathMatch:
'full' },
               { path: 'bymanager', component:
EditproductbymanagerComponent },
              { path: 'bysalesrep', component:
EditproductbyesalesrepComponent },
  ],
  },
        {path: 'productdetails/:id', component:
ProductdetailsComponent},
];

@NgModule({
  declarations: [
                ProductsComponent,
                 AddproductComponent,
                 EditproductComponent,
                 ProductdetailsComponent,
                 EditproductbyesalesrepComponent,
                 EditproductbymanagerComponent],
  imports: [
    CommonModule,
    RouterModule.forChild(productRoutes)
  ]
})
export class ProductModule { }
```

Next, we need to put a **<router-outlet>** on **`editproduct.component.html`**, in which child route will be dynamically loaded. See *Code Listing 9.22*:

Code Listing 9.22

```
  <div class="row">
   <h2>Product to Edit : {{productidtoedit}} </h2>
  </div>

  <nav>
    <a [routerLink]="['bysalesrep']">SalesRep</a>
    <a [routerLink]="['bymanager']">Manager</a>
  </nav>

  <router-outlet></router-outlet>

  <br/>
  <br/>
  <button class="btn btn-default" [routerLink]="['/
products']">Back</button>
  <br/>
```

Here, we are creating navigation and then loading child routes in the **<router-outlet>**. You can read route parameter passed from parent route in child route component as shown in *code listing 9.23*:

Code Listing 9.23

```
export class EditproductbymanagerComponent implements
OnInit {

  productid: any;
  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    //   this.route.parent.data.subscribe(data => {
    //     this.productid = data['id'];
    // });
    this.route.parent.params.subscribe(d => {
      this.productid = d['id'];
    });
  }
}
```

We are using parent method of **`ActivateRoute`** to fetch parameter of parent route. On running application, you will find child route for edit product route enabled as shown in the *figures 9.11 & 9.12*:

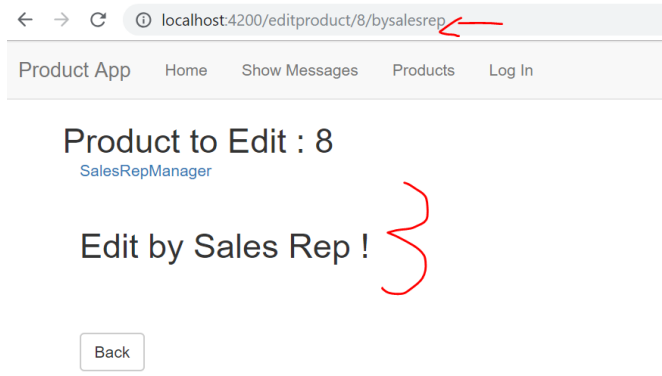


Figure 9.11

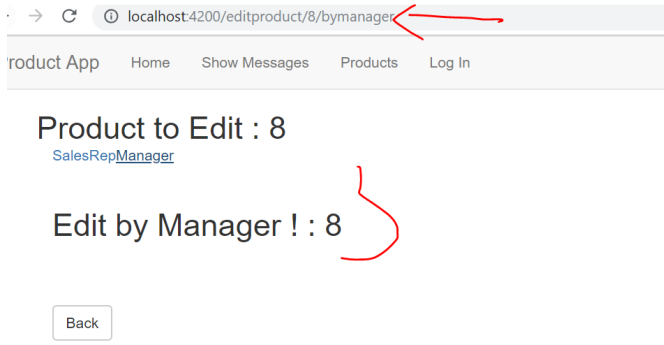


Figure 9.12

As you see in these images, child route is getting activated for **`editproduct`** route.

Auxiliary Route

Angular provides us to have more than one **`<router-outlet>`** in application. You can have named **`<router-outlet>`** and configure routes to load component in a specific named **`<router-outlet>`**. Secondary route is also known as Secondary

Route. To create an Auxiliary route, add **<router-outlet>** as shown in the *code listing 9.24*:

Code Listing 9.24

app.component.html

```
            <div class="col-md-9">
                 <router-outlet></router-outlet>
             </div>
             <div class="col-md-3">
                   <router-outlet name="showmessage"></
router-outlet>
             </div>
```

Now while configuring route, you can pass outlet value to determine in which router outlet, component would be loaded. You can do that as shown in the *code listing 9.25*:

Code Listing 9.25

app-routing.module.ts

```
//other codes
const routes: Routes = [
  {path: 'home', component: HomeComponent},
  {path: '', redirectTo: 'home', pathMatch: 'full'},
  {path: 'login', component: LoginComponent},
   {path: 'showmessage', component: WelcomeComponent,
outlet: 'showmessage' },
  {path: '**', component: PagenotfoundComponent}
];

@NgModule({
   imports: [RouterModule.forRoot(routes, { useHash:
false})],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

You can activate or navigate to auxiliary route as shown in the *code listing 9.26*:

Code Listing 9.26

```
<a [routerLink]="[{outlets:{showmessage:
['showmessage']}}]"> Show Messages</a>
```

While activating, you can pass route name and outlet name. Now you can see in *figure 9.13* that on click of Show Messages, URL is changing and in auxiliary route, Welcome component is loaded.

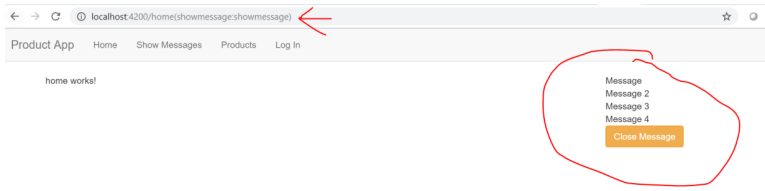


Figure 9.13

You can close an auxiliary route by navigating to **`showmessage`** route in the code. Let us say you have a button on template of **`WelcomeComponent`** and on click of that button, you wish to close secondary route that can be done as shown in *code listing 9.27*:

Code Listing 9.27

`welcome.component.ts`

```
constructor(private router: Router) { }
  closemessage() {
    // To naviagte
     // this.router.navigate([{outlets: {showmessage:
['showmessage']}}]);
    // To close the router
        this.router.navigate([{outlets:  {showmessage:
null}}]);
  }
```

You can have URL changing for auxiliary routes as shown in the *figures 9.14 & 9.15*:

URL : **`baseurl/home`**

`home(showmessage:showmessage)`

`AppComponent`

URL :

`AppComponent`

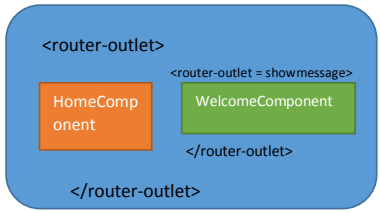


Figure 9.14

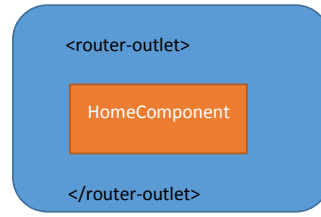


Figure 9.15

You should use auxiliary routes when you have to load more than one component dynamically. The **`<router-outlet>`** without name is default. Besides that, you can have any number of named **`<router-outlet>`** as auxiliary or secondary routes. This is useful in master-detail scenario.

Route Guards

There are four types of route guards available in the Angular routing. They are as follows:

1. CanActivate
2. CanActivateChild
3. CanLoad
4. CanDeactivate

Each route guards have different purposes. For example, **`CanActivate`** route guard controls whether a particular route will be activated or not and **`CanActivateChild`** controls whether child routes of a particular route will be activated or not. Summary of purposes of all four-route guards is shown in the *figure 9.16*.

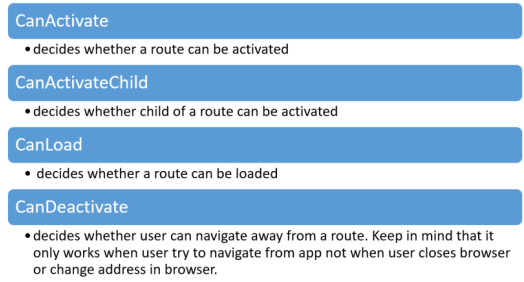


Figure 9.16

Let us create a simple route guard. To create a route guard, you need to follow steps:

1. Create a class.
2. Implement required route guard interface.

To create **`canActivate`** route guard, you need to implement **`canActivate`** interface in a class as shown in *code listing 9.28*:

Code Listing 9.28

```
import { Injectable } from '@angular/core';
import {
    CanActivate,
    ActivatedRouteSnapshot,
    RouterStateSnapshot
} from '@angular/router';

@Injectable()
export class CanActivateProductRouteGuard implements
CanActivate {

    constructor() { }

     canActivate(route: ActivatedRouteSnapshot, state:
RouterStateSnapshot): boolean {
        return false;
    }
}
```

In above route guard, activation logic is set to false, however in real application you will use a authentication service to determine, whether a particular route should be activated or not. Let us assume that you have a service to find whether a user should navigate to product route or not. In that case, you can use the service as shown in *code listing 9.29.*

Code Listing 9.29

```
 import { Injectable } from '@angular/core';
import {
    CanActivate,
    ActivatedRouteSnapshot,
    RouterStateSnapshot
```

```
} from '@angular/router';

import { ProductAuthService } from './product.auth.
service';

@Injectable()
export class CanActivateProductRouteGuard implements
CanActivate {

    constructor(private loginauth: ProductAuthService)
{ }

     canActivate(route: ActivatedRouteSnapshot, state:
RouterStateSnapshot): boolean {
        return this.loginauth.isUserAuthenticated();
    }
}
```

To use a route guard, first you need to pass it in providers array of module, and then use it in route as shown in *code listing 9.30*.

Code Listing 9.30

```
const productRoutes: Routes = [
    {path: 'products', component: ProductsComponent,
canActivate : [CanActivateProductRouteGuard]},
  {path: 'addproduct', component: AddproductComponent},
        {path: 'productdetails/:id', component:
ProductdetailsComponent}
];
```

You are passing route guard in **`canActivate`** property of route. In this way, you can create and use a route guard in Angular routing.

Summary

In this chapter, we learnt about Routing and navigating from one route to another. We also learnt how there can be different routes like parent and child routes. In this chapter you learnt about following topics:

- Creating Routes
- Route Strategies

- Dynamic route Parameters
- Navigating using code
- Query Parameter
- Route Guards

——****——

CHAPTER 10

Change Detection

In this chapter, you will learn about Change Detection in Angular. Following topics will be covered in this chapter:

- Change Detection
- Default Strategy
- onPush Strategy

Change Detection

Change Detection means updating the DOM whenever data is changed. There could be various reasons of Angular change detector to come into action and start traversing the component tree. They are:

- Events fired such as button click, etc.
- AJAX call or XHR requests.
- Use of JavaScript timer functions such as setTimeOut , SetInterval.

Angular provides two strategies for Change Detection.

1. Default strategy
2. onPush strategy

Default Strategy

In default strategy, whenever any data is mutated or changed, Angular will run change detector to update the DOM. In onPush strategy, Angular will run change detector only when new reference is passed to **@Input()** data. To update the DOM with updated data, Angular provides its own change detector to each component, which is responsible to detect change and update the DOM.

Let us say we have a **`MessageComponent`** as shown in *code listing 10.1*.

Code Listing 10.1

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

```
@Component({
  selector: 'app-message',
  template: `
    <h2>
    Hey {{person.firstname}} {{person.lastname}} !
    </h2>
    `
})
export class MessageComponent {

  @Input() person;
}
```

In addition, we are using **MessageComponent** inside **AppComponent** as shown in *code listing 10.2*.

Code Listing 10.2

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

@Component({
    selector: 'app-root',
    template: `
  <app-message [person]='p'></app-message>
  <button (click)='changeName()'>Change Name</button>
  `
})
export class AppComponent implements OnInit {
    p: any;
    ngOnInit(): void {
        this.p = {
            firstname: 'Brad',
            lastname: 'Cooper'
        };
    }
}
```

Let us talk through the code: all we are doing is using **`MessageComponent`** as child inside **`AppComponent`** and setting value of person using the property binding. At this point on running the application, you will get name printed as output.

Next, let us go ahead and update firstname property on the button click in **`AppComponent`** class as shown in *code listing 10.3.*

Code Listing 10.3

```
changeName() {
        this.p.firstname = 'Foo';
    }
```

As soon as we changed the property of mutable object P, Angular fires the change detector to make sure that the DOM (or view) is in sync with the model (in this case object p). For each property changes, Angular change detector will traverse the component tree and update the DOM.

Let us start with understanding about component tree. An Angular application can be seen as a component tree. It starts with a root component and then go through to child components. In Angular, data flows from top to bottom in the component tree as shown in *figure 10.1.*

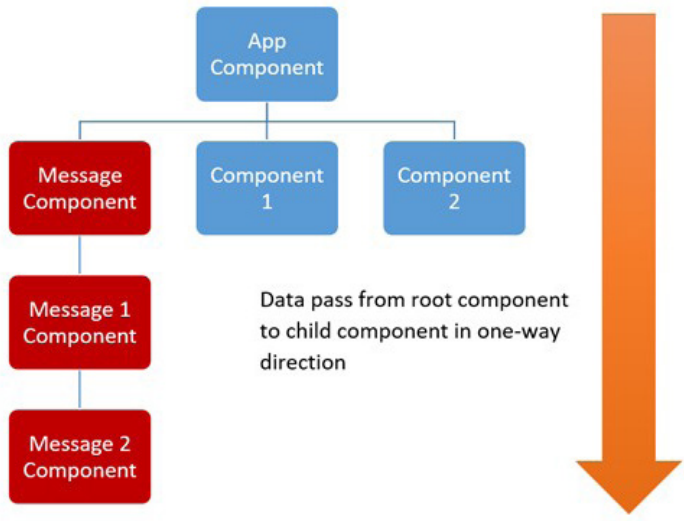


Figure 10.1

Whenever @**Input** type property will be changed, Angular change detector will start form the root component and traverse all child components to update the DOM. Any changes in primitive type's property will cause Angular change detection to detect the change and update the DOM.

In previous examples, you will find that on click of the button, the first name in the model will be changed. Then, change detection will be fired to traverse from root to bottom to update the view in **MessageComponent**.

Now, as you see, a single property change can cause change detector to traverse through the whole component tree. Traversing and change detection is a heavy process, which may cause performance degradation of application. Imagine that there are thousands of components in the tree and mutation of any data property can cause change detector to traverse all thousand components to update the DOM. To avoid this, there could be scenario when you may want to instruct Angular that when change detector should run for a component and its subtree, you can instruct a component's change detector to run only when object references changes instead of mutation of any property by choosing **onPush** Change Detection strategy.

onPush Strategy

You may wish to instruct Angular to run change detection on components and their sub-tree only when new references are passed to them versus when data is simply mutated by setting change detection strategy to **`onPush`**.

Let us go back to our example where we are passing object to **`MessageComponent`**. In the last example, we just changed firstname property and that causes change detector to run and to update the view of **`MessageComponent`**. However, now we want change detector to only run when reference of passed object is changed instead of just a property value. To do that let us modify **`MessageComponent`** to use **`OnPush ChangeDetection`** strategy. To do this set **changeDetection** property of **`@Component`** decorator to **`ChangeDetectionStrategy.OnPush`** as shown in *code listing 10.4*.

Code Listing 10.4

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

@Component({
  selector: 'app-message',
  template: `
    <h2>
    Hey {{person.firstname}} {{person.lastname}} !
    </h2>
```

```
    `,
    changeDetection: ChangeDetectionStrategy.OnPush,
})
export class MessageComponent {

  @Input() person;
}
```

At this point when you run the application, on click event of the button in **AppComponent** change detector will not run for **MessageComponent**, as only a property is being changed and reference is not changing. Since change detection strategy is set to **onPush**, now change detector will only run when reference of **@Input** property is changed as shown in *code listing 10.5.*

Code Listing 10.5

```
changeName() {
        this.p = {
            firstname: 'Foo',
            lastname: 'Kooper'
        };
    }
```

We are changing reference of object instead of just mutating just one property. Now when you run application, you will find on the click of the button that the DOM is being updated with new value.

By using **onPush** Change Detection, Angular will only check the tree if the reference passed to the component is changed instead of some property changed in the object. We can summarize that, and Use Immutable Object with **onPush** Change Detection to improve performance and run the change detector for component tree when object reference is changed.

We can further improve performance by using RxJS Observables because they emit new values without changing the reference of the object. We can subscribe to the observable for new value and then manually run **ChangeDetector**. Let us modify **AppComponent** to pass an observable to **MessageComponent** as shown in *code listing 10.6.*

Code Listing 10.6

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

```
@Component({
    selector: 'app-root',
    template: `
  <app-message [person]='data'></app-message>
  <button (click)='changeName()'>Change Name</button>
  `
})
export class AppComponent implements OnInit {
    p: any;
    data: any;
    ngOnInit(): void {
        this.p = {
            firstname: 'Brad',
            lastname: 'Cooper'
        };
        this.data = new BehaviorSubject(this.p);
    }

    changeName() {

        this.p = {
            firstname: 'Foo',
            lastname: 'Kooper'

        };
        this.data.next(this.p);
    }
}
```

In the code, we are using **`BehaviorSubject`** to emit next value as an observable to the **`MessageComponent`**. We have imported **`BehaviorSubject`** from RxJS and wrapped object p inside it to create an observable. On the click event of the button, it's fetching the next value in observable stream and passing to **`MessageComponent`**.

In the **`MessageComponent`**, we have to subscribe to the person to read the data as shown in *code listing 10.7.*

Code Listing 10.7

```
import { Component, Input, ChangeDetectionStrategy,
OnInit } from '@angular/core';
import { Observable } from 'rxjs';
```

```
@Component({
  selector: 'app-message',
  template: `
    <h2>
Hey {{_data.firstname}} {{_data.lastname}} !
   </h2>
    `,
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class MessageComponent implements OnInit {
  @Input() person: Observable<any>;
  _data;
  ngOnInit() {
    this.person.subscribe(data => {
      this._data = data;

    });
  }
}
```

Now, on click of the button, a new value is being created, however, a new reference is not being created as object is an observable object. Since a new reference is not created, due to **`onPush ChangeStrategy`**, Angular is not doing change detection. In this scenario, to update the DOM with new value of observable, we have to manually call the change detector as shown in *code listing 10.8.*

Code Listing 10.8

```
import { Component, Input, ChangeDetectionStrategy,
OnInit, ChangeDetectorRef } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-message',
  template: `
    <h2>
    Hey {{_data.firstname}} {{_data.lastname}} !
    </h2>
    `,
  changeDetection: ChangeDetectionStrategy.OnPush,
```

```
})
export class MessageComponent implements OnInit {

  @Input() person: Observable<any>;
  _data;

  constructor(private cd: ChangeDetectorRef) { }
  ngOnInit() {
      this.person.subscribe(data => {
          this._data = data;
          this.cd.markForCheck();
      });
  }
}
```

We have imported **`ChangeDetectorRef`** service and injected it, and then calling **`markForCheck()`** manually to cause change detector to run each time observable emits a new value. Now when you run application, and click on button, observable will emit new value and change detector will update the DOM also, even though a new reference is not getting created.

In a nutshell,

- If Angular ChangeDetector is set to default then for any change in any model property, Angular will run change detection traversing component tree to update the DOM.
- If Angular ChangeDetetor is set to onPush then Angular will run change detector only when new reference is being passed to the component.
- If observable is passed to the onPush change detector strategy enabled component then Angular ChangeDetctor has to be called manually to update the DOM.

Summary

Angular change detection strategy is very essential for performance of an application. You need to know when to use default strategy and when to use **`onPush`** strategy. In this chapter, you learnt about:

- Change Detection
- Default Strategy
- onPush Strategy

——****——

CHAPTER 11

Services and Providers

Service is one of the most important concept of Angular. Usually Service is a singleton class, and it is injected in a component or other service by Angular DI container. In this chapter, you will learn about the following:

- Services
- Providers
- Injectors

Services

Service is a singleton class. Angular creates one instance of service class per injection. Services is a class in Angular with a specific purpose. You may write service for the following scenarios:

- Logging
- Authentication
- Working with API
- Authorization
- Sharing data between unrelated components

Angular service should be singleton and we should configure that while creating. You can use Angular CLI to create a service. Use the following command::

`ng` *Generate Service App*

This command will generate **`AppService`** as shown in *code listing 11.1*.

Code Listing 11.1

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

@Injectable({
  providedIn: 'root'
})
```

```
export class AppService {

  constructor() { }
}
```

Let us talk through the code,

- Service is a class decorated with @Injectable() decorator.
- provideIn property of injector makes service singleton by injecting it at root module.
- In constructor, you can inject another services such as $http to make API call etc.

Let us modify **`AppService`** such that we can use it inside a component. We modified service and added a function **`getData`**. Right now function is returning simple string, however it can return observables and perform complex operations. Consider *Code Listing 11.2*.

Code Listing 11.2

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

@Injectable({
  providedIn: 'root'
})
export class AppService {

  constructor() { }

  getData() {
    return 'Hey I am from Service';
  }
}
```

To use service, do the following steps:

1. Import it in module in which you want to use.
2. Pass it in a provider array. You can learn more about providers in next section.
3. Inject it in constructor of Component to use it.

In component constructor pass service as shown in *code listing 11.3*, such as Angular perform Dependency Injection to pass object to component.

Code Listing 11.3

```
constructor(private service: AppService) {

  }
```

You can use service methods as shown in *code listing 11.4*.

Code Listing 11.4

```
export class AppComponent implements OnInit {
  data;
  constructor(private service: AppService) {

  }
  ngOnInit() {
    this.data = this.service.getData();
    console.log(this.data);
  }
}
```

In above listing we are calling **`getData`** method of service inside **`ngOnInit`** life cycle hook and assigning return data to local variable.

Services are mainly used to work with API using Angular built-in service $http, authentications, authorizations, logging, and so on.

Providers

An Angular Service provider delivers a runtime version of a dependency value. Therefore, when you inject a service, the Angular injector looks at the providers to create the instance of the service. It is the provider that determines which instance or value should be injected at the runtime in component, pipes, or directives. There are many jargons involved here, so to understand purpose of types of providers, let us start with creating a service. Let's say we have a service called **`ErrorService`**, which is just logging the error message as shown in *code listing 11.5*.

Code Listing 11.5

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

@Injectable()
export class ErrorService {
```

```
     logError(message: string) {
        console.log(message);
    }
}
```

Now, we can use this service in a component, as shown in *code listing 11.6.*

Code Listing 11.6

```
import { Component } from '@angular/core';
import { ErrorService } from './errormessage.service';

@Component({
    selector: 'app-root',
    template: `
  <input [(ngModel)]='err' type='text'/>
  <button (click)='setError()'>set error</button>
  `,
    providers: [ErrorService]
})
export class AppComponent {
    constructor(private errorservice: ErrorService) { }
    err: string;
    setError() {
        this.errorservice.logError(this.err);
    }

}
```

We are importing the service, passing it in the providers array, and injecting it in the constructor of component. We are calling the service method on click of the button, and you can see error message passed in the console. Very simple right?

Here, Angular will rely on the values passed in the providers array of component (or module) to find which instance should be injected at the run time.

A Provider determines how object of certain token can be created.

useClass

When you pass a service name in providers array of either component or module, as shown in *code listing 11.7*:

Code Listing 11.7

```
providers: [ErrorService]
```

Here, Angular is going to use token value **`ErrorService`** and, for token **`ErrorService`**, it will create object of **`ErrorService`** class. The above syntax is a shortcut of the syntax shown in *code listing 11.8*.

Code Listing 11.8

```
  providers: [{
    provide: ErrorService, useClass: ErrorService
}]
```

The provide property holds the token that serves as the key for the following:

- Locating the dependency value.
- Registering the dependency.

The second property (it is of four types) is used to create the dependency value. There are four possible values of second parameter, as follows:

1. useClass
2. useExisting
3. useValue
4. useFactory

We just saw example of **`useClass`**. Now, consider a scenario that you have a new class for better error logging called **`NewErrorService`** as shown in *code listing 11.9*.

Code Listing 11.9

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

@Injectable()
export class NewErrorService {

    logError(message: string) {
        console.log(message);
```

```
        console.log('logged by DJ');
    }

}
```

useExisting

Now, we want that instead of the instance of **ErrorService**, the instance of **NewErrorService** should be injected. Also, ideally, both classes must be implementing the same Interface, which means they will have same method signatures with different implementation. So now, for the token **ErrorService**, we want the instance of **NewErrorService** to be injected. It can be done by using **useClass**, as shown in *code listing 11.10*.

Code Listing 11.10

```
providers: [
    NewErrorService,
    { provide: ErrorService, useClass: NewErrorService
}
]
```

The problem with the above approach is that there will be two instances of **NewErrorService**. This can be resolved by the use of **useExisting** as shown in *code listing 11.11*.

Code Listing 11.11

```
providers: [
    NewErrorService,
   { provide: ErrorService, useExisting: NewErrorService
}
]
```

Now there will be only one instance of **NewErrorService** and for token **ErrorService** instance of **NewErrorService** will be created. Let us modify component to use **NewErrorService** as shown in *code listing 11.12*.

Code Listing 11.12

```
import { Component } from '@angular/core';
import { ErrorService } from './errormessage.service';
import { NewErrorService } from './newerrormessage.
service';
```

```
@Component({
    selector: 'app-root',
    template: `
  <input [(ngModel)]='err' type='text'/>
  <button (click)='setError()'>set error</button>
    <button  (click)='setnewError()'>Set  New  eroor</
button>
  `,
    providers: [
        NewErrorService,
              { provide: ErrorService, useExisting:
NewErrorService }
    ]
})
export class AppComponent {
      constructor(private errorservice: ErrorService,
private newerrorservice: NewErrorService) { }
    err: string;
    setError() {
        this.errorservice.logError(this.err);
    }
    setnewError() {
        this.newerrorservice.logError(this.err);
    }

}
```

Here, for demonstration purpose, I am injecting service in the component, however, to use service at the module level you can inject in the module itself. Now, on the click of set error button **`NewErrorService`** would be called.

useValue

Both **`useClass`** and **`useExisting`** create instance of a service class to inject for a particular token, but sometimes you want to pass value directly instead of creating instance. So, if you want to pass readymade object instead of instance of a class, you can use **`useValue`** as shown in *code listing 11.13*.

Code Listing 11.13

```
providers: [
  {
      provide: ErrorService, useValue: {
          logError: function (err) {
             console.log('inhjected directly ' + err);
          }
      }
  }
```

So here, we are injecting a readymade object using **`useValue`**. So for token **`ErrorService`**, Angular will inject the object.

useFactory

There could be scenario where, until runtime, you do not have idea about what instance is needed. You need to create dependency on the basis of information you do not have until the last moment. Let us consider example that you may need to create instance of **`ServiceA`** if user is logged in or **`ServiceB`** if user is not logged in. Information about logged in user may not be available until last time or may change during the use of application.

We need to use **`useFactory`** when information about dependency is dynamic. Let us consider our two services used in previous examples:

1. ErrorMessageService
2. NewErrorMessageService

For token **`ErrorService`** on a particular condition we want either instance of **`ErrorMessageService`** or **`newErrorMessageService`**. We can achieve that using **useFactory** as shown in *code listing 11.14*.

Code Listing 11.14

```
providers: [
  {
      provide: ErrorService, useFactory: () => {
          let m = 'old'; // this value can change
          if (m === 'old') {
              return new ErrorService();
          } else {
              return new NewErrorService();
```

```
            }
        }
    }
]
```

Here, we have taken a very hardcoded condition. You may have complex conditions to dynamically create instance for a particular token. Also, keep in mind that in **`useFactory`**, you can pass dependency. So assume that you have dependency on **`LoginService`** to create instance for **`ErrorService`** token. For that pass **`LoginService`** is deps array as shown in *code listing 11.15*.

Code Listing 11.15

```
providers: [
  {
      provide: ErrorService, useFactory: () => {
          let m = 'old'; // this value can change
          if (m === 'old') {
              return new ErrorService();
          } else {
              return new NewErrorService();
          }
      },
      deps: [LoginService]
  }
]
```

These are fours ways of working with providers in Angular.

Using an Injector

To understand injector, let us revisit Dependency Injection; it is a design pattern, in which a class request external program called DI container for its dependencies. Angular framework has DI container in its design.

For example, **`AppComponent`** has dependency on **`ErrorService`**. Now instead of creating instance of **`ErrorService`** itself, **`AppComponent`** will ask Angular DI container to create an object and pass it to that. Whenever you pass a service in constructor of an Angular class, DI container will create the object of service class and inject to the component. In *code listing 11.16*, Angular will inject object of **`ErrorService`** to **`AppComponent`**.

Code Listing 11.16

```
export class AppComponent {

    constructor(private d: ErrorService){

    }
}
```

Keep in mind that DI container provides declared dependency when the class is initiated. DI container injects the class in a component, which is decorated with **`@Injectable`**. Any class with **`@Injectable`** decorator can be injected, however to inject it you have to configure the provider, which you learnt in previous section.

A provider instructs an injector how to create the service. You must configure an injector with a provider before that injector can create a service. Injector can be configured at various level of app:

- In the @Injectable() decorator for the service itself. Service will be provided at application root level and will be available in entire application.
- In the @NgModule() decorator for an NgModule. Service will be provided at module level and will be available in the configured module.
- In the @Component() decorator for a component. Service will be provided at component level and will be scoped to the component.

In Service using `provideIn`

At time of creating service, you can use property of **`@Injectable`** decorator as shown in *code listing 11.17* to determine the injector.

Code Listing 11.17

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

@Injectable({
  providedIn: 'root'
})
export class AppService {

  constructor() { }
}
```

@Injector() provideIn property provides or inject the service at the root level of application. When you provide service using **`provideIn`**, Angular provides this service at **`App Root level`**. Since service is provided at root level, Angular can easily optimize the app by removing service reference, if it is not being used. One other important thing, you keep in mind that service provided using **`@Injectbale() provideIn`** is available in entire application. All components, sub modules etc. will use the same service object created at the application root level. If you have re injected the same service in some feature module that would be ignored an object created at application root level will be used.

In `ngModule` Providers Array

You can provide service at a particular module level also, for that; you have to pass service in providers array of **`@ngModule`** decorator as shown in *code listing 11.18*.

Code Listing 11.18

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

In this scenario, service will be scoped to **`AppModule`** level. Any component, sub modules, pipes etc. of **`AppModule`** can use **`AppService`**.

In Component

You can provide service at a particular component level also, for that; you have to pass service in providers array of **`@component`** decorator as shown in *code listing 11.19.*

Code Listing 11.19

```
@Component({
```

```
    selector: 'app-root',
    providers: [AppService],
    templateUrl: 'app.component.html'
  `
})
export class AppComponent {
```

In this scenario, service will be scoped to **`AppComponent`** level.

Summary

Services are essential of Angular application. In this chapter, you learnt about following topics:

- Services
- Providers
- Injectors

———****———

CHAPTER 12

Working with API and $http

Angular provides $http service to work with REST API in an Angular application. You can use $http service with RxJS operators to perform various tasks with the API. You can also use $http services to execute various HTTP request from Angular application to an API to perform CRUD operations on the data residing on the server. In this chapter, you will learn:

- Angular in-memory web api
- Setting up Angular in-memory web api
- Create service to perform HTTP operations
- Read Data
- Create Data
- Update Data
- Delete Data

Angular in-memory Web Api

In real application, you will have data residing on the server, and an Angular application will perform the CRUD operations on the data of server through API. Standard is, you perform various HTTP operations to execute CRUD operations. To read data from the server you perform HTTP GET operation and to create data you perform HTTP POST operation. You perform various operations as shown in figure 12.1

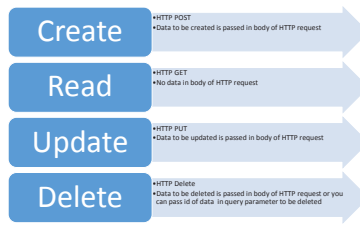


Figure 12.1

In real application, you will have real server and API. However, for development and testing purpose you can fake a backend server. There are three ways; you can fake a backend server in Angular.

1. Return hardcode data from a local file
2. Use a local JSON file
3. Use angular in-memory web api

Problem with hardcoded data is you do not have to make http request to read data from a text file. With local JSON file, you can make http request but again JSON file is not best suited for all CRUD operations. To overcome these issues, Angular comes up with in-memory web api to simulate a data server. You should use Angular in-memory web api for demos and tests that emulates CRUD operations over a REST API.

Angular in-memory web api is very useful for development and testing purpose. Once you have installed it in your application, it intercepts HTTP request and returns mock data. You do not have to hardcode data or need ready backend API for development or testing purposes. You can almost do everything with in-memory web API like backend API such as:

- Setting request header
- Setting response header
- Setting response type

Angular in-memory web API intercepts Angular **`Http`** and **`HttpClient`** requests that would otherwise go to the remote API and redirects them to an in-memory data store that you have created. There are various use cases when you should use Angular in-memory web api such as:

- To mock Server API for demo Apps without having a real server. You can mock persistence of Data of CRUD operation locally.
- To simulate operations against data, yet to be created on server.
- To unit test of services doing read and write of data using http.
- To perform end-to-end test without affecting real database at the server.
- To use in continuous integration (CI) builds in which you need to mock real database server.

Setting Up Angular in-memory Web Api

You can install Angular in-memory web api in your project using **`npm`** as shown in *code listing 12.1*.

Code Listing 12.1

```
npm install angular-in-memory-web-api --save-dev
```

After successful installation, you can see dependency added for Angular in-memory web api in **`package.json`** file line number 35 as shown in *figure 12.2.*

Figure 12.2

We have added Angular in-memory web api dependencies in the project, now to simulate CRUD operations, create an entity class for Car as shown in *code listing 12.2*.

Code Listing 12.2

```
export class Car {
    constructor ( public id = 0,
                  public model = '',
                  public price= 0,
                  public color = '') {

    }
}
```

Once entity class is created, you need to override **`createDb()`** method of **`InMemoryDbService`** abstract class, to do that, create a new class and implement **`InMemoryDbService`** class as shown in the *code listing 12.3*.

Code Listing 12.3

```
import {InMemoryDbService} from 'angular-in-memory-
web-api';
```

```
import { Car } from './car';

export class CarInMemDataService implements
InMemoryDbService {
    createDb() {
      const cars: Car[] = [
          { id: 1, model: 'BMW', price: 40000, color:
'Red'  },
          { id: 2, model: 'Audi', price: 45000, color:
'Blue'  },
          { id: 3, model: 'Tata', price: 20000, color:
'White'  },
         { id: 4, model: 'Honda', price: 30000, color:
'Black' },
       { id: 5, model: 'GM', price: 30000, color: 'Red'
}
       ];
      return { cars};
    }
}
```

As you see, we are overriding created method, which returns collections. In this case, there is only one-collection car returns; however, you can return any number of collections. To use in-memory web api, you need to import in the required module. You can import it in **`AppModule`** as shown in *code listing 12.4*.

Code Listing 12.4

```
import { InMemoryWebApiModule } from 'angular-in-
memory-web-api';
import { CarInMemDataService } from './car-in-mem-data.
service';

@NgModule({
  imports: [
    BrowserModule,
    InMemoryWebApiModule.forRoot(CarInMemDataService)
  ],
```

We are passing **`CarInMemDataService`** class in which created method is implemented to **`InMemoryWebApiModule`**.

So far, we have added dependency of Angular in-memory web api, created entity class, and implemented created method. Now Angular in-memory web api is ready to be used for CRUD operations. You can access created API at URL - … **/api/cars.** Various HTTP operations can be performed as shown in the following code:

```
http.post(api/cars, undefined);
http.get(api/cars);
http.post('commands/cars, '{"delay":1000}');
```

Now we have Angular in-memory web api to perform CRUD operations on Car data. In next section, we will use to perform operations.

Create Service to Perform HTTP Operations

Let us create an Angular service, in which we will perform HTTP operations. We will inject this service in the required components. You can add service in the project using CLI command:

ng g Service App

You will have **AppService** created after successful running of command. In created service import items as shown in *code listing 12.5* to perform HTTP operations:

Code Listing 12.5

```
import { HttpClient, HttpHeaders, HttpParams } from '@
angular/common/http';
import { Observable, throwError } from 'rxjs';
import { tap, catchError, map } from 'rxjs/operators';
import { Car } from './car';
```

After importing required files, we need to inject **HttpClient** service in **AppService** as shown in *code listing 12.6.*

Code Listing 12.6

```
@Injectable({
  providedIn: 'root'
})
export class AppService {

  apiurl = 'api/cars';
```

```
    headers = new HttpHeaders().set('Content-Type',
'application/json').set('Accept', 'application/json');
  httpOptions = {
    headers: this.headers
  };

  constructor(private http: HttpClient) {

  }

  private handleError(error: any) {
    console.error(error);
    return throwError(error);
  }

  getCars(): Observable<Car[]> {
    return this.http.get<Car[]>(this.apiurl).pipe(
      tap(data => console.log(data)),
      catchError(this.handleError)
    );
  }
}
```

We have also added **`handleError`** method to log any error encounters while making HTTP calls. In real applications, you should make sure to log error to the server. In addition, we have created a **`httpOptions`** variable to be used while making http operations.

Next, you need to import **`HttpClientModule`** in application module to work with **`HttpClient`** service. You can import **`HttpClientModule`** with other required module as shown in *code listing 12.7.*

Code Listing 12.7

```
import { InMemoryWebApiModule } from 'angular-in-
memory-web-api';
import { CarInMemDataService } from './car-in-mem-data.
service';
import { HttpClientModule } from '@angular/common/
http';

@NgModule({
```

```
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    InMemoryWebApiModule.forRoot(CarInMemDataService)
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

As of now, we have created service class and injected **`HttpClientModule`** in that to perform HTTP operations.

Read Data

To read data from API, you need to use **`http.get()`** method. In the method, pass **apiurl** as shown in *code listing 12.8*.

Code Listing 12.8

```
  getCars(): Observable<Car[]> {
    return this.http.get<Car[]>(this.apiurl).pipe(
      tap(data => console.log(data)),
      catchError(this.handleError)
    );
  }
```

We are using pipe and tap operators to read data from observable. If there is any error in HTTP get method request, Angular will call **`catchError`** function.

You can use **`getCars`** method in a component as shown in *code listing 12.9*.

Code Listing 12.9

```
export class AppComponent implements OnInit {

  cars: Car[] = [];
  constructor(private appservice: AppService) {
```

```
  }
  ngOnInit() {
    this.getCars();
  }

  getCars() {
    this.appservice.getCars().subscribe(data => {
      this.cars = data;
    });
  }
}
```

Let us talk though code:

- We are injecting AppService in AppComponent
- Calling getCars method and subscribing to returned observable
- Assigning returned data to local variable cars
- Calling the getCars method in onInit life cycle hook

On the template, you can simply use cars array that holds data from Angular in-memory web api to display data as shown in *code listing 12.10*.

Code Listing 12.10

```
<table>
  <tr *ngFor="let c of cars">
    <td>
      {{c.id}}
    </td>
    <td>
      {{c.model}}
    </td>
    <td>
      {{c.color}}
    </td>
    <td>
      {{c.price}}
    </td>
  </tr>
</table>
```

You can read a particular car with its id property. You can read data with 'id' as shown in *code listing 12.11*. Add method **`getCar`** in **`AppService`**.

Code Listing 12.11

```
getCar(id: number): Observable<Car> {
    const url = `${this.apiurl}/${id}`;
    return this.http.get<Car>(url).pipe(
      catchError(this.handleError)
    );
  }
```

You can use **getCar** method in component as shown in *code listing 12.12.*

Code Listing 12.12

```
idtofetch = 1;
  getCar() {
                this.appservice.getCar(this.idtofetch).
subscribe(data => {
      this.car = data;
    });
  }
```

We are subscribing to returned observable from service and assigning returned data to local variable car. Also while calling **`getCar`** method of service, you need to pass id to be fetched. On the template, user can enter ID and fetched result can be displayed as shown in *code listing 12.13.*

Code Listing 12.13

```
<input      type="number"      [(ngModel)]='idtofetch'
placeholder="Enter Id" />
<button (click)='getCar()'>Fetch Car </button>
<div>
  <h3>Fetched Car with Id : {{idtofetch}}</h3>
  {{car.id}}
  {{car.model}}
  {{car.price}}
  {{car.color}}
</div>
```

Create Data

To create a record in Angular in-memory web api, you need to perform **`HTTP POST`** operation and pass object to be inserted in the database. To

do that add a method **addCar** in **AppService** as shown in *code listing 12.14.*

Code Listing 12.14

```
addCar (car: Car): Observable<Car> {
         return this.http.post<Car>(this.apiurl, car,
this.httpOptions).pipe(
      tap(data => console.log(data)),
      catchError(this.handleError)
    );
  }
```

We are performing HTTP POST operation, passing car object to be inserted and setting HTTP options such as HTTP headers to Application/JSON. Angular in-memory web api will return newly created object ON successful completion of operation.

Next, you can use **addCar** method from **AppService** in component as shown in *code listing 12.15.*

Code Listing 12.15

```
addCar() {
      this.appservice.addCar(this.carFormGroup.value).
subscribe(data => {
      this.car = data;
      console.log(this.car);
    });
    this.getCars();
  }
```

We are calling **addCar** method of service and passing car object to be inserted. We have created car object using Reactive Form as shown in *code listing 12.16.*

Code Listing 12.16

```
this.carFormGroup = new FormGroup(
      {
        model : new FormControl(this.carToAdd.model),
        price : new FormControl(this.carToAdd.price),
        color: new FormControl(this.carToAdd.color)
      }
    );
```

On the template, form is created as shown in *code listing 12.17*.

Code Listing 12.17

```
<form [formGroup]='carFormGroup' (ngSubmit)='addCar()'
novalidate >
<input       type="text"       formControlName='model'
placeholder="Enter Model"   />
<input       type="text"       formControlName='color'
placeholder="Enter Color"   />
<input      type="number"      formControlName='price'
placeholder="Enter Price"   />
<button>Add Car</button>
```

Update Data

To update a record in Angular in-memory web api, you need to perform **`HTTP PUT`** operation and pass object to be updated in the database. To do that, add a method **`updateCar`** in **`AppService`** as shown in *code listing 12.18*.

Code Listing 12.18

```
updateCar (car: Car): Observable<null | Car> {
     return this.http.put<Car>(this.apiurl, car, this.
httpOptions).pipe(
       tap(data => console.log(data)),
       catchError(this.handleError)
     );
  }
```

We are performing HTTP PUT operation, passing car object to be updated and setting HTTP options such as HTTP headers to Application/JSON. Next, you can use **`updateCar`** method from **`AppService`** in component as shown in *code listing 12.19*.

Code Listing 12.19

```
 updateCar() {
               this.appservice.getCar(this.idtoupdate).
subscribe(data => {
       this.carToUpdate = data;
       this.carToUpdate.model = 'updated model';
```

```
          this.appservice.updateCar(this.carToUpdate).
subscribe(data1 => {
        this.getCars();
      });
    });
```

On template, you can have a form like create data example to accept user input for update.

Delete Data

To delete a record in Angular in-memory web api, you need to perform **HTTP DELETE** operation and pass id o to be deleted in the database. To do that add a method **`deleteCar`** in **`AppService`** as shown in *code listing 12.20.*

Code Listing 12.20

```
deleteCar (id: number): Observable<Car> {
    const url = `${this.apiurl}/${id}`;
   return this.http.delete<Car>(url, this.httpOptions).
pipe(
      catchError(this.handleError)
    );
  }
```

We are performing HTTP DELETE operation, passing car id to be deleted and setting HTTP options such as HTTP headers to Application/JSON. Next, you can use **`deleteCar`** method from **`AppService`** in component as shown in *code listing 12.21.*

Code Listing 12.21

```
  deleteCar() {
          this.appservice.deleteCar(this.idtodelete).
subscribe(data => {
       this.getCars();
    });
  }
```

On template, you can have a form like create data example to accept user input for delete.

Summary

In Angular, you use $http service to perform HTTP requests to API. You can have local fake API to perform CRUD operations using Angular in-memory web api. In this chapter, you learnt the following topics:

- Angular in-memory web api
- Setting up Angular in-memory web api
- Create service to perform HTTP operations
- Read Data
- Create Data
- Update Data
- Delete Data

——****——

CHAPTER 13

Advanced Components

In this chapter, you will learn about Content Projection and some other ways of component communication. Following topics will be covered in this chapter:

- Content Projection
- Using ViewChild
- Using ContentChild

Content Projection

In Angular, content projection is used to project content in a component. Content projection allows you to insert a shadow DOM in your component. To put it simply, if you want to insert HTML elements or other components in a component, then you do that using the concept of content projection. In Angular, you achieve content projection using **`< ng-content>< /ng-content >`**. You can make reusable components and scalable application by right use of content projection.

Using Content Projection

To understand content projection, let us consider **GreetComponent** as shown in the *code listing 13.1*.

Code Listing 13.1

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

@Component({
    selector: 'app-greet',
    template: `{{message}}`
})
export class GreetComponent {
```

```
    @Input() message: string;
}
```

Using the **`@Input()`** decorator, you can pass string, numbers or object an to the **`GreetComponent`**, but what if you need to pass different types of data to the **`GreetComponent`** such as:

- Inner HTML
- HTML Elements
- Styled HTML
- Another Component

To project styled HTML or another component, content projection is used.

Code Listing 13.2

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

@Component({
    selector: 'app-greet',
    template: `<div>
    <ng-content></ng-content>
    </div>`
})
export class GreetComponent {
    @Input() message: string;
}
```

In the *Code listing 13.2*, you are projecting styled HTML to the **`GreetComponent`** and you'll get the output as in *figure 13.1*:

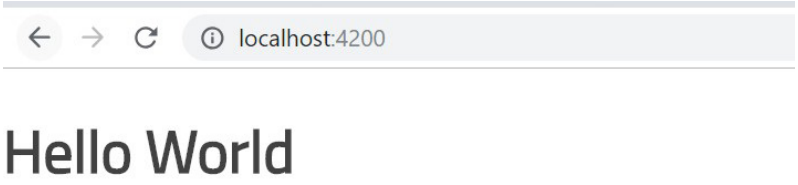


Figure 13.1

This is an example of Single Slot Content Projection. Whatever you pass to the **`GreetComponent`** will be projected. So, let us pass more than one HTML element to the **`GreetComponent`** as shown in the *code listing 13.3*.

Code Listing 13.3

```
<div>
  <app-greet>
    <h2>Hello World</h2>
    <button>Say Hello</button>
    <p>This is Content Projection</p>
  </app-greet>
</div>
```

Here we are passing three HTML elements to the **`GreetComponent`**, and all of them will be projected. You will get the output as shown in the *figure 13.2*.

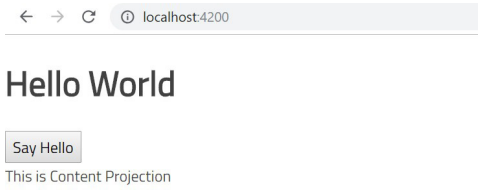


Figure 13.2

In the DOM, you can see that inside the **`GreetComponent`**, all HTML elements are projected. See *figure 13.3*.

Figure 13.3

Multi Slot Projection

You may have a requirement to project elements in multiple slots of the component. Multiple slots mean you have more than one **<ng-content>**. Let us modify **GreetComponent** as shown in the *code listing 13.4*:

Code Listing 13.4

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

@Component({
    selector: 'app-greet',
    template: `<div>
    <h2>{{message}}</h2>
    <ng-content></ng-content>
    <br/>
    <button (click)='sayHello()'>Hello</button>
    <ng-content></ng-content>
    </div>`
})
export class GreetComponent {
     message = 'Greetings';
     sayHello() {
        console.log('hello');
     }
}
```

Here we're using **ng-content** two times. Now, the question is, do we select a particular **ng-content** to project particular element? You can select a particular slot for projection using the **<ng-content>** selector. There are four types of selectors:

1. Project using tag selector
2. Project using class selector
3. Project using id selector
4. Project using attribute selector

You can use the tag selector for multi-slot projection as shown in the *code listing 13.5*.

Code Listing 13.5

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

```
@Component({
    selector: 'app-greet',
    template: `<div>
    <h2>{{message}}</h2>
    <ng-content select="p"></ng-content>
    <br/>
    <button (click)='sayHello()'>Hello</button>
    <ng-content select="button"></ng-content>
    </div>`
})
export class GreetComponent {
     message = 'Greetings';
     sayHello() {
        console.log('hello');
     }
}
```

Next, you can project content to the **`GreetComponent`** as shown in the *code listing 13.6*:

Code Listing 13.6

```
<div>
  <app-greet>
    <p>Jason</p>
   <button style ='background:red'>Register me</button>
  </app-greet>
  <app-greet>
    <p>Jason</p>
      <button style ='background:blue'>Register me</
button>
  </app-greet>
</div>
```

As you can see, we are using the **`GreetComponent`** twice and projecting different **p** and button elements with different style.

The problem with using tag selectors is that all **p** elements will get projected to the **`GreetComponent`**. In many scenarios, you may not want that and can use other selectors such as a class selector or an attribute selector, as shown in the *code listing 13.7*:

Code Listing 13.7

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

@Component({
    selector: 'app-greet',
    template: `<div>
    <h2>{{message}}</h2>
    <ng-content select=".paratext"></ng-content>
    <br/>
    <button (click)='sayHello()'>Hello</button>
    <ng-content select="[btnRegister]"></ng-content>
    </div>`
})
export class GreetComponent {
     message = 'Greetings';
     sayHello() {
        console.log('hello');
     }
}
```

Next, you can project content to the **`GreetComponent`** as shown in the *code listing 13.8*:

Code Listing 13.8

```
<div>
  <app-greet>
    <p class='paratext'>Jason</p>
   <button btnRegister style ='background:red'>Register
me</button>
  </app-greet>
  <app-greet>
    <p class='paratext'>Jason</p>
   <button btnRegister style ='background:blue'>Register
me</button>
  </app-greet>
</div>
```

You'll get the same output as above, however this time you are using the class name and attribute to project the content. When you inspect an

element on the DOM, you will find the attribute name and the class name of the projected element as shown in the *figure 13.4*:

```
▼<div _ngcontent-c0>
  ▶<app-greet _ngcontent-c0>…</app-greet>
  ▼<app-greet _ngcontent-c0>
    ▼<div>
       <h2>Greetings</h2>
       <p _ngcontent-c0 class="paratext">Jason</p>
       <br>
       <button>Hello</button>
       <button _ngcontent-c0 btnregister style="background:blue">Register me</button>
     </div>
   </app-greet>
```

Figure 13.4

Content Projection is very useful to project HTML, to other component.

ViewChild

In the previous section, you learnt about various ways of Component Communication. One of them is **ViewChild** and **ContentChild**. Since now, you know about Content Projection, you should be able to understand **ContentChild** besides **ViewChild**. Essentially **ViewChild** and **ContentChild** are used for component communication in Angular. Therefore, if a parent component wants access of child component then it uses **ViewChild** or **ContentChild**.

```
@Component({
    selector: 'app-greet',
    template: `
    <div>
    <h2>{{message}}</h2>
    <ng-content select=".paratext"></ng-content>
    <br/>
    <button (click)='sayHello()'>Hello</button>
    <ng-content select="[btnRegister]"></ng-content>
    </div>
    `
})
```

ViewChild

ContentChild

Figure 13.5

Any component, directive, or element which is part of a template is **ViewChild** and any component or element which is projected in the template is **ContentChild**.

If you want to access the following inside the Parent Component, use **@ViewChild** decorator of Angular.

- Child Component
- Directive
- DOM Element

ViewChild returns the first element that matches the selector. Let us assume that we have a component **MessageComponent** as shown in the *code listing 13.9*:

Code Listing 13.9

```
import { Component, Input } from '@angular/core';
@Component({
    selector: 'app-message',
    template: `<h2>{{message}}</h2>`
})
export class MessageComponent {
    @Input() message: string;
}
```

We are using **MessageComponent** inside **AppComponent** as shown in *code listing 13.10*:

Code Listing 13.10

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-root',
    template: `
  <div>
  <h1>Messages</h1>
  <app-message [message]='message'></app-message>
  </div>`
})
export class AppComponent implements OnInit {
    message: any;
    ngOnInit() {
```

```
        this.message = 'Hello World !';
    }
}
```

On running you will get output as *image 13.6*:

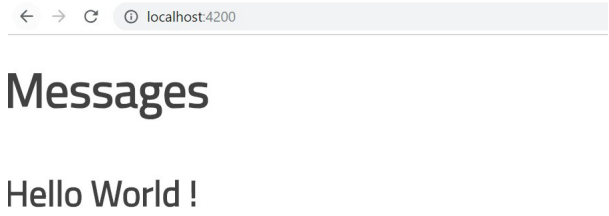


Figure 13.6

Here, **`MessageComponent`** has become child of **`AppComponent`**. Therefore, we can access it as a **`ViewChild`**. Definition of **`ViewChild`** is:

*The Child Element which is located inside the component template***,**

Here **`MessageComponent`** is located inside template of **`AppComponent`**, so it can be accessed as **`ViewChild`**. See *code listing 13.11*.

Code Listing 13.11

```
export class AppComponent implements OnInit,
AfterViewInit {
  message: any;
      @ViewChild(MessageComponent)  messageViewChild:
MessageComponent;

  ngAfterViewInit() {
    console.log(this.messageViewChild);
  }

  ngOnInit() {
    this.message = 'Hello World !';
  }
}
```

We need to do following tasks to work with **`ViewChild`:**

- Import ViewChild and AfterViewInit from @angular/core
- Implement AfterViewInit life cycle hook to component class
- Create a variable with decorator @ViewChild
- Access that inside ngAfterViewInit life cycle hook

In the output console you will find reference of **`MessageComponent`**, also if you can notice that __proto__ of **`MessageComponent`** is set to Object. See *figure 13.7*.

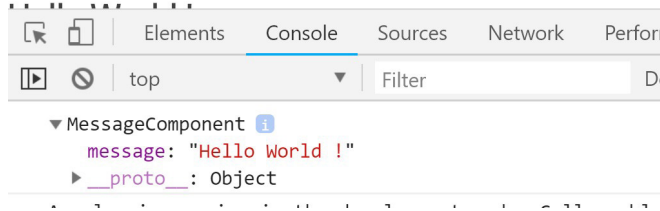


Figure 13.7

Now let us try to change value of **`MessageComponent`** property. See *code listing 13.12*.

Code Listing 13.12

```
  ngAfterViewInit() {
    console.log(this.messageViewChild);
      this.messageViewChild.message = 'Passed as View
Child';
  }
```

Here we are changing the value of **`ViewChild`** property, you will notice that value has been changed and you are getting output as shown in the *figure 13.8*:

Figure 13.8

However, in the console you will find an error: *Expression has changed after it was last checked***.** See *figure 13.9.*

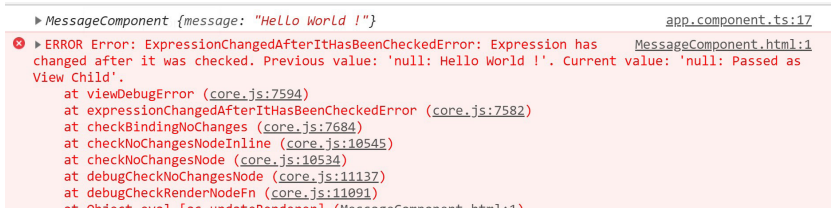


Figure 13.9

This error can be fixed in two ways,

- By changing the ViewChild property in ngAfterContentInit life cycle hook
- Manually calling change detection using ChangeDetectorRef

To fix it in **`ngAfterContentInit`** life cycle hook, you need to implement **`AfterContentInit`** interface as in *code listing 13.13.*

Code Listing 13.13

```
ngAfterContentInit() {
      this.messageViewChild.message = 'Passed as View
Child';
  }
```

Only problem with this approach is when you work with more than one **`ViewChild`** also known as **`ViewChildren`**. Reference of **`ViewChildren`** is not available in **`ngAfterContentInit`** life cycle hook. In that case, to fix the above error, you will have to use a change detection mechanism. To use the change detection mechanism:

- Import ChangeDetectorRef from @angular/core
- Inject it to the constructor of Component class
- Call detectChanges() method after ViewChild property is changed

You can use manual change detection as shown in *code listing 13.14*:

Code Listing 13.14

```
constructor(private cd: ChangeDetectorRef) {}

  ngAfterViewInit() {
```

```
      console.log(this.messageViewChild);
       this.messageViewChild.message = 'Passed as View
Child';
      this.cd.detectChanges();
  }
```

Manually calling change detection will fix *Expression has changed after it was last checked*, error and it can be used with **`ViewChildren`** also.

To understand **`ViewChildren`**, let us consider **`AppComponent`** class created as shown in *code listing 13.15*:

Code Listing 13.15

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-root',
    template: `
  <div>
  <h1>Messages</h1>
 <app-message *ngFor="let f of messages" [message]='f'></
app-message>
  </div>`
})
export class AppComponent implements OnInit {
    messages: any;
    ngOnInit() {
        this.messages = this.getMessage();
    }
    getMessage() {
        return [
            'Hello India',
            'Which team is winning Super Bowl? ',
            'Have you checked Ignite UI ?',
            'Take your broken heart and make it to the
art'
        ];
    }
}
```

We are using **`MessageComponent`** inside a ***`ngFor`** directive hence there are multiple references of **`MessageComponent`**. We can access it now as

`ViewChildren` and **`QueryList`** as shown in the *code listing 13.16*:

Code Listing 13.16

```
@ViewChildren(MessageComponent) messageViewChildren:
QueryList<MessageComponent>;
  ngAfterViewInit() {
    console.log(this.messageViewChildren);
  }
```

To work with **ViewChildren** and **QueryList**, you need to do the following tasks:

- Import ViewChildren , QueryList , AfterViewInit from @angular/core
- Make reference of ViewChildren with type QueryList
- Access ViewChildren reference in ngAfterViewInit() life cycle hook

In the output, you will get various reference of **`MessageComponent`** as **`ViewChildern`** as shown in the *figure 13.10:*

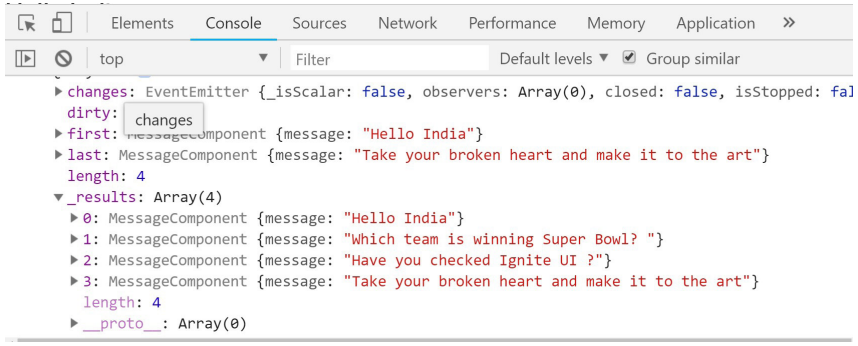


Figure 13.10

Now let us try to update properties of **`ViewChildren`** as shown in the *code listing 13.17*:

Code Listing 13.17

```
  ngAfterViewInit() {
    console.log(this.messageViewChildren);
    this.messageViewChildren.forEach((item) => { item.
message = 'Infragistics'; });
  }
```

As you see, we are iterating through each item of ViewChildren and updating each property. This will update property value but again you will get the error, *Expression has changed after it was last checked* as shown in the *figure 13.11*:

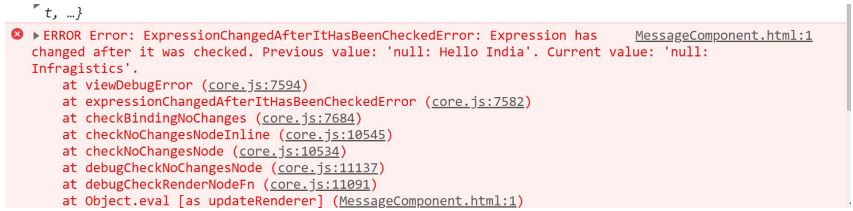


Figure 13.11

You can again fix it by manually calling change detection like **`ViewChild`**. Keep in mind that we do not have **`ViewChildren`** reference available in **`AfterContentInit`** life cycle hook. You will get undefined in **`ngAfterContentInit()`** life cycle hook for **`ViewChildren`** reference as shown in the *code listing 13.18*:

Code Listing 13.18

```
ngAfterContentInit() {
  console.log(this.messageViewChildren); // undefined
}
```

However, you can manually call change detection to fix error: *Expression has changed after it was last checked*

To use a change detection mechanism:

- Import ChangeDetectorRef from @angular/core
- Inject it to the constructor of Component class
- Call detectChanges() method after ViewChild property is changed

You can use a manual change detection like shown in *code listing 13.19*:

Code Listing 13.19

```
@ViewChildren(MessageComponent) messageViewChildren:
QueryList<MessageComponent>;
  constructor(private cd: ChangeDetectorRef) {
  }
  ngAfterViewInit() {
    console.log(this.messageViewChildren);
```

```
    this.messageViewChildren.forEach((item) => { item.
message = 'Infragistics'; });
    this.cd.detectChanges();
  }
```

In this way, you can work with **`ViewChild`** and **`ViewChildren`**.

`ContentChild`

Let us start with understanding about **`ContentChild`**. Any element which is located inside the template, is **`ContentChild`**. To understand it let us consider **`MessageContainerComponent`** as in the *code listing 13.20.*

Code Listing 13.20

```
import { Component } from '@angular/core';
@Component({
    selector: 'app-messagecontainer',
    template: `
    <div>
    <h3>{{greetMessage}}</h3>
    <ng-content select="app-message"></ng-content>
    </div>
    `
})
export class MessageContainerComponent {
    greetMessage = 'Ignite UI Rocks!';
}
```

In this component, we are using Angular Content Projection. You learnt about it in last section.

Any element or component projected inside **`<ng-content>`** becomes a **`ContentChild`**. If you want to access and communicate with **`MessageComponent`** projected inside **`MessageContainerComponent`**, you need to read it as **`ContentChild`**.

Before we go ahead and learn to use **`ContentChild`**, first see how **`MessageContainerComponent`** is used and **`MessageComponent`** is projected in the *code listing 13.21*:

Code Listing 13.21

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

```
@Component({
  selector: 'app-root',
  template: `
  <div>
  <app-messagecontainer>
  <app-message [message]='message'></app-message>
  </app-messagecontainer>
  </div>`
})
export class AppComponent implements OnInit {
  message: any;
  ngOnInit() {
    this.message = 'Hello World !';
  }
}
```

As you see in the above listing that in the **`AppComponent`**, we are using **`MessageContainerComponent`** and passing **`MessageComponent`** to be projected inside it. Since **`MessageComponent`** is used in **`MessageContainerComponent`** using content projection, it becomes **`ContentChild`**.

Now, you will get output as shown in *figure 13.12*:

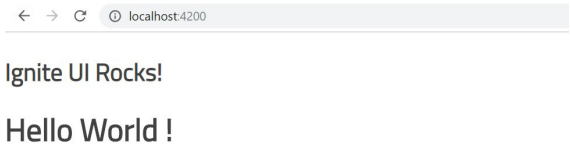


Figure 13.12

Since **`MessageComponnet`** is projected and is being used inside the template of **`MessageContainerComponent`**, it can be used as **`ContentChild`** as shown in the *code listing 13.22*:

Code Listing 13.22

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

```
import { MessageComponent } from './message.component';
@Component({
    selector: 'app-messagecontainer',
    template: `
    <div>
    <h3>{{greetMessage}}</h3>
    <ng-content select="app-message"></ng-content>
    </div>
    `
})
export class MessageContainerComponent implements
AfterContentInit {
    greetMessage = 'Ignite UI Rocks!';
                        @ContentChild(MessageComponent)
MessageComponnetContentChild: MessageComponent;
    ngAfterContentInit() {
       console.log(this.MessageComponentContentChild);
    }
}
```

We need to do the following tasks:

- Import ContentChild and AfterContentInit from @angular/core
- Implement AfterContentInit life cycle hook to component class
- Create a variable with decorator @ContentChild
- Access that inside ngAfterContentInit life cycle hook

In the output console you will find a reference of **MessageComponent**, also if you can notice that __proto__ of **MessageComponent** is set to **Object**. See *figure 13.13*.

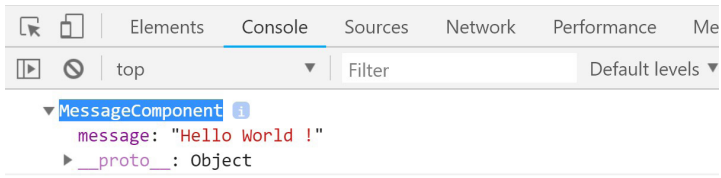


Figure 13.13

You can modify the **`ContentChild`** property inside **`ngAfterContentInit`** life cycle hook of the component. Let us assume that there is more than one **`MessageComponent`** is projected as shown in the *code listing 13.23*:

Code Listing 13.23

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-root',
    template: `
  <div>
    <app-messagecontainer>
         <app-message  *ngFor='let  m  of  messages'
[message]='m'></app-message>
    </app-messagecontainer>
  </div>`
})
export class AppComponent implements OnInit {
    messages: any;
    ngOnInit() {
        this.messages = this.getMessage();
    }
    getMessage() {
        return [
            'Hello India',
            'Which team is winning Super Bowl? ',
            'Have you checked Ignite UI ?',
            'Take your broken heart and make it to the
art'
        ];
    }
}
```

In the output, you will get many **`MessageComponent`** projected as *figure 13.14*:

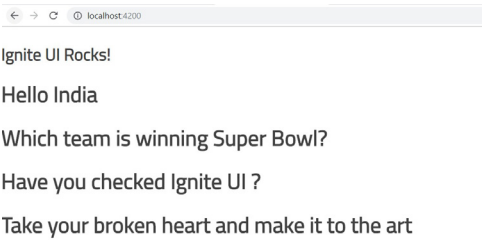


Figure 13.14

Now we have more than one **`ContentChild`**, so we need to access them as **`ContentChildren`** as shown in the *code listing 13.24*:

Code Listing 13.24

```
export class MessageContainerComponent implements
AfterContentInit {
    greetMessage = 'Ignite UI Rocks!';
                    @ContentChildren(MessageComponent)
MessageComponentContentChild:
QueryList<MessageComponent>;
    ngAfterContentInit() {
       console.log(this.MessageComponentContentChild);
    }
}
```

To work with **`ContentChildren`** and **`QueryList`**, you need to do following tasks:

- Import ContentChildren , QueryList , AfterContentInit from @ angular/core
- Make reference of ContentChildren with type QueryList
- Access ContentChildren reference in ngAfterContentInit() life cycle hook

In the output, you will get various reference of **`MessageComponent`** as **`ContentChildren`** as shown in the *figure 13.15*:

```
▼QueryList
  ▶changes: EventEmitter {_isScalar: false, observers: Array(0), closed: false, isStopped
   dirty: false
  ▶first: MessageComponent {message: "Hello India"}
  ▶last: MessageComponent {message: "Take your broken heart and make it to the art"}
   length: 4
  ▼_results: Array(4)
    ▶0: MessageComponent {message: "Hello India"}
    ▶1: MessageComponent {message: "Which team is winning Super Bowl? "}
    ▶2: MessageComponent {message: "Have you checked Ignite UI ?"}
    ▶3: MessageComponent {message: "Take your broken heart and make it to the art"}
     length: 4
```

Figure 13.15

You can query each item in **`ContentChildren`** and modify property as shown in the *code listing 13.25*:

Code Listing 13.25

```
ngAfterContentInit() {
         this.MessageComponentContentChild.forEach((m)
=> m.message = 'Foo');
    }
}
```

In this way, you can work with **`ContentChildren`** in Angular.

Summary

In this chapter, we learnt about projecting content and different ways of component communication which advance the use of components in Angular. In this chapter you learnt about following topics:

- Content Projection
- ViewChild
- ContentChild
- ViewChildren & ContentChildren

——****——

CHAPTER 14

Ignite UI for Angular

Ignite UI for Angular is Angular material based UI components, which help you to create enterprise application faster. There are more than 50 components in Ignite UI for Angular library. However, in this chapter, we will focus on one of most popular Ignite UI for Angular component called `igxGrid`. Ignite UI for Angular Grid is fastest Angular Grid and very easy to work with. In this chapter following topics will be covered:

- Introduction of Ignite UI for Angular
- Ignite UI for Angular Grid
- Add igx-grid to an Angular project
- Add igx-grid component
- Reading a Grid in the Component Class
- Configuring Columns
- Creating Column Templates
- Enable Pagination
- Enable Sorting
- Enable Filtering

Ignite UI for Angular

Ignite UI for Angular is material based Angular components, which help you to write enterprise angular application faster. There are more than 50 components in Ignite UI for Angular library. You can learn more about Ignite UI for Angular here: https://www.infragistics.com/products/ignite-ui-angular

You can use high performance grid, data visualizations charts, calendars, List View, and so on. from Ignite UI for Angular library in your enterprise application to speed up development process and render high performance application. Most popular components in Ignite UI for Angular library are as follows:

- Data Grid
- Tree Grid
- List View
- Combo
- Pie Chart
- Financial Chart
- Category Chart
- Navigation Drawer
- Navbar
- TimePicker
- DatePicker

There are many other components, directives, and services available than above mentioned list in Ignite UI for Angular library.

Ignite UI for Angular Grid

The Ignite UI for Angular Grid is the fastest Angular Grid. It is material-based and can be used as a native component in an Angular application. Ignite UI for Angular comes with rich a set of features, including:

- Virtualization
- Editing
- Paging
- Filtering
- Sorting
- Group By
- Summary
- Column moving , pinning, hiding, template
- Searching
- Selection
- Export to Excel
- Copy Paste from Excel
- Conditional Cell Styling

Learn more about Ignite UI for Angular Grid here: https://www.infragistics.com/products/ignite-ui-angular/angular/components/grid.html

Ignite UI for Angular, which may also be referred to as `<igx-grid>` or `IgxGridComponent` throughout this chapter.

Add igx-grid to an Angular Project

There are three ways to add an **igx-grid** to an Angular project:

1. If starting a new project, use the Ignite UI CLI to scaffold the project. You can use command line options to add the **igx-grid**, including dependency installation.
2. In an existing project, you can use the Ignite UI for Angular Toolbox Extension to add an **igx-grid** in the project. Learn how, in this blog post.
3. You can use **npm** to install Ignite UI for Angular dependencies in your project.

If you have created a project using the Angular CLI, you can use **npm** to install Ignite UI for Angular dependencies to your project. Use **npm** to install dependencies of Ignite UI for Angular and HammerJS.

npm install igniteui-angular

npm install Hammerjs

After successful installation, you need to modify the **angular.json** file as shown in *code listing 14.1*.

Code Listing 14.1

```
    "styles": [
              "src/styles.css",
                "node_modules/igniteui-angular/styles/
igniteui-angular.css"
            ],
            "scripts": ["node_modules/hammerjs/hammer.
min.js" ]
```

In addition, you need to modify **style.css** to import **Material Icons** as Ignite UI for Angular, so you can use them. To do that, open the **style.css** file and, in the top section, add the code as shown in *code listing 14.2*.

Code Listing 14.2

```
@import              url('https://fonts.googleapis.com/
icon?family=Material+Icons');
```

As the last step, you need to import **hammerjs** in **main.ts**. Open the **main.ts** file and add the code shown in *code listing 14.3*.

Code Listing 14.3

```
import 'hammerjs';
```

At this point, you have installed dependencies of Ignite UI for Angular in your project and configured it to use Ignite UI for Angular components.

Before we move ahead, keep in mind that you can use the Ignite UI for Angular Toolbox from Visual Studio Marketplace to right click, add a desired Ignite UI for Angular component, and install all required dependencies in your project.

Add igx-grid Component

To add an. igx-grid or Ignite UI for Angular Grid component, first import the required module in the **`AppModule`** as shown in *code listing 14.4*.

Code Listing 14.4

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

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    IgxGridModule.forRoot()
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Keep in mind that you need to call **`forRoot()`** on **`IgxGridModule`** to register it with the application root module.

To start, we are going to bind local data in the igx-grid. To do that, in **`AppComponent`** class, add a **`getData`** function as shown in *code listing 14.5*.

Code Listing 14.5

```
 getData() {
```

```
    return [
      {
        "name": "Leticia Grisewood",
        "email": "lgrisewoodbn@youtube.com",
        "company": "Innotype",
        "position": "Administrative Assistant III",
        "city": "Aiken",
        "post_code": "29805",
        "state": "SC",
        "country": "United States",
       "created_on": "Sat Mar 31 2018 00:00:00 GMT-0400
(Eastern Daylight Time)",
       "last_activity": "Tue Apr 24 2018 00:00:00 GMT-
0400 (Eastern Daylight Time)",
        "estimated_sales": "2026416",
        "actual_sales": "714549",
        "tags": "demo"
      },
      {
        "name": "Roderic Gwilt",
        "email": "rgwilt6d@ox.ac.uk",
        "company": "Minyx",
        "position": "Developer IV",
        "city": "Akron",
        "post_code": "44393",
        "state": "OH",
        "country": "United States",
       "created_on": "Fri Apr 06 2018 00:00:00 GMT-0400
(Eastern Daylight Time)",
       "last_activity": "Sat Apr 21 2018 00:00:00 GMT-
0400 (Eastern Daylight Time)",
        "estimated_sales": "4575528",
        "actual_sales": "365964",
        "tags": "cold"
      },
      {
        "name": "Marlee Roote",
        "email": "mroote6v@vk.com",
        "company": "Skalith",
```

```
        "position": "Tax Accountant",
        "city": "Albany",
        "post_code": "12210",
        "state": "NY",
        "country": "United States",
       "created_on": "Sun Dec 17 2017 00:00:00 GMT-0500
(Eastern Standard Time)",
       "last_activity": "Mon Jan 01 2018 00:00:00 GMT-
0500 (Eastern Standard Time)",
        "estimated_sales": "120282",
        "actual_sales": "3000442",
        "tags": "Invalid Date"
      },
      {
        "name": "Jaine Schustl",
        "email": "jschustlav@utexas.edu",
        "company": "Gigazoom",
        "position": "Food Chemist",
        "city": "Albany",
        "post_code": "12232",
        "state": "NY",
        "country": "United States",
       "created_on": "Tue Sep 12 2017 00:00:00 GMT-0400
(Eastern Daylight Time)",
       "last_activity": "Thu Sep 28 2017 00:00:00 GMT-
0400 (Eastern Daylight Time)",
        "estimated_sales": "2002221",
        "actual_sales": "2596208",
        "tags": "pro"
      },
      {
        "name": "Kally Foux",
        "email": "kfoux9c@e-recht24.de",
        "company": "Jaxspan",
        "position": "Nurse",
        "city": "Albuquerque",
        "post_code": "87195",
        "state": "NM",
        "country": "United States",
```

```
      "created_on": "Thu Mar 29 2018 00:00:00 GMT-0400
(Eastern Daylight Time)",
       "last_activity": "Fri Mar 30 2018 00:00:00 GMT-
0400 (Eastern Daylight Time)",
        "estimated_sales": "1952070",
        "actual_sales": "1023200",
        "tags": "cold"
      }
    ];
  }
```

`getData()` function returns an array with different contacts. Create a local variable, set as a data source of igx-grid, and assign result of **`getData()`** function in that as shown in the *code listing 14.6*.

Code Listing 14.6

```
  localData: any = [];

  ngOnInit() {
     this.localData = this.getData();
  }
```

Instead of **`ngOnInit()`** life cycle, you can also call **`getData()`** function inside constructor. Putting everything together, the **`AppComponent`** code should look like as shown in *code listing 14.7*.

Code Listing 14.7

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {

  localData: any = [];
  ngOnInit() {
     this.localData = this.getData();
  }
```

```
  getData() {
    return [
      {
        "name": "Leticia Grisewood",
        "email": "lgrisewoodbn@youtube.com",
        "company": "Innotype",
        "position": "Administrative Assistant III",
        "city": "Aiken",
        "post_code": "29805",
        "state": "SC",
        "country": "United States",
       "created_on": "Sat Mar 31 2018 00:00:00 GMT-0400
(Eastern Daylight Time)",
       "last_activity": "Tue Apr 24 2018 00:00:00 GMT-
0400 (Eastern Daylight Time)",
    "estimated_sales": "2026416",
    "actual_sales": "714549",
        "tags": "demo"
      },
      {
        "name": "Roderic Gwilt",
        "email": "rgwilt6d@ox.ac.uk",
        "company": "Minyx",
        "position": "Developer IV",
        "city": "Akron",
        "post_code": "44393",
        "state": "OH",
        "country": "United States",
       "created_on": "Fri Apr 06 2018 00:00:00 GMT-0400
(Eastern Daylight Time)",
       "last_activity": "Sat Apr 21 2018 00:00:00 GMT-
0400 (Eastern Daylight Time)",
        "estimated_sales": "4575528",
        "actual_sales": "365964",
        "tags": "cold"
      },
      {
        "name": "Marlee Roote",
        "email": "mroote6v@vk.com",
```

```
        "company": "Skalith",
        "position": "Tax Accountant",
        "city": "Albany",
        "post_code": "12210",
        "state": "NY",
        "country": "United States",
       "created_on": "Sun Dec 17 2017 00:00:00 GMT-0500
(Eastern Standard Time)",
       "last_activity": "Mon Jan 01 2018 00:00:00 GMT-
0500 (Eastern Standard Time)",
        "estimated_sales": "120282",
        "actual_sales": "3000442",
        "tags": "Invalid Date"
      },
      {
        "name": "Jaine Schustl",
        "email": "jschustlav@utexas.edu",
        "company": "Gigazoom",
        "position": "Food Chemist",
        "city": "Albany",
        "post_code": "12232",
        "state": "NY",
        "country": "United States",
       "created_on": "Tue Sep 12 2017 00:00:00 GMT-0400
(Eastern Daylight Time)",
       "last_activity": "Thu Sep 28 2017 00:00:00 GMT-
0400 (Eastern Daylight Time)",
        "estimated_sales": "2002221",
        "actual_sales": "2596208",
        "tags": "pro"
      },
      {
        "name": "Kally Foux",
        "email": "kfoux9c@e-recht24.de",
        "company": "Jaxspan",
        "position": "Nurse",
        "city": "Albuquerque",
        "post_code": "87195",
        "state": "NM",
```

```
        "country": "United States",
       "created_on": "Thu Mar 29 2018 00:00:00 GMT-0400
(Eastern Daylight Time)",
       "last_activity": "Fri Mar 30 2018 00:00:00 GMT-
0400 (Eastern Daylight Time)",
        "estimated_sales": "1952070",
        "actual_sales": "1023200",
        "tags": "cold"
      }
    ];
  }
}
```

On the **`AppComponet`** template, you can add igx-grid as shown in *code listing 14.8*.

Code Listing 14.8

```
<igx-grid #grid1 id="grid1"
          [data]="localData"
          [autoGenerate]="true">
</igx-grid>
```

Bare minimum properties you need to set:

- Data: determines the DataSource of the igx-grid.
- Autogenerate: determines whether columns will be generated automatically or not. If it is set to false, you need to configure columns manually.

If everything works as expected, you should see a simple igx-grid rendered in the Angular application as shown in the *figure 14.1*.

Figure 14.1

Reading a Grid in the Component Class

There may be requirements to read the igx-grid in the component class.

That can be useful, in using **`IgxGridComponent`** properties and events in the class. To do this, first import **`IgxGridComponent`** as shown in the *code listing 14.9.*

Code Listing 14.9

```
import { IgxGridComponent } from 'igniteui-angular';
```

Next, create **ViewChild** type local variable as shown in *code listing 14.10.*

Code Listing 14.10

```
@ViewChild('grid1', { read: IgxGridComponent })
public grid: IgxGridComponent;
```

`grid1` is a template reference variable name. Now, you have reference of **`IgxGridComponent`** and you can use that in **`ngOnInit()`** or **ngAfterViewInit()** life cycle hook as shown in *code listing 14.11.*

Code Listing 14.11

```
ngOnInit() {
   console.log(this.grid);
}
```

If everything works as expected, you should see an **`IgxGridComponent`** in the browser console as shown *figure 14.2.*

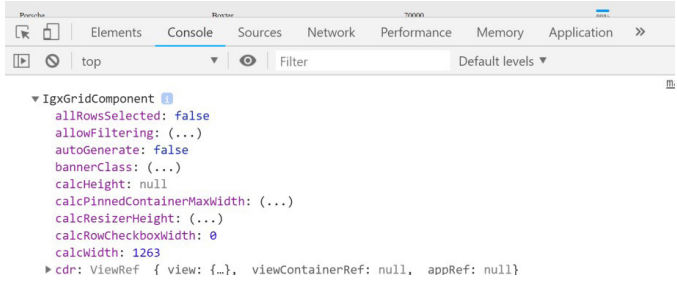


Figure 14.2

Configuring Columns

To configure columns manually, first set the **`AutoGenerate`** value property to false. You need to use **`igx-column`** or **`IgxColumnComponent`** to configure the columns. To do that, modify the igx-grid as shown in *code listing 14.12.*

Code Listing 14.12

```
<igx-grid #grid1 id="grid1"
          [data]="localData"
          [autoGenerate]="false">
           <igx-column field="name" header="Name"></igx-
column>
            <igx-column field="email"  header="Email"></
igx-column>
        <igx-column field="company"  header="Company"></
igx-column>
       <igx-column field="position" header="Position"></
igx-column>
          <igx-column field="city"  header="City"></igx-
column>
           <igx-column field="post_code"  header="Postal
Code"></igx-column>
         <igx-column field="state" header="State"></igx-
column>
        <igx-column field="country"  header="Country"></
igx-column>
         <igx-column field="created_on"  header="Created
On"></igx-column>
          <igx-column field="last_activity" header="Last
Activity"></igx-column>
                    <igx-column  field="estimated_sales"
header="Estimated Sales"></igx-column>
        <igx-column field="actual_sales"  header="Actual
Sales"></igx-column>
          <igx-column field="tags"  header="Tags"></igx-
column>
</igx-grid>
```

Figure

Besides header and field properties, there are many other properties that can be set on **`IgxColumnComponent`**. We will cover that in further sections.

If everything works as expected, you should see a simple igx-grid rendered in Angular application as shown in the *figure 14.3*.

name	email	company	position	city	post_code	state	country	created_on	last_a
Leticia Grise…	lgrisewoodbn…	Innotype	Administrativ…	Aiken	29805	SC	United States	Sat Mar 31 20…	Tue A
Roderic Gwilt	rgwilt6d@ox.…	Minyx	Developer IV	Akron	44393	OH	United States	Fri Apr 06 20…	Sat Ap
Marlee Roote	mroote6v@vk…	Skalith	Tax Accountant	Albany	12210	NY	United States	Sun Dec 17 2…	Mon J
Jaine Schnstl	jschnstlav@ut…	Gigazoom	Food Chemist	Albany	12232	NY	United States	Tue Sep 12 2…	Thu S
Kally Foux	kfoux9c@e-re…	Jaxspan	Nurse	Albuquerque	87195	NM	United States	Thu Mar 29 2…	Fri M

Figure 14.3

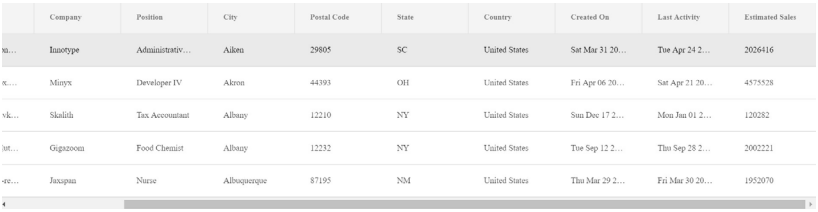
As you notice, now the header is changed. While configuring columns manually, you can opt to exclude certain columns. For example, you can emit the tags, **actual_sales** columns by excluding it in columns configuration as shown in the *code listing 14.13.*

Code Listing 14.13

```
<igx-grid #grid1 id="grid1"
          [data]="localData"
          [autoGenerate]="false">
           <igx-column field="name" header="Name"></igx-
column>
            <igx-column field="email"  header="Email"></
igx-column>
        <igx-column field="company"  header="Company"></
igx-column>
       <igx-column field="position" header="Position"></
igx-column>
          <igx-column field="city"  header="City"></igx-
column>
           <igx-column field="post_code"  header="Postal
Code"></igx-column>
         <igx-column field="state" header="State"></igx-
column>
        <igx-column field="country"  header="Country"></
igx-column>
         <igx-column field="created_on"  header="Created
On"></igx-column>
          <igx-column field="last_activity" header="Last
Activity"></igx-column>
                    <igx-column  field="estimated_sales"
```

```
header="Estimated Sales"></igx-column>
</igx-grid>
```

If everything works as expected, you should see a simple igx-grid rendered in Angular application as shown in the *figure 14.4.*



	Company	Position	City	Postal Code	State	Country	Created On	Last Activity	Estimated Sales
…n…	Innotype	Administrativ…	Aiken	29805	SC	United States	Sat Mar 31 20…	Tue Apr 24 2…	2026416
…x…	Minyx	Developer IV	Akron	44393	OH	United States	Fri Apr 06 20…	Sat Apr 21 20…	4575528
…vk…	Skalith	Tax Accountant	Albany	12210	NY	United States	Sun Dec 17 2…	Mon Jan 01 2…	120282
…ut…	Gigazoom	Food Chemist	Albany	12232	NY	United States	Tue Sep 12 2…	Thu Sep 28 2…	2002221
…re…	Jaxspan	Nurse	Albuquerque	87195	NM	United States	Thu Mar 29 2…	Fri Mar 30 20…	1952070

Figure 14.4

Creating Column Templates

While creating an enterprise level application, you may have to create custom templates for the columns. You need custom templates to display other components or customize a design inside a column. For example, if you need to display a progress bar, a data chart, or an image inside a column, you can do that by creating a custom column template. **`IgxColumnComponent`** supports custom templates.

Let us modify the data source and add a new property rating as shown in *code listing 14.14.*

Code Listing 14.14

```
getData() {
    return [
      {
        "name": "Leticia Grisewood",
        "email": "lgrisewoodbn@youtube.com",
        "company": "Innotype",
        "position": "Administrative Assistant III",
        "city": "Aiken",
        "post_code": "29805",
        "state": "SC",
        "country": "United States",
       "created_on": "Sat Mar 31 2018 00:00:00 GMT-0400
(Eastern Daylight Time)",
```

```
        "last_activity": "Tue Apr 24 2018 00:00:00 GMT-
0400 (Eastern Daylight Time)",
        "estimated_sales": "2026416",
        "actual_sales": "714549",
        "tags": "demo",
        "rating": "7"
      },
      {
        "name": "Roderic Gwilt",
        "email": "rgwilt6d@ox.ac.uk",
        "company": "Minyx",
        "position": "Developer IV",
        "city": "Akron",
        "post_code": "44393",
        "state": "OH",
        "country": "United States",
       "created_on": "Fri Apr 06 2018 00:00:00 GMT-0400
(Eastern Daylight Time)",
        "last_activity": "Sat Apr 21 2018 00:00:00 GMT-
0400 (Eastern Daylight Time)",
        "estimated_sales": "4575528",
        "actual_sales": "365964",
        "tags": "cold",
        "rating": "8"
      },
      {
        "name": "Marlee Roote",
        "email": "mroote6v@vk.com",
        "company": "Skalith",
        "position": "Tax Accountant",
        "city": "Albany",
        "post_code": "12210",
        "state": "NY",
        "country": "United States",
       "created_on": "Sun Dec 17 2017 00:00:00 GMT-0500
(Eastern Standard Time)",
        "last_activity": "Mon Jan 01 2018 00:00:00 GMT-
0500 (Eastern Standard Time)",
        "estimated_sales": "120282",
```

```
        "actual_sales": "3000442",
        "tags": "Invalid Date",
        "rating": "9"
      },
      {
        "name": "Jaine Schustl",
        "email": "jschustlav@utexas.edu",
        "company": "Gigazoom",
        "position": "Food Chemist",
        "city": "Albany",
        "post_code": "12232",
        "state": "NY",
        "country": "United States",
       "created_on": "Tue Sep 12 2017 00:00:00 GMT-0400
(Eastern Daylight Time)",
       "last_activity": "Thu Sep 28 2017 00:00:00 GMT-
0400 (Eastern Daylight Time)",
        "estimated_sales": "2002221",
        "actual_sales": "2596208",
        "tags": "pro",
        "rating": "7"
      },
      {
        "name": "Kally Foux",
        "email": "kfoux9c@e-recht24.de",
        "company": "Jaxspan",
        "position": "Nurse",
        "city": "Albuquerque",
        "post_code": "87195",
        "state": "NM",
        "country": "United States",
       "created_on": "Thu Mar 29 2018 00:00:00 GMT-0400
(Eastern Daylight Time)",
       "last_activity": "Fri Mar 30 2018 00:00:00 GMT-
0400 (Eastern Daylight Time)",
        "estimated_sales": "1952070",
        "actual_sales": "1023200",
        "tags": "cold",
```

```
        "rating": "6.5"
      }
    ];
  }
```

We wish to display ratings in a Linear Progress Bar. Ignite UI for Angular provides a Linear Progress Bar component. To use **`igx-linear-bar`** inside **`IgxColumnComponent`**, you need to create a template for the column, which can be done as shown in *code listing 14.15*.

Code Listing 14.15

```
<igx-column field="rating" header="Ratings">
          <ng-template igxCell let-value>
                    <igx-linear-bar   [value]="value"
[max]="10"></igx-linear-bar>
          </ng-template>
      </igx-column>
```

As you can see, we are using **`<ng-template>`** inside **`<igx-column>.`** Inside the template, **`<igx-linear-bar>`** is used. You can read the column value using the [value] property binding and passing *value* to it. Each of the grid columns can be templated separately. The column expects **`ng-template`** tags decorated with one of the grid module directives.

Keeping everting together, **`igx-grid`** with template column will look like the *code listing 14.16*.

Code Listing 14.16

```
<igx-grid    #grid1    id="grid1"    [data]="localData"
[autoGenerate]="false">
  <igx-column field="name" header="Name"></igx-column>
 <igx-column field="email" header="Email"></igx-column>
   <igx-column field="company" header="Company"></igx-
column>
  <igx-column field="position" header="Position"></igx-
column>
  <igx-column field="city" header="City"></igx-column>
  <igx-column field="post_code" header="Postal Code"></
igx-column>
 <igx-column field="state" header="State"></igx-column>
```

```
   <igx-column field="country" header="Country"></igx-
column>
  <igx-column field="created_on" header="Created On"></
igx-column>
     <igx-column field="last_activity" header="Last
Activity"></igx-column>
  <igx-column field="estimated_sales" header="Estimated
Sales"></igx-column>
     <igx-column field="actual_sales" header="Actual
Sales"></igx-column>
  <igx-column field="tags" header="Tags"></igx-column>
  <igx-column field="rating" header="Ratings">
    <ng-template igxCell let-value>
     <igx-linear-bar [value]="value" [max]="10"></igx-
linear-bar>
    </ng-template>
  </igx-column>
</igx-grid>
```

If everything works as expected, you should see a simple igx-grid rendered in Angular application as shown in the *figure 14.5*.

	Postal Code	State	Country	Created On	Last Activity	Estimated Sales	Actual Sales	Tags	Ratings
	29805	SC	United States	Sat Mar 31 20…	Tue Apr 24 2…	2026416	714549	demo	70%
	44393	OH	United States	Fri Apr 06 20…	Sat Apr 21 20…	4575528	365964	cold	80%
	12210	NY	United States	Sun Dec 17 2…	Mon Jan 01 2…	120282	3000442	Invalid Date	90%
	12232	NY	United States	Tue Sep 12 2…	Thu Sep 28 2…	2002221	2596208	pro	70%
te	87195	NM	United States	Thu Mar 29 2…	Fri Mar 30 20…	1952070	1023200	cold	65%

Figure 14.5

Enable Pagination

Configuring the Ignite UI for Angular Grid for Pagination is very simple. To enable pagination, you just have to do property binding of **paging** and **`perPage`** properties of the **`igxGridComponent`** as shown in *code listing 14.17*.

Code Listing 14.17

```
<igx-grid #grid1 id="grid1" [data]="localData"
```

```
          [autoGenerate]="false"
          [paging]="true"
          [perPage]="5">
  <igx-column field="name" header="Name"></igx-column>
  <igx-column field="email" header="Email"></igx-column>
   <igx-column field="company" header="Company"></igx-
column>
  <igx-column field="position" header="Position"></igx-
column>
  <igx-column field="city" header="City"></igx-column>
  <igx-column field="post_code" header="Postal Code"></
igx-column>
  <igx-column field="state" header="State"></igx-column>
   <igx-column field="country" header="Country"></igx-
column>
  <igx-column field="created_on" header="Created On"></
igx-column>
     <igx-column field="last_activity" header="Last
Activity"></igx-column>
  <igx-column field="estimated_sales" header="Estimated
Sales"></igx-column>
     <igx-column field="actual_sales" header="Actual
Sales"></igx-column>
  <igx-column field="tags" header="Tags"></igx-column>
  <igx-column field="rating" header="Ratings">
    <ng-template igxCell let-value>
     <igx-linear-bar [value]="value" [max]="10"></igx-
linear-bar>
    </ng-template>
  </igx-column>
</igx-grid>
```

As you might have noticed in previous code snippets:

1. Set paging property to true.
2. To set per page rows, set a number to **perPage** property.

Also, to display pagination, let us update the data source with more rows. If everything works as expected, you should see a simple igx-grid rendered in the Angular application as shown in the *figure 14.6.*

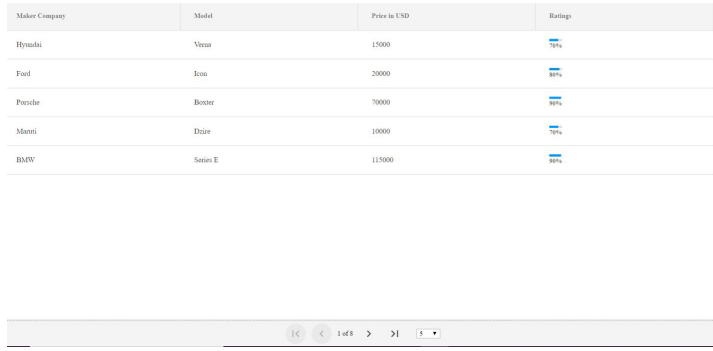


Figure 14.6

Enable Sorting

In the Ignite UI for Angular Grid, **sorting** is enabled at the column level. Thus, you can have a mix of sort enabled and disabled columns. For the igx-grid, you can configure sorting at the igx-column level as shown in the *code listing 14.18.*

Code Listing 14.18

```
<igx-grid #grid1 id="grid1"
          [data]="localData"
           [autoGenerate]="false">
  <igx-column field="make" header="Maker Company"></igx-
column>
        <igx-column    field="model"    sortable='true'
header="Model"></igx-column>
 <igx-column field="price" sortable='true' header="Price
in USD"></igx-column>
       <igx-column    field="rating"    sortable='true'
header="Ratings">
    <ng-template igxCell let-value>
     <igx-linear-bar [value]="value" [max]="10"></igx-
linear-bar>
    </ng-template>
  </igx-column>
</igx-grid>
```

You have to set value of sortable to true, to configure columns for sorting.

As you can see, we have a combination of columns configured for sorting and not sorting.

If everything works as expected, you should see a simple igx-grid rendered in the Angular application as shown in the *figure 14.7*.

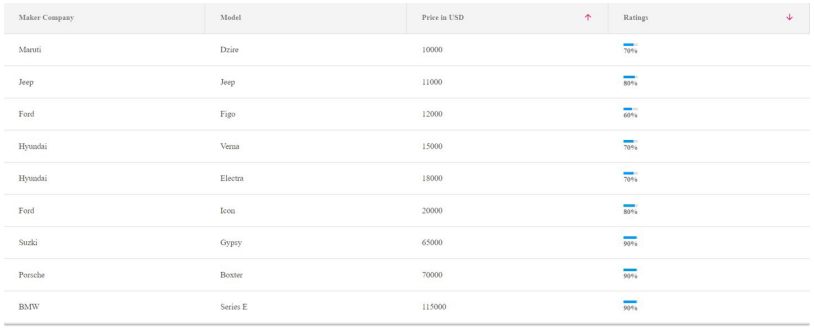


Figure 14.7

Some points you should keep in mind about sorting:

1. Sorting does not change the underlying data source
2. You can set the initial sorting state using the **`sortingExpressions`** property of the igx-grid
3. You can perform Remote Sorting as well
4. You can also use **`IgxGridComponent`** API to perform sorting.

Enabling Filtering

In the Ignite UI for Angular Grid, you can enable filtering by setting the **`allowFiltering`** property to true on **`<igx-grid>`**. You can set it following the *code listing 14.19*.

Code Listing 14.19

```
<igx-grid #grid1 id="grid1" [data]="localData"
          [autoGenerate]="false"
          [allowFiltering]="true">
  <igx-column field="make" header="Maker Company"></igx-
column>
  <igx-column field="model" header="Model"></igx-column>
  <igx-column field="price" header="Price in USD"></igx-
column>
```

```
  <igx-column field="rating" header="Ratings">
    <ng-template igxCell let-value>
     <igx-linear-bar [value]="value" [max]="10"></igx-
linear-bar>
    </ng-template>
  </igx-column>
</igx-grid>
```

If everything works as expected, you should see a simple igx-grid rendered in the Angular application as shown in the *figure 14.8.*

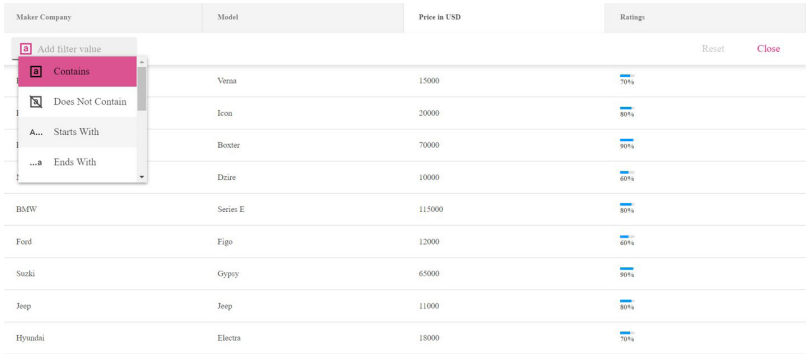


Figure 14.8

You can disable filtering at the column level by switching the **`filterable`** input to false. You can disable filtering for the Ratings column as shown in *code listing 14.20.*

Code Listing 14.20

```
<igx-grid #grid1 id="grid1" [data]="localData"
          [autoGenerate]="false"
          [allowFiltering]="true">
  <igx-column field="make" header="Maker Company"></igx-
column>
  <igx-column field="model" header="Model"></igx-column>
  <igx-column field="price" header="Price in USD"></igx-
column>
      <igx-column   [filterable]="false"   field="rating"
header="Ratings">
    <ng-template igxCell let-value>
      <igx-linear-bar [value]="value" [max]="10"></igx-
```

```
linear-bar>
    </ng-template>
  </igx-column>
</igx-grid>
```

You should see a simple igx-grid rendered in the Angular application as shown in the *figure 14.9*.

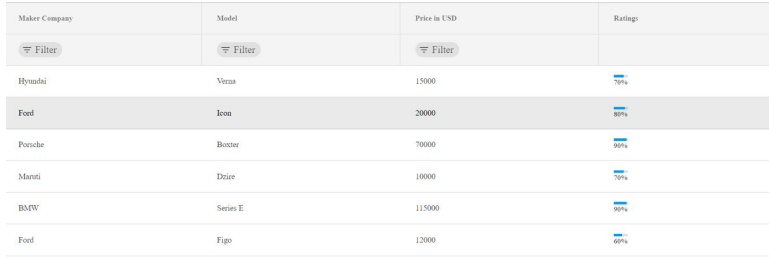


Figure 14.9

Summary

Now you have all you need to get started with the high performance igx-grid in an Angular application. In this tutorial, we covered configuring Ignite UI for Angular in a project, creating a grid, configuring columns, pagination, sorting, filtering.

In addition to these features, the grid also offers:

- Virtualization
- Editing
- Group By
- Summaries
- Column Moving
- Column Pinning
- Multi Column Headers

Configuring these features is also very easy. In this chapter, you learnt about the following:

- Introduction of Ignite UI for Angular
- Ignite UI for Angular Grid
- Add igx-grid to an Angular project
- Add igx-grid component

- Reading a Grid in the Component Class
- Configuring Columns
- Creating Column Templates
- Enable Pagination
- Enable Sorting
- Enable Filtering

——****——