# Features of Angular

Angular is loaded with Power-packed Features. Some of the features are listed below

Two-Way [Data Binding](https://www.tektutorialshub.com/angular/angular-data-binding/)  
This is the coolest feature of the Angular. Data binding is automatic and fast. changes made in the View is automatically updated in the component class and vice versa

Powerful [Routing](https://www.tektutorialshub.com/angular/angular-routing-navigation/) Support  
The Angular Powerful routing engine loads the page asynchronously on the same page enabling us to create a Single Page Applications.

Expressive HTML  
Angular enables us to use programming constructs like if conditions, for loops, etc to render and control how the HTML pages.

Modular by Design  
Angular follows the modular design. You can create [Angular modules](https://www.tektutorialshub.com/angular/angular-modules/) to better organize and manage our codebase

Built-in Back End Support  
Angular has built-in support to communicate with the back-end servers and execute any business logic or retrieve data

Active Community  
Angular is Supported by google and has a very good active community of supporters. This makes a lot of difference as your queries are quickly resolved.

Angular has changed massively from the AngularJS. Angular completely redesigned from scratch. There are many concepts of angularJS that have changed in Angular.

Key differences between AngularJs & Angular

Support for ES6

Angular is completely written in Typescript and meets the ECMAScript 6 specification. This means that it has support for ES6 Modules, Class frameworks, etc.

Components are new controllers

In AngularJS we had Controllers. In Angular Controllers are replaced with [Angular Components](https://www.tektutorialshub.com/angular/angular-component/). The controllers and view in AngularJS is defined as follows

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | //View  <body ng-controller=’appController’>      <h1>vm.message<h1>  </body>    //Controller angular.module(‘app’).controller(‘appController’,appcontroller) {      message=’Hello Angular’;  } |

In Angular, we are using Components. The simple component is shown below written using Typescript.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import { Component } from '@angular/core';    @Component({      selector: 'app',      template: '<h1>{{message}} </h1>'  })  export class AppComponent  {        message: string=’Hello Angular’;  } |

In Angular, a component represents a UI element. You can have many such components on a single web page. Each component is independent of each other and manages a section of the page. The components can have child components and parent components.

Directives

The AngularJS had a lot of directives. Some of the most used directives are ng-repeat & ng-if

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <ul>      <li ng-repeat =customer in vm.customers>          {{customer.name}}      </li>  </ul>  <div ng-if=”vm.isVIP”>     <h3> VIP Customer </h3> </div> |

The Angular also has [directives,](https://www.tektutorialshub.com/angular/angular-directives/) but with a different syntax. It has a \* before the directive name indicating it as a structural directive

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <ul>      <li \*ngFor =#customer of customers>          {{customer.name}}      </li>  </ul>  <div \*ngIf=”vm.isVIP”>      <h3> VIP Customer </h3>  </div> |

The style directives like ng-style, ng-src & ng-href are all gone. These are now replaced by property binding HTML elements to the class properties

The creation of Custom Directives is vastly simplified in angular as shown in the example below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Directive({     selector: '[MyDirective]'  })  class MyDirective { } |

Data Bindings

The powerful angular data bindings stay the same,  with minor syntax changes.

Interpolation

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //AngularJS  <h3> {{vm.customer.Name}}</h3>    //Angular  <h3> {{customer.Name}}</h3> |

Note that we used controller alias VM to specify the controller instance in AngularJS. In Angular, the context is implied.

One way Binding

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //AngularJS  <h3> ng-bind=vm.customer.name></h3>    //Angular  <h3 [innerText]=”customer.name” ></h3> |

The Angular can bind to any property of the HTML element.

Event Binding

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //AngularJS  <button ng-click=”vm.save()”>Save<button>    //Angular  <button (click)=”save()”>Save<button> |

The AngularJS uses the ngClick directive to bind to the event. In Angular ngClick Directive is removed. You can bind directly to the DOM events

Two- way binding

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //AngularJS  <input ng-model=”vm.customer.name”>    //Angular  <input [(ng-model)]=”customer.name”> |

$scopes are out

Angular is not using $scope anymore to glue view and controller.

AngularJS used to run a dirty checking on the scope objects to see if any changes occurred. Then it triggers the watchers. And then it used to re-running the dirty checking again.

The Angular is using zone.js to detect changes. Zone.js apply patches on all the global asynchronous operations like click event, timer events, HTTP requests, etc. It then intimates the Angular, whenever the changes occur in Angular Application. The Angular then runs the change detection for the entire application

Filters are renamed to Pipes

In AngularJS, we used Filters and as shown below

|  |  |
| --- | --- |
| 1  2  3 | <td>{{vn.customer.name | uppercase}}</td> |

Angular uses the same syntax but names them as [pipes](https://www.tektutorialshub.com/angular/angular-pipes/)

|  |  |
| --- | --- |
| 1  2  3 | <td>{{customer.name | uppercase}}</td> |

Platform-specific Bootstrap

In AngularJS we used the ng-app directive in our HTML, then the Angular would bootstrap and attach itself the ng-app

|  |  |
| --- | --- |
| 1  2  3 | <body ng-app=’app’> </html> |

The bootstrapping in Angular is done through code. The bootstrapping of Angular is not simple as that of AngularJS. The sample code below shows how Angular application bootstraps the AppModule using platformBrowserDynamic Module

|  |  |
| --- | --- |
| 1  2  3  4 | import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  import {AppModule } from './app.module'; platformBrowserDynamic().bootstrapModule(AppModule); |

The Bootstrap is also Platform-specific in Angular. You can have different bootstrapper for mobile & Web application.

You can read about [Bootstrapping Angular Application](https://www.tektutorialshub.com/angular/angular-bootstrapping-application/)

Services

The AngularJS had Services, Factories, Providers, Constants and values, which used to create reusable code. These are then injected into Controllers so that it can use it

[The Angular all the above is merged into a Service](https://www.tektutorialshub.com/angular/angular-services/) class.

Mobile Support

AngularJS was not built with mobile support in mind. Angular designed with mobile development in mind.

Angular Modules

As seen from the above diagram the Angular Application consists of several building blocks like Components, Services, Directives. We create more such blocks as the application grows. The Angular provides a nice way to organize these building blocks using the concept called [Angular Modules](https://www.tektutorialshub.com/angular/angular-modules/).

We will create components, services, and directivesand put them inside the [Angular Modules](https://www.tektutorialshub.com/angular/angular-modules/). We use the special directive to create the Modules. The [Angular Module](https://www.tektutorialshub.com/angular/angular-modules/) also called as [ngModule](https://www.tektutorialshub.com/angular/angular-modules/)

Use Angular Module ( or ngModule) to organize of the Angular code within a bigger Angular Application. The Application is made up of several Angular Modules acting together. Each Module implementing a specific feature or functionality of the application.

Javascript Modules (ES2015)

Do not confuse [Angular Modules](https://www.tektutorialshub.com/angular/angular-modules/) with JavaScript Modules. The Angular Modules are specific to Angular. The JavaScript Modules are part of the JavaScript language specification. The ES2015 specifications set the standard for defining modules.

In JavaScript as per the ES2015 specification, each file is a module. There is one module per file and one file per module.

These Modules define the scope of the variables, functions, and classes defined inside the module. These variables are always local to the module and are not visible outside the module

The modules without public methods or properties are useless. The modules help us to create public members by using the **Export statement**. The other modules can use these public members using the **Import statement**.

Angular uses the JavaScript modules via the Typescript.

We build Component, Services, directives using the JavaScript Module. Each Component is a JavaScript Module. Each Service is a JavaScript Module. And then we combine them together using the [Angular Module](https://www.tektutorialshub.com/angular/angular-modules/).

Anatomy of Angular JavaScript Module

In Angular Application, we mainly build [**Components**](https://www.tektutorialshub.com/angular/angular-component/), [Services](https://www.tektutorialshub.com/angular/angular-services/) & [Directives](https://www.tektutorialshub.com/angular/angular-directives/). Angular provides a very consistent way to define these building blocks. These are defined inside a **JavaScript Module** and follows the pattern shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import { Component } from '@angular/core';    @Component({  })  export class AppComponent  {      SomeObject : any;      Title : string ="Hello Angular";        getSomeObject(): any {         return this.SomeObject;      }  } |

The above example has three Parts, An import statement at the beginning. A class (AppComponent) at the bottom, which we define with an export statement.We decorate the class with a decorator @Component immediately above the class.

Import Statement

The import statement tells Angular where to find the external functions that we are using in our module. All the external dependencies like third party libraries, our own modules or modules from Angular must be imported. An Import statement is part of the ES2015 specifications. It is similar to the Import statement of Java or Using statement of C#

You can import only the exported members from the other modules.

The class

The Class contains the logic of the application. It can contain methods & properties just like c# or java classes. The class must be defined with the export keyword if you want to use the class in another module.

The class Decorator

AppComponent is just a class. There is nothing Angular about it. It is the decorator, which tells angular how to treat the class.

For Example, @Component decorator tells the Angular that the Class is a Component. Similarly, a @Directive tells the Angular that the class is a Directive. Angular currently has following class decorators

@Component

@Directive

@Injectable

@NgModule

@Pipe

Building Blocks of Angular Application

Looking at task application in the previous section, you can identify the seven main building blocks of an Angular Application.

Component

Templates

Metadata

Data Binding

Directives

Services

Dependency Injection

Component

The [Angular Component](https://www.tektutorialshub.com/angular/angular-component/) is a class, which we decorate by @Component class decorator

The Component controls the part of our user interface (or view). The Task List application listed above has three components. The TaskComponent displays the list of Tasks The TaskAddComponent helps us to create new tasks. The rootComponent is the Parent component of these components and it only displays the name of the application.

The component, that we had created in the Create Your First Angular Application as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'GettingStarted';  } |

The Component has four important parts

Import Statement

Class

Template

Metadata

The Import statement imports the dependencies required by this component. The class contains the application logic. It is decorated by the **@Component** class decorator.

Template

The Component needs a View to display. The Template defines that View.

The Template is just a subset of HTML, that tells Angular how to display the view. It is a normal HTML page using tags like h1, h2, etc. It also uses the Angular-specific markup like {} (for interpolation), [] (for Property binding) etc.

Metadata

Metadata tells angular how to Process the class.

We attach the Metadata to a class using a class decorator. When we attach @Component class decorator to the class, then it becomes Component class.

The class decorator uses the configuration object, which provides the Angular information it needs to create the component. For Example, @Component directives come with configuration options like selector, templateURL (or template), directives, etc

Data Binding

Angular uses the [Data Binding](https://www.tektutorialshub.com/angular/angular-data-binding/) to get the data from the Component to our View (Template). This is done using the special HTML Angular-specific markup known as the Template Syntax

The Angular supports four types of Data binding

[Interpolation](https://www.tektutorialshub.com/angular/angular-data-binding/)  
Data is bind from component to View

[Property Binding](https://www.tektutorialshub.com/angular/angular-data-binding/)  
Data is bind from component to the property of an HTML control in the view like

[Event Binding](https://www.tektutorialshub.com/angular/angular-data-binding/)  
The DOM Events are bind from View to a method in the Component

[Two-way Binding](https://www.tektutorialshub.com/angular/angular-data-binding/)/Model Binding  
The data flow in both directions from view to component or from component to view

Directive

The [Directives](https://www.tektutorialshub.com/angular/angular-directives/) help us to manipulate the view.

A directive is a class, which we create using the @Directive class decorator. It contains the metadata and logic to manipulate the DOM

The Views are created by the Angular by using the Templates defined in the Components. These templates are dynamic and are transformed as per the Directives.

Angular supports two types of directives. One is [**structural directives**](https://www.tektutorialshub.com/angular/angular-directives/#Structural-Directives) which change the structure of the View and the other one is [**attribute directive**](https://www.tektutorialshub.com/angular/angular-directives/#Attribute-Directives), which alters the style of our view.

Services

The [Services](https://www.tektutorialshub.com/angular/angular-services/) provide service to the Components or to the other Services.

Angular does not have any specific definition for Services. You just create a class, export a method, that does some specific task and it becomes a service. You do not have to do anything else.

For Example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | export class MyLogger {      AddTolog(msg: any)      {          console.log(msg);      }  } |

And in any of our controllers, we can just invoke it using

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | log :MyLogger = new MyLogger();    constructor() {      this.log.AddTolog("Component Created");  } |

These are plain Javascript Modules. There is nothing Angular about these Services.

What Angular does is to make these services are available to the components using what is known as dependency injection.

Dependency Injection

[Dependency injection](https://www.tektutorialshub.com/angular/angular-dependency-injection/) is a method by which, a new instance of a service is injected into the component, which requires it.

The dependencies injection is mostly used to inject services into components or to other services.

The Angular does this using the [injector](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/). When a component is created, Angular looks at the Component Metadata for the Services, on which the Component is required. The injector then creates the instance of the Service and injects it into the component using its constructor.

If the service is already created, then the injector does not create the service but uses it. The Service needs to tell the Angular that it can be injected into any components, which requires it. This is done by using the @Injectable call decorator

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Injectable()  export class MyLogger {      AddTolog(msg: any)      {          console.log(msg);      }  } |

Angular Modules

As seen from the above diagram the Angular Application consists of several building blocks like Components, Services, Directives. We create more such blocks as the application grows. The Angular provides a nice way to organize these building blocks using the concept called Angular Modules.

We will create components, services, and directivesand put them inside the Angular Modules. We use the special directive to create the Modules. The Angular Module also called as ngModule

Use Angular Module ( or ngModule) to organize of the Angular code within a bigger Angular Application. The Application is made up of several Angular Modules acting together. Each Module implementing a specific feature or functionality of the application.

Javascript Modules (ES2015)

Do not confuse Angular Modules with JavaScript Modules. The Angular Modules are specific to Angular. The JavaScript Modules are part of the JavaScript language specification. The ES2015 specifications set the standard for defining modules.

In JavaScript as per the ES2015 specification, each file is a module. There is one module per file and one file per module.

These Modules define the scope of the variables, functions, and classes defined inside the module. These variables are always local to the module and are not visible outside the module

The modules without public methods or properties are useless. The modules help us to create public members by using the Export statement. The other modules can use these public members using the Import statement.

Angular uses the JavaScript modules via the Typescript.

We build Component, Services, directives using the JavaScript Module. Each Component is a JavaScript Module. Each Service is a JavaScript Module. And then we combine them together using the Angular Module.

Anatomy of Angular JavaScript Module

In Angular Application, we mainly build Components, Services & Directives. Angular provides a very consistent way to define these building blocks. These are defined inside a JavaScript Module and follows the pattern shown below

@Component({

})

export class AppComponent

{

SomeObject : any;

Title : string ="Hello Angular";

getSomeObject(): any {

return this.SomeObject;

}

}

The above example has three Parts, An import statement at the beginning. A class (AppComponent) at the bottom, which we define with an export statement. We decorate the class with a decorator @Component immediately above the class.

Import Statement

The import statement tells Angular where to find the external functions that we are using in our module. All the external dependencies like third party libraries, our own modules or modules from Angular must be imported. An Import statement is part of the ES2015 specifications. It is similar to the Import statement of Java or Using statement of C#

You can import only the exported members from the other modules.

The class

The Class contains the logic of the application. It can contain methods & properties just like c# or java classes. The class must be defined with the export keyword if you want to use the class in another module.

The class Decorator

AppComponent is just a class. There is nothing Angular about it. It is the decorator, which tells angular how to treat the class.

For Example, @Component decorator tells the Angular that the Class is a Component. Similarly, a @Directive tells the Angular that the class is a Directive. Angular currently has following class decorators

@Component

@Directive

@Injectable

@NgModule

@Pipe

Building Blocks of Angular Application

Looking at task application in the previous section, you can identify the seven main building blocks of an Angular Application.

Component

Templates

Metadata

Data Binding

Directives

Services

Dependency Injection

Component

The Angular Component is a class, which we decorate by @Component class decorator

The Component controls the part of our user interface (or view). The Task List application listed above has three components. The TaskComponent displays the list of Tasks The TaskAddComponent helps us to create new tasks. The rootComponent is the Parent component of these components and it only displays the name of the application.

The component, that we had created in the Create Your First Angular Application as shown below

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

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

title = 'GettingStarted';

}

The Component has four important parts

Import Statement

Class

Template

Metadata

The Import statement imports the dependencies required by this component. The class contains the application logic. It is decorated by the @Component class decorator.

Template

The Component needs a View to display. The Template defines that View.

The Template is just a subset of HTML, that tells Angular how to display the view. It is a normal HTML page using tags like h1, h2, etc. It also uses the Angular-specific markup like {} (for interpolation), [] (for Property binding) etc.

Metadata

Metadata tells angular how to Process the class.

We attach the Metadata to a class using a class decorator. When we attach @Component class decorator to the class, then it becomes Component class.

The class decorator uses the configuration object, which provides the Angular information it needs to create the component. For Example, @Component directives come with configuration options like selector, templateURL (or template), directives, etc

Data Binding

Angular uses the Data Binding to get the data from the Component to our View (Template). This is done using the special HTML Angular-specific markup known as the Template Syntax

The Angular supports four types of Data binding

Interpolation

Data is bind from component to View

Property Binding

Data is bind from component to the property of an HTML control in the view like

Event Binding

The DOM Events are bind from View to a method in the Component

Two-way Binding/Model Binding

The data flow in both directions from view to component or from component to view

Directive

The Directives help us to manipulate the view.

A directive is a class, which we create using the @Directive class decorator. It contains the metadata and logic to manipulate the DOM

The Views are created by the Angular by using the Templates defined in the Components. These templates are dynamic and are transformed as per the Directives.

Angular supports two types of directives. One is structural directives which change the structure of the View and the other one is attribute directive, which alters the style of our view.

Services

The Services provide service to the Components or to the other Services.

Angular does not have any specific definition for Services. You just create a class, export a method, that does some specific task and it becomes a service. You do not have to do anything else.

For Example.

export class MyLogger {

AddTolog(msg: any)

{

console.log(msg);

}

}

And in any of our controllers, we can just invoke it using

log :MyLogger = new MyLogger();

constructor() {

this.log.AddTolog("Component Created");

}

These are plain Javascript Modules. There is nothing Angular about these Services.

What Angular does is to make these services are available to the components using what is known as dependency injection.

Dependency Injection

Dependency injection is a method by which, a new instance of a service is injected into the component, which requires it.

The dependencies injection is mostly used to inject services into components or to other services.

The Angular does this using the injector. When a component is created, Angular looks at the Component Metadata for the Services, on which the Component is required. The injector then creates the instance of the Service and injects it into the component using its constructor.

If the service is already created, then the injector does not create the service but uses it. The Service needs to tell the Angular that it can be injected into any components, which requires it. This is done by using the @Injectable call decorator

@Injectable()

export class MyLogger {

AddTolog(msg: any)

{

console.log(msg);

}

}

# Bootstrapping in Angular

Bootstrapping is a technique of initializing or loading our Angular application.

what happens at each stage and how our AppComponent gets loaded and displays “app works!”. The Angular takes the following steps to load our first view.

Index.html loads

Angular, Third-party libraries & Application loads

Main.ts the application entry point

Root Module

Root Component

Template

Index.html Loads First

Web apps need a starting point. Index.html is usually the first page to load. Let us open the file and find out what it contains. You will find it under the src folder.

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>GettingStarted</title>

<base href="/">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="icon" type="image/x-icon" href="favicon.ico">

</head>

<body>

<app-root></app-root>

</body>

</html>

There are no javascript files in the index.html. Neither you can see a stylesheet file. The body of the files has the following HTML tag.

<app-root></app-root>

Building Application

To run our application, we use the [Angular CLI](https://www.tektutorialshub.com/angular/angular-cli-tutorial/) command ng serve or NPM command npm start (npm start command actually translates into ng serve.)

ng serve does build our application but does not save the compiled application to the disk. It saves it in memory and starts the development server.

We use ng build to build our app. Open the command prompt and run the command. This will build and copy the output files to the dist folder

|  |  |
| --- | --- |
| 1  2  3 | ng build |

Use ng build --prod to build and distribute the app for production. For testing/debugging use ng build. The production build optimizes, minimize and uglify the code.

Now open the dist and open the index.html.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | <!doctype html>  <html lang="en">  <head>    <meta charset="utf-8">    <title>GettingStarted</title>    <base href="/">    <meta name="viewport" content="width=device-width, initial-scale=1">    <link rel="icon" type="image/x-icon" href="favicon.ico">  </head>  <body>    <app-root></app-root>      <script src="runtime-es2015.js" type="module"></script>    <script src="runtime-es5.js" nomodule defer></script>    <script src="polyfills-es5.js" nomodule defer></script>    <script src="polyfills-es2015.js" type="module"></script>    <script src="styles-es2015.js" type="module"></script>    <script src="styles-es5.js" nomodule defer></script>    <script src="vendor-es2015.js" type="module"></script>    <script src="vendor-es5.js" nomodule defer></script>    <script src="main-es2015.js" type="module"></script>    <script src="main-es5.js" nomodule defer></script></body>  </html> |

You can see that the compiler included five script files. They are runtime, polyfills, styles, vendor, & main. All these files have two versions one is es5 & the other one es2015

Since the Angular 7**,**we have new feature called **conditional polyfill loading**. Now Angular builds two script files, one for es2015 & another for es5. The es2015 (es6) is for modern browser and es5 is older browsers, which do not support the new features of es2015.  
  
Note the nomodule attribute, which tells the modern browser to ignore the script and do not load it. Hence es5 scripts are not loaded in the modern browsers

runtime.js: Webpack runtime file  
polyfills.js – Polyfill scripts for supporting the variety of the latest modern browsers  
styles.js – This file contains the global style rules bundled as javascript file.  
vendor.js – contains the scripts from the Angular core library and any other 3rd party library.  
main.js – code of the application.

The Angular Version 2 generated only three script files ( inline.js, styles.bundle.js & main.bundle.js).

These files are added by the Webpack module loader.

What is Webpack?

Webpack is a bundler. it scans our application looking for javascript files and merges them into one ( or more) big file. Webpack has the ability to bundle any kind of file like JavaScript, CSS, SASS, LESS, images, HTML, & fonts, etc.

The [Angular CLI](https://www.tektutorialshub.com/angular/angular-cli-tutorial/) uses Webpack as a module bundler. Webpack needs a lot of configuration options to work correctly. The Angular CLI sets up all these configuration options behind the scene.

The Webpack traverses through our application looking for javascript and other files and merges all of them into one or more bundles. In our example application, it has created five files.

Application Loads

So when index.html is loaded, the Angular core libraries, third-party libraries are loaded. Now the angular needs to locate the entry point.

Application Entry point

The entry point of our application is main.ts. You will find it under the src folder.

angular.json

The Angular finds out the entry point from the configuration file angular.json. This file is located in the root folder of the project. The relevant part of the angular.json is shown below

The angular-cli.json was the configuration file in Angular 5 and before. It is now angular.json since the version Angular 6.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | {    "$schema": "./node\_modules/@angular/cli/lib/config/schema.json",    "version": 1,    "newProjectRoot": "projects",    "projects": {      "GettingStarted": {        "projectType": "application",        "schematics": {},        "root": "",        "sourceRoot": "src",        "prefix": "app",        "architect": {          "build": {            "builder": "@angular-devkit/build-angular:browser",            "options": {              "outputPath": "dist/GettingStarted",              "index": "src/index.html",              "main": "src/main.ts",                        <====              "polyfills": "src/polyfills.ts",              "tsConfig": "tsconfig.app.json",              "aot": false,              "assets": [                "src/favicon.ico",                "src/assets"              ],              "styles": [                "src/styles.css"              ],              "scripts": []            }, |

The main entry under the node projects -> GettingStarted -> architect -> build -> options points towards the src/main.ts. This file is the entry point of our application.

main.ts Application entry point

The main.ts file is as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import { enableProdMode } from '@angular/core';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';    import { AppModule } from './app/app.module';  import { environment } from './environments/environment';    if (environment.production) {    enableProdMode();  }    platformBrowserDynamic().bootstrapModule(AppModule)    .catch(err => console.error(err)); |

Let us look at the relevant code in detail.

|  |  |
| --- | --- |
| 1  2  3 | import { platformBrowserDynamic } from '@angular/platform-browser-dynamic'; |

This line imports the module platformBrowserDynamic from the library@angular/platform-browser-dynamic.

What is platformBrowserDynamic

platformBrowserDynamic is the module, which is responsible for loading the Angular application in the desktop browser.

The Angular Applications can be bootstrapped in many ways and in many platforms. For example, we can load our application in a Desktop Browser or in a mobile device with Ionic or NativeScript.

If you are using the nativescript, then you will be using platformNativeScriptDynamic from nativescript-angular/platform library and will be calling platformNativeScriptDynamic().bootstrapModule(AppModule). Read more about [Angular Nativescript bootstrap process from here](https://docs.nativescript.org/404.html?path=core-concepts/angular-bootstrap)

|  |  |
| --- | --- |
| 1  2  3 | import { AppModule } from './app/app.module'; |

The above line imports AppModule. The AppModule is the Root Module of the app. The Angular applications are organized as modules. Every application built in Angular must have at least one module. The module, which is loaded first when the application is loaded is called a root module.

|  |  |
| --- | --- |
| 1  2  3  4 | platformBrowserDynamic().bootstrapModule(AppModule)    .catch(err => console.error(err)); |

TheplatformBrowserDynamic loads the root module by invoking the bootstrapModule and giving it the reference to our Root module i.e AppModule

Root Module

The angular bootstrapper loads our root module AppModule. The AppModule is located under the folder src/app. The code of our Root module is shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';    import { AppRoutingModule } from './app-routing.module';  import { AppComponent } from './app.component';    @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule,      AppRoutingModule    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

The root module must have at least one root component. The root component is loaded, when the module is loaded by the Angular.

In our example, AppComponent is our root component. Hence we import it.

|  |  |
| --- | --- |
| 1  2  3 | import { AppComponent } from './app.component'; |

We use @NgModule class decorator to define a [Angular Module](https://www.tektutorialshub.com/angular/angular-modules/) and provide metadata about the Modules.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule,      AppRoutingModule    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

The [@NgModule](https://www.tektutorialshub.com/angular/angular-modules/#ngmodule) has several metadata properties.

imports

We need to list all the external modules required including other Angular modules, that is used by this Angular Module

Declarations

The Declarations array contains the list of components, directives, & pipes that belong to this Angular Module. We have only one component in our application AppComponent.

Providers

The [Providers](https://www.tektutorialshub.com/angular/angular-providers/) array, is where we register the services we create. The [Angular Dependency injection](https://www.tektutorialshub.com/angular/angular-dependency-injection/) framework injects these [services](https://www.tektutorialshub.com/angular/angular-services/) in components, directives. pipes and other services.

Bootstrap

The component that angular should load, when this Angular Module loads. The component must be part of this module. We want AppComponent load when AppModule loads, hence we list it here.

The Angular reads the bootstrap metadata and loads the AppComponent

Component

Finally, we arrive at AppComponent, which is the root [component](https://www.tektutorialshub.com/angular/angular-component/) of the AppModule. The code of our AppComponent is shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'GettingStarted';  } |

The Class AppComponent is decorated with @Component Class Decorator.

The @Component class decorator provides the metadata about the class to the Angular. It has 3 properties in the above code. Selector, templateURL & styleUrls

templateURL

This property contains an HTML template, which is going to be displayed in the browser. The template file is app.component.html

Selector

This property specifies the CSS Selector, where our template will be inserted into the HTML. The CSS Selector in our code is app-root

Template

The AppComponent defines the template as app.component.htmland the CSS Selector is app-root

Our index.html already have the app-root CSS selector defined

|  |  |
| --- | --- |
| 1  2  3  4  5 | <body>    <app-root></app-root>  </body> |

The Angular locates app-root in our index.html and renders our template between those tags.

# Components

What is an Angular Component

The Component is the main building block of an Angular Application.

The Component contains the data & user interaction logic that defines how the View looks and behaves. A view in Angular refers to a template (HTML).

The Angular Components are plain [JavaScript](https://www.tektutorialshub.com/javascript-tutorial/) classes and defined using **@Component Decorator**. This Decorator provides the component with the View to display & Metadata about the Component

The Component is responsible to provide the data to the view. The Angular does this by using [data binding](https://www.tektutorialshub.com/angular/angular-data-binding/) to get the data from the Component to the View. This is done using the special HTML markup knows as the Angular Template Syntax. The Component can also get notified when the View Changes.

The Angular applications will have lots of components. Each component handles a small part of UI. These components work together to produce the complete user interface of the application

The Components consists of three main building blocks

Template

Class

Building blocks of the Angular Components:

Template (View)

The template defines the layout and content of the View. Without the template,  there is nothing for Angular to render to the DOM.

The Templates are nothing but HTML codes along with the Angular specific special HTML markups ( knows as the Angular Template Syntax).

You can add [Angular directives](https://www.tektutorialshub.com/angular/angular-directives/) , [Angular Pipes](https://www.tektutorialshub.com/angular/angular-pipes/) & Other Angular Components on the template.

The data to Template comes from the Component, which in turn gets it from a [Angular Service](https://www.tektutorialshub.com/angular/angular-services/). Using the [data binding](https://www.tektutorialshub.com/angular/angular-data-binding/) techniques, we can keep the Template in sync with the Component. The templates can use the [Event Binding](https://www.tektutorialshub.com/angular/event-binding-in-angular/) or [two way data binding](https://www.tektutorialshub.com/angular/ngmodel-two-way-data-binding-in-angular/) to notify the component, when user changes something on the View.

There are two ways you can specify the Template in Angular.

Defining the Template Inline

Provide an external Template

Class

The Class provides the data & logic to the View. It contains the JavaScript code associated with Template (View). We use [TypeScript](https://www.tektutorialshub.com/typescript-tutorial/) to create the class, but you can also use JavaScript directly in the class.

Class Contains the Properties & Methods. The Properties of a class can be bind to the view using [Data Binding](https://www.tektutorialshub.com/angular/angular-data-binding/).

The simple Angular Class

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | export class AppComponent  {      title : string ="app"  } |

By convention we prefix the Component class with Component so as to easily identify them.

Metadata

Metadata Provides additional information about the component to the Angular. Angular uses this information to process the class. We use the @Component decorator to provide the Metadata to the Component.

@Component decorator

A decorator is a function that adds metadata to class, its methods & to its properties. The Components are defined with a @Component class decorator.

When Angular sees a class with @Component decorator, it treats the class as Component.

A Decorator is always prefixed with @. We must place the Decorator immediately before the class definition. We can also build our own decorators. The decorators are Similar to attributes in C#

Important Component metadata properties

Selector

Selector specifies the simple CSS selector. The Angular looks for the CSS selector in the template and renders the component there.

Providers

The Providers are the [Angular Services](https://www.tektutorialshub.com/angular/angular-services/), that our component going to use. The Services provide service to the Components or to the other Services.

Directives

The[directives](https://www.tektutorialshub.com/angular/angular-directives/) that this component going to use are listed here.

Styles/styleUrls

The CSS styles or style sheets, that this component needs. Here we can use either external stylesheet (using styleUrls) or inline styles (using Styles). The styles used here are specific to the component

template/templateUrl

The HTML template that defines our View. It tells Angular how to render the Component’s view. The templates can be inline (using a template) or we can use an external template (using a templateUrl). The Component can have only one template. You can either use inline template or external template and not both

Ad by Valueimpression

Creating the Component

We have already shown how to create the Angular Application using Angular CLI in [how to create first Angular application](https://www.tektutorialshub.com/angular/angular-create-first-application/) tutorial.  The Angular CLI has automatically created the root component **app.component.ts**.

In this tutorial, we will not create the Angular Component, but let us see the Component creation process in detail. The creation of the Angular component requires you to follow these steps

Create the Component file

Import the required external Classes/Functions

Create the Component class and export it

Add @Component decorator

Add metadata to @Component decorator

Create the Template

Create the CSS Styles

Register the Component in Angular Module

1. Creating the Component File

The Component app.component.ts. is already been created for us by Angular CLI under the folder src.

By Convention, the file name starts with the **feature name** (app) and then followed by the **type of class**(component). These are separated by a dot. The extension used is **ts** indicating that this is a typescript module file.

You can read more about naming conventions from the [Angular Style Guide](https://angular.io/guide/styleguide)

2. Import the Angular Component Library

Before we use any Angular (or external) functions or classes, we need to tell Angular how and where to find it. This is done using the Import statement. The Import statement is similar to the using statement in c#, which allows us to use the external modules in our class

To define the Component class, we need to use the @Component decorator. This function is part of the Angular Core library. So we import it in our class as shown below

|  |  |
| --- | --- |
| 1  2  3 | import { Component } from '@angular/core'; |

3. Create the Component Class and export it

The third step is to create the Component class using the export keyword. The Export keyword allows other components to use this component importing it. The AppComponent class is shown below

|  |  |
| --- | --- |
| 1  2  3  4  5 | export class AppComponent {    title = 'app';  } |

Note we are using Pascal case naming conventions for the class name. In the above class defines a Property named title and assign a default value “app”

4. Add @Component decorator

The next step is to inform the Angular that this is a Component class. We do that by adding the @Component decorator. We must add the decorator immediately above the class definition.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Component({  })  export class AppComponent {    title = 'app';  } |

5. Add meta data to @Component decorator

The next step is to add the metadata to the component using the @component decorator. Add the following to the component metadata

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  }) |

selector

The angular places the view (template) inside the selector app-root

templateUrl

In the above example, we have used an external template using templateUrl metadata. The templateUrl points to the external HTML file **app.component.html**.

styleUrls

Defines the styles for our template. The metadata points to the external CSS file **app.component.css**. The Component specific CSS styles can be specified here

6. Create the Template (View)

Template is nothing but an HTML file, which component must display it to the user

The Angular knows which template display, using the templateUrl metadata, which points to **app.component.html**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | <!--The content below is only a placeholder and can be replaced.-->  <div style="text-align:center">    <h1>      Welcome to {{title}}!    </h1>    <img width="300" src="data”  </div>  <h2>Here are some links to help you start: </h2>  <ul>    <li>      <h2><a target="\_blank" href="https://angular.io/tutorial">Tour of Heroes</a></h2>    </li>    <li>      <h2><a target="\_blank" href="https://github.com/angular/angular-cli/wiki">CLI Documentation</a></h2>    </li>    <li>      <h2><a target="\_blank" href="https://blog.angular.io//">Angular blog</a></h2>    </li>  </ul> |

By Convention, the file is named after the component file it is bound to with HTML extension.

This is a simple HTML file, except for the initial h1 tag

|  |  |
| --- | --- |
| 1  2  3  4  5 | <h1>      Welcome to {{title}}!    </h1> |

Note that **title inside the double curly bracket**. When rendering the view, the Angular looks for **title Property** in our component and binds the property to our view. This is called [data binding](https://www.tektutorialshub.com/angular/angular-data-binding/).

The double curly bracket syntax is known as [interpolation](https://www.tektutorialshub.com/angular/angular-data-binding/#Interpolation), which we will look at in our next tutorial

7. Add the Styles

The next step is to add the CSS Styles. The styleUrls metadata tells Angular, where to find the CSS File. This property points to external file **app.component.css**

By convention, we name the file after the component file with .css extension

Note that styleUrls metadata can accept multiple CSS Files.

8. Register the Angular Component in Angular Module

We have created the Angular Component. The next step is to register it with the [Angular Module](https://www.tektutorialshub.com/angular/angular-modules/)

The Angular Module organizes the components, [directives](https://www.tektutorialshub.com/angular/angular-directives/), [pipes](https://www.tektutorialshub.com/angular/angular-pipes/), and [services](https://www.tektutorialshub.com/angular/angular-services/)that are related and arrange them into cohesive blocks of functionality.

We use [@NgModule](https://www.tektutorialshub.com/angular/angular-modules/) class decorator to define a [Angular Module](https://www.tektutorialshub.com/angular/angular-modules/) and provide metadata about the Modules.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';    import { AppComponent } from './app.component';    @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

The **declaration arrays** is where we include the components, [pipes](https://www.tektutorialshub.com/angular/angular-pipes/) and [directives](https://www.tektutorialshub.com/angular/angular-directives/) that are part of this module.

We add all the other [Angular Modules](https://www.tektutorialshub.com/angular/angular-modules/) that this module uses in the **imports array**.

Include all the [Angular Services](https://www.tektutorialshub.com/angular/angular-services/) that are part of this module in the [**providers**](https://www.tektutorialshub.com/angular/angular-providers/)**‘ array**.

The Component that angular should load, when the app.module is loaded is assigned to the bootstrap property.

The AppComponent imported

|  |  |
| --- | --- |
| 1  2  3 | import { AppComponent } from './app.component'; |

and then added to the declarations array.

|  |  |
| --- | --- |
| 1  2  3  4 | @NgModule({    declarations: [ AppComponent ], |

We want appComponent to be loaded when Angular starts, thus we assign it to bootstrap property

|  |  |
| --- | --- |
| 1  2  3 | bootstrap: [AppComponent] |

Thats it

Finally, run the application from the command line using **ng serve**( or npm start). You should see this

Creating the inline Template & StyleUrls

In the above example, we have used the external template & Styles.

We can also specify the Template, Styles inline using the template and styles property of @Component metadata as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    template: '<h1> {{title}} works </h1>',    styles: ['h1 { font-weight: bold; }']  })  export class AppComponent {    title = 'app';  } |

The Template can get pretty long. In the case of a Multi-line template, you can use **BackTicks** ( **`**) to enclose the template string.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    template: `            <h1> {{title}} works </h1>            <p> a long inline template </p>             `,    styles: ['h1 { font-weight: bold; }']  })  export class AppComponent {    title = 'app';  } |

Specifying the Templates and styles inline has few disadvantageous. The Template can get pretty big and clutter your code. You will also not get the IntelliSense help while editing the template. In such a case you can write your Html template in an external file and link that file in your component code.

The Component selector

The Angular renders the components view in the DOM inside the **CSS selector**, that we defined in the Component decorator

|  |  |
| --- | --- |
| 1  2  3  4 | @Component({    selector: 'app-root', |

The selector **<app-root></app-root>**  is in the **index.html** (under src folder)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | <!doctype html>  <html lang="en">    <head>      <metacharset="utf-8">      <title>Angular4</title>      <basehref="/">      <metaname="viewport"content="width=device-width, initial-scale=1">      <linkrel="icon"type="image/x-icon"href="favicon.ico">    </head>    <body>      <app-root></app-root>    </body>  </html> |

When we build Angular Components, we are actually building new HTML elements. We specify the name of the HTML element in the selector property of the component metadata. And then we use it in our HTML.

The Angular looks for the selector in the current view. If it finds a selector, it renders the view of the component that is associated with that selector at that location.

Remember Angular is Single Page application (SPA)

The only page is shown to you is index.html. Components are added and removed from the index.html

Contents of app.component.html is inserted into the location <app-root></app-root>. Because app-root is the selector we used for the AppComponent. AppComponent is the first component that Angular loads, hence it is our Root Component.

The app.component.html may refer to <customer-list></customer-list>. The Angular looks for the component (for example CustomerListComponent) with selector customer-list and renders it there. This component now becomes the child component of the AppComponent.

The Template of CustomerListComponent may include more selectors. The Angular recursively going to look for them and renders them. They will become child of CustomerListComponent.

This will create a tree of components. You can refer to the tutorial on how to create a child component from our tutorial [Angular child component](https://www.tektutorialshub.com/angular/angular-adding-child-component/)

There are several ways you can specify the Component selector

Using the CSS class name

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Component({    selector: '.app-root'  }) |

And in the HTML markup use the CSS class name

|  |  |
| --- | --- |
| 1  2  3 | <div class="app-root"></div> |

Using attribute name

|  |  |
| --- | --- |
| 1  2  3 | @Component({ selector: '[app-root]' }) |

And you can now the attribute as follows

|  |  |
| --- | --- |
| 1  2  3 | <div app-root></div> |

Using attribute name and value

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Component({     selector: 'div[app=components]'  }) |

now you can use it as follows

|  |  |
| --- | --- |
| 1  2  3 | <div app="components"></div> |

What is Angular Data Binding

Data binding is a technique, where the data stays in sync between the component and the view. Whenever the user updates the data in the view, Angular updates the component. When the component gets new data, the Angular updates the view.

There are many uses of data binding. You can show models to the user, dynamically Change element style, respond to user events, etc

Data Binding in Angular

The data binding in Angular can be broadly classified into two groups. One way binding or two-way binding

One way binding

In one way binding data flows from one direction. Either from view to component or from component to view.

From Component to View

To bind data from component to view, we make use of Interpolation & Property Binding.

Interpolation

[Interpolation](https://www.tektutorialshub.com/angular/interpolation-in-angular/) allows us to include expressions as part of any string literal, which we use in our HTML. The angular evaluates the expressions into a string and replaces it in the original string and updates the view. You can use interpolation wherever you use a string literal in the view

The Angular uses the {{ }} (double curly braces) in the template to denote the interpolation. The syntax is as shown below

{{ templateExpression }}

The content inside the double braces is called **Template Expression**

The Angular first evaluates the Template Expression and converts it into a string. Then it replaces Template expression with the result in the original string in the HTML. Whenever the template expression changes, the Angular updates the original string again

Example

|  |  |
| --- | --- |
| 1  2  3 | Welcome,  {{firstName}} {{lastName}} |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    firstName= 'Sachin';    lastName=”Tendulkar”  } |

Run the app and you will see Welcome, Sachin Tendulkar in the output. The Angular replaces both {{firstName}} & {{lastName}} with the values of firstName & lastName variable from the component.

Also, whenever the values of firstName & lastName change, Angular updates the view. But not the other way around.

Property binding

The [Property binding](https://www.tektutorialshub.com/angular/property-binding-in-angular/) allows us to bind HTML element property to a property in the component. Whenever the value of the component changes, the Angular updates the element property in the View. You can set the properties such as class, href, src, textContent, etc using property binding. You can also use it to set the properties of custom components or directives (properties decorated with @Input).

The Property Binding uses the following Syntax

[binding-target]=”binding-source”

The binding-target (or target property) is enclosed in a square bracket []. It should match the name of the property of the enclosing element.

Binding-source is enclosed in quotes and we assign it to the binding-target. The Binding source must be a template expression. It can be property in the component, method in component, a template reference variable or an expression containing all of them.

Whenever the value of Binding-source changes, the view is updated by the Angular.

Example

app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <h1 [innerText]="title"></h1>  <h2>Example 1</h2>  <button [disabled]="isDisabled">I am disabled</button> |

app.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    title="Angular Property Binding Example"      //Example 1    isDisabled= true;    } |

The title property of the component class is bound to the innerText property of the h1 tag. Disabled Property of the button is bound to the isDisabled Property of the component

Whenever we modify the title or isDisabled in the component, the Angular automatically updates the view.

The property binding has special syntaxes for setting the class & styles. Also, both interpolation & property binding does not set the attributes of the HTML elements. Hence we have an attribute binding to such situations

**Read More**[**Angular Property Binding**](https://www.tektutorialshub.com/angular/property-binding-in-angular/)

[Class Binding](https://www.tektutorialshub.com/angular/class-binding-in-angular/)

You can set the class in the following ways. Click on the links to find out more

[ClassName Property binding](https://www.tektutorialshub.com/angular/class-binding-in-angular/#class-binding-with-classname)

Set the Class attribute with [class binding](https://www.tektutorialshub.com/angular/class-binding-in-angular/)

[ngClass directive](https://www.tektutorialshub.com/angular/angular-ngclass-directive/)

[Style Binding](https://www.tektutorialshub.com/angular/angular-style-binding/)

You can set the class in the following ways. Click on the links to find out more

[Style Binding](https://www.tektutorialshub.com/angular/angular-style-binding/)

[ngStyle directive](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/)

Attribute binding

Sometimes there is no HTML element property to bind to. The examples are [aria](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA) (accessibility) Attributes & [SVG](https://developer.mozilla.org/en-US/docs/Web/SVG). In such cases, you can make use of attribute binding

The attribute syntax starts with attr followed by a dot and then the name of the attribute as shown below

|  |  |
| --- | --- |
| 1  2  3 | <button [attr.aria-label]="closeLabel" (onclick)="closeMe()">X</button> |

**Read More**

[**ngClass Directive**](https://www.tektutorialshub.com/angular/angular-ngclass-directive/)

[**Style Binding**](https://www.tektutorialshub.com/angular/angular-style-binding/)

[**ngStyle directive**](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/)

From View to Component

Event Binding

Event binding allows us to bind events such as keystrokes, clicks, hover, touch, etc to a method in component. It is one way from view to component. By tracking the user events in the view and responding to it, we can keep our component in sync with the view. For Example, when the user changes input in a text box, we can update the model in the component, run some validations, etc. When the user submits the button, we can then save the model to the backend server.

Angular uses the following syntax for event binding

<target-event)="TemplateStatement"

Angular event binding syntax consists of a target event name within parentheses on the left of an equal sign, and a quoted template statement on the right.

For Example,

|  |  |
| --- | --- |
| 1  2  3 | <button (click)="onSave()">Save</button> |

The above example binds the click event of a button to a onSave() method in the component class. Whenever the user clicks on the button, the Angular invokes the onSave() method.

Two Way binding

Two-way binding means that changes made to our model in the component are propagated to the view and that any changes made in the view are immediately updated in the underlying component

Two-way binding is useful in data entry forms. Whenever a user makes changes to a form field, we would like to update our model. Similarly, when we update the model with new data, we would like to update the view as well

The two-way binding uses the special syntax known as a banana in a box [()]

<someElement [(someProperty)]="value"></someElement>.

The above syntax sets up both property binding & event binding. But to make use of it, the property must have the change event with the name <propertyName>Change

But, angular has a special directive ngModel, which sets up the two-way binding

ngModel

The Angular uses the ngModel directive to achieve the two-way binding on HTML Form elements. It binds to a form element like input, select, selectarea. etc.

The ngModel directive is not part of the Angular Core library. It is part of the @angular/forms. You need to import the FormsModule package into your Angular module.

|  |  |
| --- | --- |
| 1  2  3 | import { FormsModule } from '@angular/forms'; |

Then you can use it using the two-way binding syntax as shown below

|  |  |
| --- | --- |
| 1  2  3 | <input type="text" name="value" [(ngModel)]="value"> |

When you bind to a ngModel directive, behind the scene it sets up property binding & event binding. It binds to the value property of the element using property binding. It then uses the ngModelChange event to sets up the event binding to listen to the changes to the value.

Summary:

The Data binding in Angular consists of interpolation & property binding which is one way from component to view. Interpolation allows us to embed an expression in a string literal. Using the Property binding we can set element property to the component class. The event binding is from view to component and allows us to respond to the events like keypress, button click, etc.. We combine property binding & event binding to create a two-way data binding. The ngModel directive from the Angular FormsModule, set up the two way data binding on form elements.

What is Interpolation in Angular

Interpolation allows us to include expressions as part of any string literal, which we use in our HTML. The angular evaluates the expressions into a string and replaces it in the original string and updates the view. You can use interpolation wherever you use a string literal in the view.

Angular interpolation is also known by the name ***string interpolation***. Because you incorporate expressions inside another string.

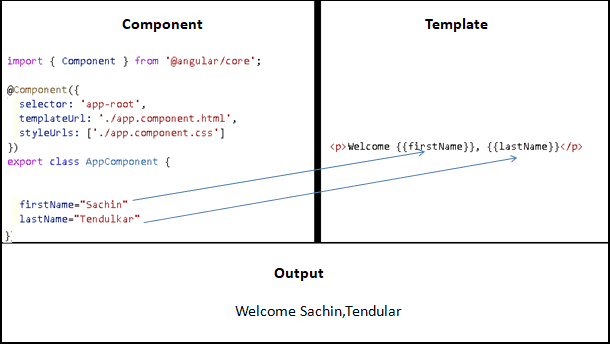
Interpolation syntax

The Angular uses the {{ }} (double curly braces) in the template to denote the interpolation. The syntax is as shown below

|  |  |
| --- | --- |
| 1  2  3 | {{ templateExpression }} |

The content inside the double braces is called Template Expression

The Angular first evaluates the Template Expression and converts it into a string. Then it replaces Template expression with the result in the original string in the HTML. Whenever the template expression changes, the Angular updates the original string again



Interpolation Example

Create a new angular application using the following command

|  |  |
| --- | --- |
| 1  2  3 | ng new interpolation |

Open the app.component.html and just add the following code

|  |  |
| --- | --- |
| 1  2  3 | {{title}} |

Open the app.component.ts and add the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'Angular Interpolation Example';  } |

Run the app. You will see the “Angular Interpolation Example” on the screen

In the example above the title is the Template Expression. We also have title property in the component. The Angular evaluates {{title}} and replaces it with the value of the title property from the component class.

If the user changes the title in the component class, the Angular updates the view accordingly.

The interpolation is much more powerful than just getting the property of the [component](https://www.tektutorialshub.com/angular/angular-component/). You can use it to invoke any method on the component class or to do some mathematical operations etc.

Notes on Interpolation

Interpolation is one-way binding

Interpolation is one way as values go from the component to the template. When the component values change, the Angular updates the view. But if the values changes in the view components are not updated.

Should not change the state of the app

The Template expression should not change the state of the application. The Angular uses it to read the values from the component and populate the view. If the Template expression changes the component values, then the rendered view would be inconsistent with the model

It means that you cannot make use of the following

Assignments (=, +=, -=, …)

Keywords like new, typeof, instanceof, etc

Chaining expressions with ; or ,

The increment and decrement operators ++ and --

bitwise operators such as | and &

The expression must result in a string

Interpolation expression must result in a string. If we return an object it will not work. If you want to bind the expression that is other than a string (for example – boolean), then [Property Binding](https://www.tektutorialshub.com/angular/property-binding-in-angular/) is the best option.

Works only on Properties & not attributes

Interpolation and property binding can set only properties, not attributes. For Attributes use attribute binding

Examples of interpolation

You can use interpolation to invoke a method in the component, Concatenate two string, perform some mathematical operations or change the property of the DOM element like color, etc.

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Invoke a method in the component

We can invoke the component’s methods using interpolation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | //Template    {{getTitle()}}      //Component  title = 'Angular Interpolation Example';  getTitle(): string {       return this.title;  } |

Concatenate two string

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <p>Welcome to {{title}}</p>  <p>{{ 'Hello & Welcome to '+ ' Angular Interpolation '}}</p>  <p>Welcome {{firstName}}, {{lastName}}</p>  <p>Welcome {{getFirstName()}}, {{getLastName()}}</p> |

Perform some mathematical operations

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <h2>Mathematical Operations</h2>    <p>100x80 = {{100\*80}}</p>  <p>Largest: {{max(100, 200)}}</p>    //Component  max(first: number, second: number): number {    return Math.max(first, second);  } |

Bind to an element property

We can use it to bind to a property of the HTML element, a component, or a directive. in the following code, we bind to the style.color property of the <p> element. We can bind to any property that accepts a string.

|  |  |
| --- | --- |
| 1  2  3  4 | <p>Show me <span class = "{{giveMeRed}}">red</span></p>  <p style.color={{giveMeRed}}>This is red</p> |

**Bind to an image source**

|  |  |
| --- | --- |
| 1  2  3 | <div><img src="{{itemImageUrl}}"></div> |

**href**

|  |  |
| --- | --- |
| 1  2  3 | <a href="/product/{{productID}}">{{productName}}</a> |

Use a template reference variable

You can also use the [template reference variable](https://www.tektutorialshub.com/angular/template-reference-variable-in-angular/). The following example creates a template variable #name to an input box. You can use it get the value of the input field {{name.value}}

|  |  |
| --- | --- |
| 1  2  3  4  5 | <label>Enter Your Name</label>  <input (keyup)="0" #name>  <p>Welcome {{name.value}} </p> |

We also use (keyup)="0" on the input element. It does nothing but it forces the angular run the change detection, which in turn updates the view.

The Angular updates the view, when it runs the change detection. The change detection runs only in response to asynchronous events, such as the arrival of HTTP responses, raising of events, etc. In the example above whenever you type on the input box, it raises the keyup event. It forces the angular run the change detection, hence the view gets the latest values.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | //Template    <p>items</p>  <ul>    <li \*ngFor="let d of items">      {{ d.name }}      </li>  </ul>    //Component    items= [      new item(1, 'Mobile'),      new item(2, 'Laptop'),      new item(3, 'Desktop'),      new item(4, 'Printer')    ]    class item {    code:string    name:string      constructor(code,name) {      this.code=code;      this.name=name    }  } |

Cross-site Scripting or XSS

Angular Sanitizes everything before inserting into DOM, thus preventing Cross-Site Scripting Security bugs (XSS). For example values from the component property, attribute, style, class binding, or interpolation, etc are sanitized. The script in the following example is not invoked but shown as it is.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | //Template  <p>{{script}}</p>  <p>{{div}}</p>    //Component  script ='<script>alert("You are hacked")</script>'  div='<div>this is a div</div>'; |

NgNonBindable

Use ngNonBindable to tell Angular not to compile or bind the contents of the current DOM element. I.e any expression is not evaluated but shown as it is.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <p>Evaluate: {{variable}}</p>  <p ngNonBindable>Do not evaluate: {{variable}}</p>      <p>Angular uses {{ variable }} syntax for Interpolation</p>  <p ngNonBindable>Angular uses {{ variable }} syntax for Interpolation</p> |

Use Pipes

You can make use of [Angular Pipes](https://www.tektutorialshub.com/angular/angular-pipes/) to transform the expression result. Like converting to an uppercase, date formats, adding currency symbols, etc

|  |  |
| --- | --- |
| 1  2  3  4  5 | <p>uppercase pipe: {{title | uppercase}}</p>  <p>pipe chain: {{title | uppercase | lowercase}}</p>  <p>json pipe: {{items | json}}</p> |

The safe navigation operator ( ? )

You can make use of a safe navigation operator ( ? ) to guards against null and undefined values.

The following code results in an error because there is no nullItem

|  |  |
| --- | --- |
| 1  2  3  4  5 | <p>The item name is: {{nullItem?.Name}}</p>    TypeError: Cannot read property 'itemName' of undefined |

Use a safe navigation operator and the error goes away. Angular replace it with an empty string

|  |  |
| --- | --- |
| 1  2  3 | <p>The null item name is {{nullItem.itemName}}</p> |

The non-null assertion operator

Typescript enforces the strict null checking if you enable the --strictNullChecks flag in your tsconfig.json. Under strict null check any variable not defined or null results in a compiler error. The type checker also throws an error if it can’t determine whether a variable will be null or undefined at runtime

You can use the non-null assertion operator to inform typescript not to throw any compile errors. Note that it is a compile-time feature & not runtime.

|  |  |
| --- | --- |
| 1  2  3 | The item's name is: {{item!.name}} |

Property Binding Syntax

The Property Binding uses the following Syntax

|  |  |
| --- | --- |
| 1  2  3 | [binding-target]=”binding-source” |

The binding-target (or target property) is enclosed in a square bracket []. It should match the name of the property of the enclosing element.

Binding-source is enclosed in quotes and we assign it to the binding-target. The Binding source must be a template expression. It can be property in the component, method in component, a template reference variable or an expression containing all of them.

Whenever the value of Binding-source changes, the view is updated by the Angular.

Property Binding Example

Create a new application

|  |  |
| --- | --- |
| 1  2  3 | ng new property |

Open app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5 | <h1 [innerText]="title"></h1>  <h2>Example 1</h2>  <button [disabled]="isDisabled">I am disabled</button> |

Open the app.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    title="Angular Property Binding Example"      //Example 1    isDisabled= true;    } |

We have two property binding in the example above

The title property of the component class is bound to the innerText property of the h1 tag. Disabled Property of the button is bound to the isDisabled Property of the component

Whenever we modify the title or isDisabled is the component, the Angular automatically updates the view.

Property Binding is one way

Property binding is one way as values go from the component to the template. When the component values change, the Angular updates the view. But if the values changes in the view, the Angular does not update the component.

Should not change the state of the app

The Angular evaluates the template expression (binding-source) to read the values from the component. It then populates the view. If the expression changes any of the component values, then the view would be inconsistent with the model. Hence we need to avoid using expression which will alter the component state.

It means that you cannot make use of the following

Assignments (=, +=, -=, …)

Keywords like new, typeof, instanceof, etc

Chaining expressions with ; or ,

The increment and decrement operators ++ and --

bitwise operators such as | and &

Return the proper type

The binding expression should return the correct type. The type that the target property expects. Otherwise, it will not work

Property name in camel case

There are few element property names in the camel case, while their corresponding attributes are not. For example rowSpan & colSpan properties of the table are in the camel case. The HTML attributes are all lowercase (rowspan & colspan)

Remember the brackets

The brackets, [], tell Angular to evaluate the template expression. If you omit the brackets, Angular treats the expression as a constant string and initializes the target property with that string:

Content Security

Angular inspects the template expression for untrusted values and sanitizes them if found any. For example, the following component variable evilText contains the script tag. This is what we call the script injection attack. **The Angular does not allow HTML with script tags**. It treats the entire content as string and displays as it is.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Component    evilText = 'Template <script>alert("You are hacked")</script> Syntax';    Template    <p [textContent]="evilText"></p> |

DOM Properties, not attributes

The property binding binds to the properties of DOM elements, components, and directives and not to HTML attributes. The angular has a special syntax for attribute binding.

Special Binding

The Angular has a special syntax for class, styles & attribute binding

The classes & styles are special because they contain a list of classes or styles. The bindings need to be more flexible in managing them. Hence we have a class & style binding.

The Property bindings cover all the properties, but there are certain HTML attributes that do not have any corresponding HTML property. Hence we have attribute binding

Class binding

You can set the class in the following ways. Click on the links to find out more

[ClassName Property binding](https://www.tektutorialshub.com/angular/class-binding-in-angular/#class-binding-with-classname)

[Set the Class attribute with class binding](https://www.tektutorialshub.com/angular/class-binding-in-angular/)

[ngClass directive](https://www.tektutorialshub.com/angular/angular-ngclass-directive/)

Style Binding

Similar to the class, the style also can be set using the following ways. Click on the links to find out more

Attribute Binding

Sometimes there is no HTML element property to bind to. The examples are [aria](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA) (accessibility) Attributes & [SVG](https://developer.mozilla.org/en-US/docs/Web/SVG). In such cases, you can make use of attribute binding

The attribute syntax starts with attr followed by a dot and then the name of the attribute as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | //Template    //Setting aria label  <button [attr.aria-label]="closeLabel" (onclick)="closeMe()">X</button>    //Table colspan  <table border="1">    <tr>      <td>Col 1</td>      <td>Col 2</td>      <td>Col 3</td>    </tr>    <tr>        <td [attr.colspan]="2">Col 1 & 2</td>        <td>Col 3</td>    </tr>    <tr>        <td>Col 1</td>        <td bind-attr.colspan = "getColspan()">Col 2 & 3 </td>    </tr>    <tr>        <td>Col 1</td>        <td>Col 2</td>        <td>Col 3</td>      </tr>    </table> |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | //Component    closeLabel="close";  getColspan() {     return "2"  } |

Property Binding Vs Interpolation

Everything that can be done from interpolation can also be done using the Property binding. Interpolation is actually a shorthand for binding to the textContent property of an element.

For example the following interpolation

|  |  |
| --- | --- |
| 1  2  3 | <h1> {{ title }} </h1> |

Is same as the following Property binding

|  |  |
| --- | --- |
| 1  2  3 | <h1 [innerText]="title"></h1> |

In fact, Angular automatically translates interpolations into the corresponding property bindings before rendering the view.

Interpolation is simple and readable. For example, the above example of setting the h1 tag, the in interpolation is intuitive and readable than the property binding syntax

Interpolation requires the expression to return a string. If you want to set an element property to a non-string data value, you must use property binding.

Property Binding Example

**Binding to innerHTML with HTML tags**

Here the Angular parses the b &p tags and renders it in the view.

|  |  |
| --- | --- |
| 1  2  3  4  5 | //Template  <p [innerHTML]="text1"></p>  <div [innerHTML]="text2"></div> |

|  |  |
| --- | --- |
| 1  2  3  4  5 | //Component  text1="The <b>Angular</b> is printed in bold"  text2="<p>This is first para</p><p>This is second para</p> " |

**img**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | //Template    <img [src]="itemImageUrl">  <img bind-src="itemImageUrl"> |

|  |  |
| --- | --- |
| 1  2  3  4 | //Component  itemImageUrl="https://angular.io/assets/images/logos/angular/logo-nav@2x.png" |

**Concatenate two string**

|  |  |
| --- | --- |
| 1  2  3 | <p [innerText]="'Hello & Welcome to '+ ' Angular Data binding '"></p> |

**Mathematical expressions**

|  |  |
| --- | --- |
| 1  2  3 | <p [innerText]="100\*80"></p> |

**setting the color**

|  |  |
| --- | --- |
| 1  2  3  4 | //template  <p [style.color]="color">This is red</p> |

|  |  |
| --- | --- |
| 1  2  3  4 | //Component  color='red' |

Event Binding

Event binding allows us to bind events such as keystroke, clicks, hover, touche, etc to a method in component. It is one way from view to component. By tracking the user events in the view and responding to it, we can keep our component in sync with the view. For Example, when the user changes to an input in a text box, we can update the model in the component, run some validations, etc. When the user submits the button, we can then save the model to the backend server.

Syntax

The Angular event binding consists of two parts

|  |  |
| --- | --- |
| 1  2  3 | (target-event)="TemplateStatement" |

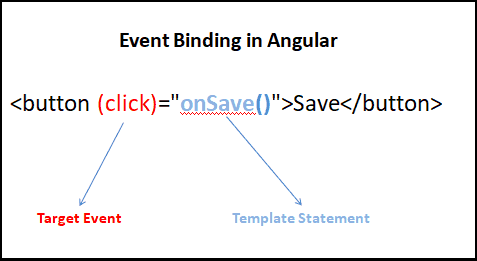
We enclose the target event name in parentheses on the left side

Assign it to a template statement within a quote on the right side

Angular event binding syntax consists of a target event name within parentheses on the left of an equal sign, and a quoted template statement on the right.

The following event binding listens for the button’s click events, calling the component’s onSave() method whenever a click occurs

|  |  |
| --- | --- |
| 1  2  3 | <button (click)="onSave()">Save</button> |



Event Binding Example

Create a new angular application

|  |  |
| --- | --- |
| 1  2  3 | ng new event |

Copy the following code to app.component.html

***app.component.html***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <h1 [innerText]="title"></h1>    <h2>Example 1</h2>  <button (click)="clickMe()">Click Me</button>  <p>You have clicked {{clickCount}}</p> |

[***Source Code***](https://stackblitz.com/edit/event-binding-in-angular-ex-1)

Add the following code to the app.component.ts

***app.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | clickCount=0    clickMe() {      this.clickCount++;    } |

[***Source Code***](https://stackblitz.com/edit/event-binding-in-angular-ex-1)

In the above example, the component listens to the click event on the button. It then executes the clickMe() method and increases the clickCount by one.

Template statements have side effects

Unlike the [Property Binding](https://www.tektutorialshub.com/angular/property-binding-in-angular/) & [Interpolation](https://www.tektutorialshub.com/angular/interpolation-in-angular/), where we use the template expression is used, in the case of event binding we use template statement.

The Template statement can change the state of the component. Angular runs the change detection and updates the view so as to keep it in sync with the component.

on-

Instead of parentheses, you can also use the on- syntax as shown below.

|  |  |
| --- | --- |
| 1  2  3 | <button on-click="clickMe()">Click Me</button> |

Multiple event handlers

You can also bind an unlimited number of event handlers on the same event by separating them with a semicolon :

Add a new component property

|  |  |
| --- | --- |
| 1  2  3 | clickCount1=0; |

[***Source Code***](https://stackblitz.com/edit/event-binding-in-angular-ex-2?file=src/app/app.component.html)

And in the template, call clickMe() method and then an assignment clickCount1=clickCount

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | //Template    <h2>Example 2</h2>  <button (click)="clickMe() ; clickCount1=clickCount">Click Me</button>  <p>You have clicked {{clickCount}}</p>  <p>You have clicked {{clickCount1}}</p> |

[***Source Code***](https://stackblitz.com/edit/event-binding-in-angular-ex-2?file=src/app/app.component.html)

$event Payload

DOM Events carries the event payload. I.e the information about the event. We can access the event payload by using $event as an argument to the handler function.

|  |  |
| --- | --- |
| 1  2  3  4 | <input (input)="handleInput($event)">  <p>You have entered {{value}}</p> |

And in the component

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | value=""  handleInput(event) {    this.value = (event.target as HTMLInputElement).value;  } |

[***Source Code***](https://stackblitz.com/edit/event-binding-in-angular-ex-3?file=src/app/app.component.ts)

The properties of a $event object vary depending on the type of DOM event. For example, a mouse event includes different information than an input box editing event.

Remember you need to use the variable as $event in the Template statement. Example handleInput($event). Otherwise, it will result in an error

Template reference variable

We can also make use of the template reference variable to pass the value instead of $event.

In the template

|  |  |
| --- | --- |
| 1  2  3  4  5 | <h2>Template Reference Variable</h2>  <input #el (input)="handleInput1(el)">  <p>You have entered {{val}}</p> |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | val="";  handleInput1(element) {    this.val=element.value;  } |

Key event filtering (with key.enter)

We use keyup/keydown events to listen for keystrokes. The following example does that

|  |  |
| --- | --- |
| 1  2  3  4 | <input (keyup)="value1= $any($event.target).value" />  <p>You entered {{value1}}</p> |

[***Source Code***](https://stackblitz.com/edit/event-binding-in-angular-ex-4-zg4ifv?file=src/app/app.component.html)

But Angular also offers a feature, where it helps to filter out certain keys. For Example, if you want to listen only to the enter keys you can do it easily

|  |  |
| --- | --- |
| 1  2  3  4 | <input (keyup.enter)="value2=$any($event.target).value">  <p>You entered {{value2}}</p> |

Here is an interesting example. On pressing enter key it updates the value3 variable and on escape clears the variable.

|  |  |
| --- | --- |
| 1  2  3  4 | <input (keyup.enter)="value3=$any($event.target).value" (keyup.escape)="$any($event.target).value='';value3=''">  <p>You entered {{value3}}</p> |

Note that we are using $any to cast $event.target to [any type](https://www.tektutorialshub.com/typescript/typescript-any-type/). Otherwise, the typescript will raise the error [Property ‘value’ does not exist on type ‘EventTarget’ Error in Angular](https://www.tektutorialshub.com/angular/property-value-does-not-exist-on-type-eventtarget-error-in-angular/)

Angular calls these pseudo-events.

You can also listen for the key combination

<input (keyup.control.shift.enter)="value4=$any($event.target).value">

<p>You entered {{value4}}</p>

Custom events with EventEmitter

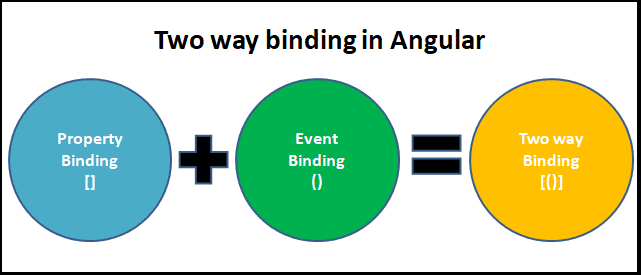
Directives & components can also raise events with [EventEmitter](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/). Using [EventEmiiiter](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) you can create a property and raise it using the EventEmitter.emit(payload). The Parent component can listen to these events using the event binding and also read the payload using the $event argument.

What is Two way data binding

Two way data binding means that changes made to our model in the component are propagated to the view and that any changes made in the view are immediately updated in the underlying component data.

Two way data binding is useful in data entry forms. Whenever a user makes changes to a form field, we would like to update our model. Similarly, when we update the model with new data, we would like to update the view as well

The two way data binding is nothing but both property binding & event binding applied together. Property Binding is one way from component to view. The event binding is one way from view to component. If we combine both we will get the Two-way binding.



Two way using property & Event Binding

The following example shows how we can achieve two-way binding using the combination of property binding & event binding

Create a new Angular application

copy the following code to app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | h2>Example 1</h2>  <input type="text" [value]="name" (input)="name=$event.target.value">  <p> You entered {{name}}</p>  <button (click)="clearName()">Clear</button> |

Update the app.component.ts with the following code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | name=""  clearName() {     this.name="";  } |

We bind the name property to the input element ([value]="name"). We also use the event binding (input)="name=$event.target.value". It updates the name property whenever the input changes. The Angular interpolation updates the {{name}}, so we know the value of name property.

$event.target.value raises the error Property ‘value’ does not exist on type ‘EventTarget’ if fullTemplateTypeCheck is set to true under angularCompilerOptions in the tsconfig.json.  
  
The error is due to the fact that the value property is not guaranteed to exist in the $event.target.  
  
To solve this problem either you can use the $any typecast function ($any($event.target).value) to stop the type checking in the template or set fullTemplateTypeCheck to false in tsconfig.json.

Two-way binding syntax

The above example uses the event & property binding combination to achieve the two-way binding. But Angular does provide a way to achieve the two-way binding using the syntax [()]. Note that both square & parentheses are used here. This is now known as **Banana in a box** syntax. The square indicates the Property binding & parentheses indicates the event binding.

For Example

|  |  |
| --- | --- |
| 1  2  3 | <someElement [(someProperty)]="value"></someElement> |

The above syntax sets up both property & event binding. But to make use of it, the property must follow the following naming convention.

If we are binding to a settable property called someProperty of an element, then the element must have the corresponding change event named somePropertyChange

But most HTML elements have a value property. But do not have a valueChange event, instead, they usually have an input event. Hence they cannot be used in the above syntax

For Example, the following will not work as there is no valueChange event supported by the input element.

Hence we have a ngModel directive.

What is ngModel

The Angular uses the ngModel directive to achieve the two-way binding on HTML Form elements. It binds to a form element like input, select, selectarea. etc.

Internally It uses the ngModel in property, binding to bind to the value property and ngModelChange which binds to the input event.

How to use ngModel

The ngModel directive is not part of the Angular Core library. It is part of the FormsModule library. You need to import the FormsModule package into your Angular module.

In the template use the following syntax

|  |  |
| --- | --- |
| 1  2  3 | <input type="text" name="value" [(ngModel)]="value"> |

The ngModel directive placed inside the square & parentheses as shown above. This is assigned to the Template Expression. Template Expression is the property in the component class

ngModel Example

Import FormsModule

Open the app.module.ts and make the following changes

|  |  |
| --- | --- |
| 1  2  3 | import { FormsModule } from '@angular/forms'; |

**Template**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <h2>Example 2</h2>  <input type="text" name="value" [(ngModel)]="value">  <p> You entered {{value}}</p>  <button (click)="clearValue()">Clear</button> |

**Component**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | value="";  clearValue() {     this.value="";  } |

The ngModel data property sets the element’s value property and the ngModelChange event property listens for changes to the element’s value.

Run the project and see that as you modify the name, the component class model is automatically updated.

Custom Two-way binding

As we mentioned earlier the [()] to work, we need to have a property with the change event as<nameofProperty>Change.

We do not have any HTML Elements which follows the above naming conventions, but we can create a custom component

create new component and name it as counter.component.ts. copy the following code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import { Component, Input, Output, EventEmitter } from '@angular/core';  @Component({  selector: 'counter',  template: `       <div>         <p>           Count: {{ count }}           <button (click)="increment()">Increment</button>         </p>       </div>     `  })  export class CounterComponent {      @Input() count: number = 0;    @Output() countChange: EventEmitter<number> = new EventEmitter<number>();      increment() {     this.count++;     this.countChange.emit(this.count);  }  } |

The component has two properties one is input property count decorated with @Input(). The other in is an event (or output property), which we decorate with @Output(). We name the input property as count. Hence the output property becomes countChange

Now we can use this component and create two-way binding to the count property using the syntax [(count)].

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <h2>Example 3</h2>  <counter [(count)]="count"></counter>  <p> Current Count {{count}}</p>  <button (click)="clearCount()">Clear</button> |

Summary

The two-way binding is a simple, but yet powerful mechanism. We use Property binding & Event binding to achieve the two-way binding. Angular does have a [(value)] syntax to which sets up the two-way binding. It automatically sets up property binding to the value property of the element. It also sets up the event binding to valueChange Property. But since we hardly have any HTML element, which follows those naming conventions unless we create our own component. This is where ngModel directive from FormsModule steps in and provides two way binding to all the known HTML form elements.

NgModelChange Example

The following is the simple example of ngModelChange.

We assign the method in the component class (handler function) to the ngModelChange using the [event binding](https://www.tektutorialshub.com/angular/event-binding-in-angular/) syntax

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | //Template    Name:  <input type="text" name="name" ngModel (ngModelChange)="nameChanged($event)"> |

nameChanged is the handler function, which we need to define in the component class. We can access the new value by using the $event as an argument to the handler function.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | //Component  nameChanged(arg) {    console.log("modelchanged " + arg);  } |

ngModel

We usually use the ngModel as follows to achieve the [two-way](https://www.tektutorialshub.com/angular/ngmodel-two-way-data-binding-in-angular/) binding. [(ngModel)]="email" keeps the email property in the component class in sync with the template.

|  |  |
| --- | --- |
| 1  2  3 | <input type="text" name="email" [(ngModel)]="email"> |

Internally, Angular converts the above syntax to the following syntax.

|  |  |
| --- | --- |
| 1  2  3 | <input [ngModel]="email" (ngModelChange)="email = $event"> |

The component property email is bound to the input element using the [property binding](https://www.tektutorialshub.com/angular/property-binding-in-angular/) ( [ngModel]="email"). Any changes made to the input is updated in the component using the (ngModelChange)="email = $event" [event binding](https://www.tektutorialshub.com/angular/event-binding-in-angular/).

ngModelChange with ngModel

Consider the following example.

|  |  |
| --- | --- |
| 1  2  3 | <input [(ngModel)]="firstName" (ngModelChange)="firstNameChanged($event)"/> |

The Angular converts the above to the following syntax. We end up with the two ngModelChange event bindings.

|  |  |
| --- | --- |
| 1  2  3 | <input [ngModel]="firstName (ngModelChange)="firstName = $event"   (ngModelChange)="firstNameChanged($event)"/> |

Here the ngModelChange fires in the order it is specified. Hence the (ngModelChange)="firstName = $event" fires first. (ngModelChange)="firstNameChanged($event)" fires later.

Hence in the component class, the arg & this.firstName is always the same.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | firstName  ;    firstNameChanged(arg) {    console.log(        "firstNameChanged  argument " + arg + "  component " + this.firstName    );  } |

But if you put ngModelChange ahead of the ngModel as in the example below

|  |  |
| --- | --- |
| 1  2  3  4 | <input (ngModelChange)="lastNameChanged($event)" [(ngModel)]="lastName" /> |

Angular internally converts it as follows

|  |  |
| --- | --- |
| 1  2  3 | <input (ngModelChange)="lastNameChanged($event)" [ngModel]="lastName" (ngModelChange)="lastName= $event"   /> |

Here (ngModelChange)="lastNameChanged($event)" fires first. Hence in the component class arg contains the latest value of the, while this.lastName **still holds the previous value**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | lastName  ;    lastNameChanged(arg) {    console.log(        "lastNameChanged  argument " + arg + "  component " + this.lastName    );  } |

Change Event

The (change) is a DOM event fires when changes to the form fields like <input>, <select>, and <textarea> is committed by the user.

This event fires when

user changes the input & moves the focus away from the text box (blur event)

On <select> it fires when the user selects a new option either by a mouse click or using a keyboard.

Fires when the state of a check box or radio button change due to users action

Change Event Example

The following example shows how to use the change event in Angular.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | Name  <input type="text" name="name1" (change)="name1Changed($event)">      <br>  country  <select name="country1" (change)="country1Changed($event)" >    <option [ngValue]="null" disabled>Select Country</option>    <option \*ngFor="let country of countries" [ngValue]="country.id">{{country.name}}</option>  </select> |

Component class

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | name1Changed(arg) {  console.log("name1Changed " + arg.target.value);  console.log(arg);  }    country1Changed(arg) {    console.log("country1Changed " + arg.target.value);    console.log(arg);  } |

The change event for text element fires when we move the focus away from the element (blurred the input). This is different from the ngModelChange, which fires the event for each input change.

The other import point is the $event parameter. In the ngModelChange, it is always the new value. But in the case of a change event, it is event data. The event data is an object containing data about the event. We need to use the target.value to access the value.

NgModelChange Vs Change

| **NgModelChange** | **Change** |
| --- | --- |
| NgModelChange is Angular specific event | Change is a DOM Event and has nothing to do with the Angular. |
| We must use the ngModelChange along with the ngModel directive | You can use change event on <input>, <select>, and <textarea> form elements. |
| ngModelChange event passes new value | Change event passes event parameter, Use the target.value to access the new value |
| ngModelChange will trigger with each input change | Change event fires when you remove the focus from input text after changing the content. |

What is a Child/Nested Component

The Angular follows component-based Architecture, where each [Angular Component](https://www.tektutorialshub.com/angular/angular-component/) manages a specific task or workflow. Each [Component](https://www.tektutorialshub.com/angular/angular-component/) is an independent block of the reusable unit.

In real life, angular applications will contain many [Angular Components](https://www.tektutorialshub.com/angular/angular-component/). The task of the root component is to just host these child components. These child components, in turn, can host the more child components creating a Tree-like structure called Component Tree.

In this tutorial, we will learn how to create a Child or nested components and host it in the App Component.

Create a new application

Create a new Angular application using the following command

|  |  |
| --- | --- |
| 1  2  3 | ng new childComponent |

Run the app and verify everything is ok.

|  |  |
| --- | --- |
| 1  2  3 | ng serve |

How to add Child Component

Create the Child Component. In the child Component, metadata specify the selector to be used

Import the Child Component in the module class and declare it in declaration Array

Use the CSS Selector to specify in the Parent Component Template, where you want to display the Child Component

Adding a Child Component in Angular

Now, let us add a Child Component to our project. In our child component, let us display a list of customers.

Create the Child Component

Creating the Child Component is no different from creating any other Parent Component. But, first, we need a customer class

Customer Class

Go to the app folder and create a file and name it as customer.ts. Copy the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | export class Customer {      customerNo: number;    name:string ;    address:string;    city:string;    state:string;    country:string;    } |

Note that we have used the export keyword. This enables us to use the above class in our components by importing it.

Create Child Component

In the app folder and create a new file and name it as customer-list.component.ts.

The code for component-list.component.ts is shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import { Component } from '@angular/core';  import { Customer } from './customer';    @Component({    selector: 'customer-list',    templateUrl: './customer-list.component.html'  })  export class CustomerListComponent  {    customers: Customer[] = [        {customerNo: 1, name: 'Rahuld Dravid', address: '', city: 'Banglaore', state: 'Karnataka', country: 'India'},      {customerNo: 2, name: 'Sachin Tendulkar', address: '', city: 'Mumbai', state: 'Maharastra', country: 'India'},      {customerNo: 3, name: 'Saurrav Ganguly', address: '', city: 'Kolkata', state: 'West Bengal', country: 'India'},      {customerNo: 4, name: 'Mahendra Singh Dhoni', address: '', city: 'Ranchi', state: 'Bihar', country: 'India'},      {customerNo: 5, name: 'Virat Kohli', address: '', city: 'Delhi', state: 'Delhi', country: 'India'},      ]  } |

First, we import the required modules & classes. Our component requires Customer class, hence we import it along with the Component.

|  |  |
| --- | --- |
| 1  2  3  4 | import { Component } from '@angular/core';  import { Customer } from './customer'; |

The next step is to add the [@Component directive.](https://www.tektutorialshub.com/angular/angular-component/) The selector clause has the value customer-list. We need to use this in our parent view to display our view. The templateURL is customer-list.component.html, which we yet to build.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Component({    selector: 'customer-list',    templateUrl: './customer-list.component.html'  }) |

The last step is to create the [Component](https://www.tektutorialshub.com/angular/angular-component/) class. We name it as CustomerListComponent. The class consists of a single property, which is a collection of customers. We initialize the customers collection with some default values. In real-life situations, you will make use of the [HTTP Client](https://www.tektutorialshub.com/angular/angular-httpclient/) to get the data from the back end server.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | export class CustomerListComponent  {    customers: Customer[] = [        {customerNo: 1, name: 'Rahuld Dravid', address: '', city: 'Banglaore', state: 'Karnataka', country: 'India'},      {customerNo: 2, name: 'Sachin Tendulkar', address: '', city: 'Mumbai', state: 'Maharastra', country: 'India'},      {customerNo: 3, name: 'Saurrav Ganguly', address: '', city: 'Kolkata', state: 'West Bengal', country: 'India'},      {customerNo: 4, name: 'Mahendra Singh Dhoni', address: '', city: 'Ranchi', state: 'Bihar', country: 'India'},      {customerNo: 5, name: 'Virat Kohli', address: '', city: 'Delhi', state: 'Delhi', country: 'India'},      ]  } |

Creating the View

The next step is to create the View to display the list of customer. Go to the app folder and create the file with the name customer-list.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | <h2>List of Customers</h2>    <table class='table'>    <thead>      <tr>        <th>No</th>        <th>Name</th>        <th>Address</th>        <th>City</th>        <th>State</th>      </tr>    </thead>    <tbody>      <tr \*ngFor="let customer of customers;">        <td>{{customer.customerNo}}</td>        <td>{{customer.name}}</td>        <td>{{customer.address}}</td>        <td>{{customer.city}}</td>        <td>{{customer.state}}</td>      </tr>    </tbody>  </table> |

To iterate through the Customer collection, we have used the [ngFor Directive](https://www.tektutorialshub.com/angular/angular-ngfor-directive/) provided by the Angular. We have a separate tutorial, which discusses the [ngFor Directive](https://www.tektutorialshub.com/angular/angular-ngfor-directive/)

The syntax for ngFor directive starts with \*ngFor. The \* indicates that it is a structural directive. i.e a directive that adds or removes the HTML elements to or from the DOM.

The expression let customer of customers is assigned to \*ngFor. The let clause assigns the instance of customer object from the Customers collection to the template reference variable or local variable customer.

We use the template reference variable customer is to build the template to display the details of the customer to the user. The ngFor directive is applied to the tr element of the table. The Angular repeats everything inside the tr element in the DOM tree.

{{customer.customerNo}} is stands for interpolation in Angular. Angular evaluates everything within the {{ }} and replaces the string with the result.

Import the Child Component in the Module

Every [Component](https://www.tektutorialshub.com/angular/angular-component/), [directive](https://www.tektutorialshub.com/angular/angular-directives/), [pipes](https://www.tektutorialshub.com/angular/angular-pipes/) we build must belong to an [Angular Module](https://www.tektutorialshub.com/angular/angular-modules/). To do that we need to register our component in the Module. A Component, directive, pipes cannot be part of more than one module.

The [Angular Modules](https://www.tektutorialshub.com/angular/angular-modules/) (or [NgModules](https://www.tektutorialshub.com/angular/angular-modules/)) are Angular ways of organizing related [components](https://www.tektutorialshub.com/angular/angular-component/), directives, pipes and services, etc into a group. To add a component to a module, you need to declare it in the declarations metadata of the Angular Module

Angular creates a top-level root module (AppModule in file app.module.ts) when we create a new Angular app. That is where we need to register our CustomerListComponent

Open the app.module.ts under the app folder and update the code as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';    import {NgbModule} from '@ng-bootstrap/ng-bootstrap';    import { AppComponent } from './app.component';  import {CustomerListComponent} from './customer-list.component';    @NgModule({    declarations: [      AppComponent, CustomerListComponent    ],    imports: [      BrowserModule,NgbModule.forRoot()    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

Registering the component or directive in the module requires two steps

First, import it

|  |  |
| --- | --- |
| 1  2  3  4 | import { AppComponent } from './app.component';  import {CustomerListComponent} from './customer-list.component'; |

And, then declare it in declaration array

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @NgModule({  declarations: [     AppComponent, CustomerListComponent  ] |

Applies to: Angular 5 to the latest edition i.e. Angular 8, Angular 9. Angular 10, Angular 11

Tell angular where to display the component

Finally, we need to inform the Angular, where to display the child Component

We want our child Component as the child of the AppComponent. Open the app.component.html and add the following template

|  |  |
| --- | --- |
| 1  2  3  4  5 | <h1>{{title}}. </h1>    <customer-list></customer-list> |

The @Component decorator of the CustomerListComponent , we used the customer-list as the selectorin the metadata for the component. This CSS selector name must match the element tag that specified within the parent component’s template.

|  |  |
| --- | --- |
| 1  2  3 | <customer-list></customer-list> |

Run the application from the command line using **ng Serve**

# Directives

The [Angular directive](https://www.tektutorialshub.com/angular/angular-directives/) helps us to manipulate the DOM. You can change the appearance, behavior, or layout of a DOM element using the directives. They help you to extend HTML. The [Angular directives](https://www.tektutorialshub.com/angular/angular-directives/) are classified into three categories based on how they behave.  They are Component, Structural and Attribute Directives

The [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/) is an Angular structural directive, which repeats a portion of the HTML template once per each item from an iterable list (Collection). The [ngSwitch](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/) allows us to Add/Remove DOM Element. It is similar to the switch statement of Javascript.  The [ngIf](https://www.tektutorialshub.com/angular/angular-ngif-directive/) allows us to Add/Remove DOM Element.

The [ngClass](https://www.tektutorialshub.com/angular/angular-ngclass-directive/) Directive is an Angular Attribute Directive, which allows us to add or remove CSS classes to an HTML element. The [ngStyle](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/) directive allows you to modify the style of an HTML element using the expression.  Using the [ngStyle](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/) you can dynamically change the style of your HTML element.

[Angular Directives](https://www.tektutorialshub.com/angular/angular-directives/)

[ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/)

[ngSwitch](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/)

[ngIf](https://www.tektutorialshub.com/angular/angular-ngif-directive/)

[ngClass](https://www.tektutorialshub.com/angular/angular-ngclass-directive/)

[ngStyle](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/)

[ngFor Trackby](https://www.tektutorialshub.com/angular/angular-track-by-to-improve-ngfor-performance/)

[Custom Directive](https://www.tektutorialshub.com/angular/custom-directive-in-angular/)

What is Angular Directive

The Angular directive helps us to manipulate the DOM. You can change the appearance, behavior, or layout of a DOM element using the Directives. They help you to extend HTML

There are three kinds of directives in Angular:

Component Directive

Structural directives

Attribute directives

Component Directive

Components are special directives in Angular. They are the directive with a template (view) We covered [how to create Components in Angular](https://www.tektutorialshub.com/angular/angular-component/) tutorial.

Ad by Valueimpression

Structural Directives

Structural directives can change the DOM layout by adding and removing DOM elements. All structural Directives are preceded by Asterix symbol

Commonly used structural directives

ngFor

The [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/) is an Angular structural directive, which repeats a portion of the HTML template once per each item from an iterable list (Collection). The [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/) is similar to [ngRepeat](https://docs.angularjs.org/api/ng/directive/ngRepeat) in AngularJS

Example of ngFor

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <tr \*ngFor="let customer of customers;">      <td>{{customer.customerNo}}</td>      <td>{{customer.name}}</td>      <td>{{customer.address}}</td>      <td>{{customer.city}}</td>      <td>{{customer.state}}</td>  </tr>  ﻿ |

ngSwitch

The [ngSwitch](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/) directive lets you add/remove HTML elements depending on a match expression. [ngSwitch](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/) directive used along with [ngSwitchCase](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/#ngswitchcase) and [ngSwitchDefault](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/#ngswitchdefault)

The example of ngSwitch

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <div [ngSwitch]="Switch\_Expression">      <div \*ngSwitchCase="MatchExpression1”> First Template</div>      <div \*ngSwitchCase="MatchExpression2">Second template</div>      <div \*ngSwitchCase="MatchExpression3">Third Template</div>      <div \*ngSwitchCase="MatchExpression4">Third Template</div>      <div \*ngSwitchDefault?>Default Template</div>  </div> |

You can read more about the [Angular ngSwitch Directive](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/) tutorial.

ngIf

The [ngIf](https://www.tektutorialshub.com/angular/angular-ngif-directive/) Directives is used to add or remove HTML elements based on an expression. The expression must return a boolean value. If the expression is false then the element is removed, else the element is inserted

Example of ngIf

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div \*ngIf="condition">      This is shown if condition is true  </div> |

You can read more about [Angular ngIf Directive](https://www.tektutorialshub.com/angular/angular-ngif-directive/) tutorial.

Attribute Directives

An Attribute or style directive can change the appearance or behavior of an element.

Commonly used Attribute directives

ngModel

The ngModel directive is used the achieve the[two-way data binding](https://www.tektutorialshub.com/angular/angular-data-binding/). We have covered ngModel directive in [Data Binding in Angular Tutorial](https://www.tektutorialshub.com/angular/angular-data-binding/)

ngClass

The [ngClass](https://www.tektutorialshub.com/angular/angular-ngclass-directive/) is used to add or remove the CSS classes from an HTML element. Using the [ngClass](https://www.tektutorialshub.com/angular/angular-ngclass-directive/) one can create dynamic styles in HTML pages

Example of ngClass

|  |  |
| --- | --- |
| 1  2  3 | <div [ngClass]="'first second'">...</div> |

ngStyle

[ngStyle](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/)is used to change the multiple style properties of our HTML elements. We can also bind these properties to values that can be updated by the user or our components.

Example of ngStyle

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div [ngStyle]="{'color': 'blue', 'font-size': '24px', 'font-weight': 'bold'}">      some text  </div> |

Building Custom Directives

You can also build custom directives in Angular. The Process is to create a JavaScript class and apply the **@Directive** attribute to that class. You can write the desired behavior in the class.

Syntax of ngFor

The syntax for the ngFor is as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5 | <html-element ngFor="let <item> of <items>;”>       <html-Template></html-Template>  </html-element> |

<html-element>:  
is the element on which we apply ngFor directive. it repeats the <html-element> .. </html-element> for each item of the collection.

\*ngFor :  
The syntax starts with \*ngFor. The \* here represents the Angular Template Syntax.

let <item> of <items>;  
item is the Template input variable. It represents the currently iterated item from the <items>. <items> is a collection, which we need to show it to the user. It is usually a property on your component class and can be anything that you can iterate over. (Usually an array)

The scope of the item is within the <html-element>..</html-element>. You can access it anywhere within that, but not outside of it.

ngFor Example

Now let use see how to use ngFor using a example.

Create a new angular Application. If you are new to angular, then you should read [Angular create new project](https://www.tektutorialshub.com/angular/angular-create-first-application/). We are using [bootstrap 4](https://getbootstrap.com/) to style our application. Hence you can add the following line to index.html

|  |  |
| --- | --- |
| 1  2  3 | <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm" crossorigin="anonymous"> |

Open the app.component.ts and add the following code. The code contains a list of Top 10 movies. Let us build a template to display the movies using ngFor.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    title: string ="Top 5 Movies" ;        movies: Movie[] =[      {title:'Zootopia',director:'Byron Howard, Rich Moore',cast:'Idris Elba, Ginnifer Goodwin, Jason Bateman',releaseDate:'March 4, 2016'},    {title:'Batman v Superman: Dawn of Justice',director:'Zack Snyder',cast:'Ben Affleck, Henry Cavill, Amy Adams',releaseDate:'March 25, 2016'},    {title:'Captain American: Civil War',director:'Anthony Russo, Joe Russo',cast:'Scarlett Johansson, Elizabeth Olsen, Chris Evans',releaseDate:'May 6, 2016'},    {title:'X-Men: Apocalypse',director:'Bryan Singer',cast:'Jennifer Lawrence, Olivia Munn, Oscar Isaac',releaseDate:'May 27, 2016'},    {title:'Warcraft',director:'Duncan Jones',cast:'Travis Fimmel, Robert Kazinsky, Ben Foster',releaseDate:'June 10, 2016'},  ]    CompositeKey (index,item){      return item.title + item.director ;     }  }    class Movie {    title : string;    director : string;    cast : string;    releaseDate : string;  } |

Using ngFor

To use ngFor,

Create a block of HTML elements, which can display a single movie.

Use the ngFor to repeat the block for each movie in the movies.

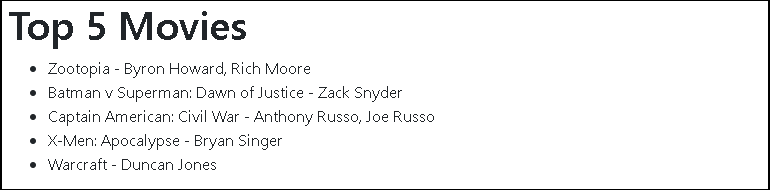
Open the app.component.html and add the following code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <h1> {{title}} </h1>      <ul>      <li \*ngFor="let movie of movies">        {{ movie.title }} - {{movie.director}}      </li>    </ul> |

We use the ul to display the movies. The li element displays a single movie. We need to repeat the li for each movie. Hence we apply the ngFor on the li element.

let movie of movies will iterate over the movies collection, which is a property on the component class. movie is the Template input variable, which represents the currently iterated movie from the collection. We use the [Angular Interpolation](https://www.tektutorialshub.com/angular/interpolation-in-angular/) to display the movie title & name of the director

Here is the output



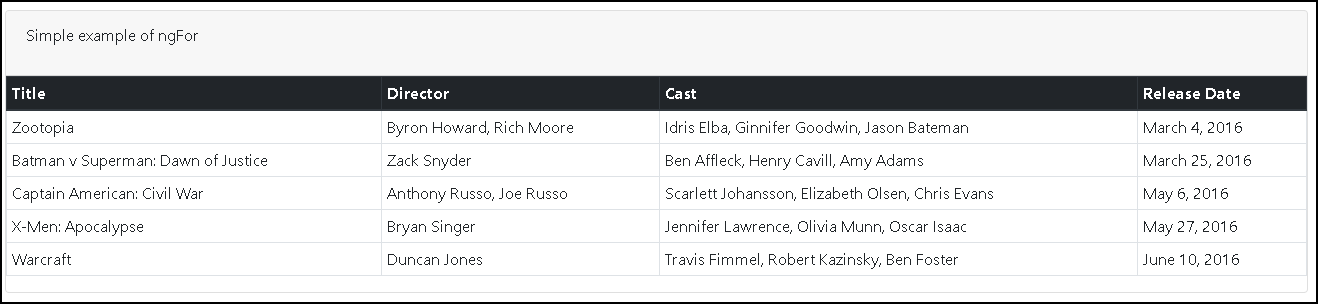
The Angular generates the following code. You can see li element for every movie.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <ul \_ngcontent-gop-c0="">    <li \_ngcontent-gop-c0=""> Zootopia - Byron Howard, Rich Moore </li>    <li \_ngcontent-gop-c0=""> Batman v Superman: Dawn of Justice - Zack Snyder </li>    <li \_ngcontent-gop-c0=""> Captain American: Civil War - Anthony Russo, Joe Russo </li>    <li \_ngcontent-gop-c0=""> X-Men: Apocalypse - Bryan Singer </li>    <li \_ngcontent-gop-c0=""> Warcraft - Duncan Jones </li>  </ul> |

Similarly, you can use the table element to display the movies as shown below. Here we need to repeat the tr element for each movie. Hence apply the directive on tr

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | <div class='panel panel-primary'>      <div class='panel-heading'>          {{title}}      </div>        <div class='panel-body'>          <div class='table-responsive'>              <table class='table'>                  <thead>                      <tr>                          <th>Title</th>                          <th>Director</th>                          <th>Cast</th>                          <th>Release Date</th>                      </tr>                  </thead>                  <tbody>                      <tr \*ngFor="let movie of movies;">                          <td>{{movie.title}}</td>                          <td>{{movie.director}}</td>                          <td>{{movie.cast}}</td>                          <td>{{movie.releaseDate}}</td>                      </tr>                  </tbody>              </table>          </div>      </div>  </div> |

Here is the output



Nested Array

The following example shows how to use the ngFor in a nested array. The employees array has nested skills array.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | employees = [      {        name: "Rahul", email: "rahul@gmail.com",        skills: [{ skill: 'Angular', exp: '2' },{ skill: 'Javascript', exp: '7' },{ skill: 'TypeScript', exp: '3' }        ]      },      {        name: "Sachin", email: "sachin@gmail.com",        skills: [{ skill: 'Angular', exp: '1' },{ skill: 'Android', exp: '3' },{ skill: 'React', exp: '2' }        ]      },      {        name: "Laxmna", email: "laxman@gmail.com",        skills: [{ skill: 'HTML', exp: '2' },{ skill: 'CSS', exp: '2' },{ skill: 'Javascript', exp: '1' }        ]      }    ] |

Inside the main loop, use the local variable employee to get the list of skills and loop through it using \*ngFor="let skill of employee.skills;"

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | <div class='card'>    <div class='card-header'>      <p>Nested Array</p>    </div>      <div class='table-responsive'>      <table class='table table-bordered table-sm '>        <thead class="thead-dark">          <tr>            <th>Name</th>            <th>Mail ID</th>            <th>Skills</th>          </tr>        </thead>        <tbody>          <tr \*ngFor="let employee of employees;">            <td>{{employee.name}}</td>            <td>{{employee.email}}</td>            <td>              <table class='table table-sm '>                <tbody>                  <tr \*ngFor="let skill of employee.skills;">                    <td>{{skill.skill}}</td>                    <td>{{skill.exp}}</td>                  </tr>                </tbody>              </table>              </td>          </tr>        </tbody>      </table>    </div>  </div> |

**Local Variables**

ngFor exposes several values, which help us to fine-tune display. We assign these values to the local variable and use it in our template

The list of exported values provided by ngFor directive

index: number: The zero-based index of the current element in the collection.

count: number: The total no of items in the collection

first: boolean: True when the item is the first item in the collection.

last: boolean: Is set to True, when the item is the last item in the collection.

even: boolean: True when the item has an even index in the collection.

odd: boolean: is set to True when the item has an odd index in the collection.

Finding the Index

To Find the index, we create another local variable i and use the let to make it equal to index.

|  |  |
| --- | --- |
| 1  2  3 | let i=index; |

The following code shows the list of movies along with the index.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <tr \*ngFor="let movie of movies; let i=index;">      <td> {{i}} </td>      <td>{{movie.title}}</td>      <td>{{movie.director}}</td>      <td>{{movie.cast}}</td>      <td>{{movie.releaseDate}}</td>  </tr> |

Formatting odd & even rows

We can use the odd & even values to format the odd & even rows alternatively. To do that create two local variables o & e. Assign the values of odd & even values to these variables using the let statement. Then use the[ngClass](https://www.tektutorialshub.com/angular/angular-ngclass-directive/) to change the class name to either odd or even. The example code is shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <tr \*ngFor="let movie of movies; let i=index; let o= odd; let e=even;"  [ngClass]="{ odd: o, even: e }">      <td> {{i}} </td>      <td>{{movie.title}}</td>      <td>{{movie.director}}</td>      <td>{{movie.cast}}</td>      <td>{{movie.releaseDate}}</td>  </tr> |

Add the appropriate background color to the odd and even classes as shown below in app.component.css

|  |  |
| --- | --- |
| 1  2  3  4 | .even { background-color: azure; }  .odd { background-color: floralwhite; } |

First and the Last element of a list

Similarly, you can use the first & last values to style the first & last element. The code below will add a CSS classes first & last to the first and last movie using the [ngClass](https://www.tektutorialshub.com/angular/angular-ngclass-directive/).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | <div class='table-responsive'>    <table class='table table-bordered table-sm '>      <thead class="thead-dark">        <tr>          <th>Index</th>          <th>Title</th>          <th>Director</th>          <th>Cast</th>          <th>Release Date</th>        </tr>      </thead>      <tbody>        <tr \*ngFor="let movie of movies; let i=index; let first= first; let last=last;" [ngClass]="{ first: first, last: last }">          <td> {{i}} </td>          <td>{{movie.title}}</td>          <td>{{movie.director}}</td>          <td>{{movie.cast}}</td>          <td>{{movie.releaseDate}}</td>        </tr>      </tbody>    </table>  </div> |

Remember to add the CSS classes to app.component.css

|  |  |
| --- | --- |
| 1  2  3  4 | .first { background-color: azure; }  .last { background-color: floralwhite; } |

Track By

The angular includes Track By clause, just like AngularJS did. Track By clause allows you to specify your own key to identify objects.

Angular uses the object identity to compare the elements in the collection to the DOM nodes. Hence when you add an item or remove an item, the Angular will track it and update only the modified items in the DOM. It does not render the entire list.

But this fails if we update the list from the back end server. That is because the retrieved objects cannot be compared with the existing objects in the list as the reference has changed. The Angular to simply remove these elements from DOM and recreates the new elements from the new data. This has a huge performance implication.

[Angular trackBy](https://www.tektutorialshub.com/angular/angular-track-by-to-improve-ngfor-performance/)clause eliminates this problem, by telling angular how to identify the similar elements. The Angular will use the value returned by the trackBy function to match the elements returned by the database and updates the DOM Elements without recreating them.

We should always specify the primary key or unique key as the trackBy clause.

Example

In our movie list example, let us make the title of the movie as the identifier.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <tr \*ngFor="let movie of movies; trackBy:trackByFn;">      <td>{{movie.title}}</td>      <td>{{movie.director}}</td>      <td>{{movie.cast}}</td>      <td>{{movie.releaseDate}}</td>  </tr> |

In the component class create a trackByFn. It gets index and the current item as its argument. It should return the unique id

|  |  |
| --- | --- |
| 1  2  3  4  5 | trackByFn(index, item) {      return item.title;    } |

Angular ngSwitch Directive

The ngSwitch is an [Angular directive](https://www.tektutorialshub.com/angular/angular-directives/), which allows us to display one or more DOM elements based on some pre-defined condition.

The following is the syntax of ngSwitch. It contains three separate directives. ngSwitch, ngSwitchCase & ngSwitchDefault.

Syntax

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <container\_element [ngSwitch]="switch\_expression">      <inner\_element \*ngSwitchCase="match\_expresson\_1">...</inner\_element>      <inner\_element \*ngSwitchCase="match\_expresson\_2">...</inner\_element>      <inner\_element \*ngSwitchCase="match\_expresson\_3">...</inner\_element>      <inner\_element \*ngSwitchDefault>...</element>  </container\_element> |

ngSwitch

ngSwitch is bound to container\_element like div etc. We assign a switch-expression to the ngSwitch via [property binding](https://www.tektutorialshub.com/angular/property-binding-in-angular/) syntax. Angular evaluates the switch\_expression at runtime and based on its value displays or removes the elements from the DOM.

ngSwitchCase

ngSwitchCase is bound to an inner\_element, which we must place inside the container\_element. We use \* (Asterix symbol), because it is a[structural directive](https://www.tektutorialshub.com/angular/angular-directives/#structural-directives). We also assign a match\_expression, which Angular evaluates at runtime. The Angular displays the inner\_element only when the value of the match\_expression matches the value of the switch\_expression else it is removed from the DOM.

If there is more than one match, then it displays all of them.

Note that the ngSwitchCase does not hide the element, but removes them from the DOM.

ngSwitchDefault

ngSwitchDefault is also bound to an inner\_element, which we must place inside the container\_element. But it does not have any match\_expression. If none of the ngSwitchCase match\_expression matches the switch\_expression, then the angular displays the element attached to the ngSwitchDefault

You can place ngSwitchDefault anywhere inside the container element and not necessarily at the bottom.

You are free to add more than one ngSwitchDefault directive. Angular displays all of them.

Important Points

You must place ngSwitchCase & ngSwitchDefault inside the ngSwitch directive

Angular displays every element, that matches the switch\_expression

If there are no matches, angular displays all the elements, which has ngSwitchDefault directive

You can place one or more than one ngSwitchDefault anywhere inside the container element and not necessarily at the bottom.

Any element within the ngSwitch statement but outside of any NgSwitchCase or ngSwitchDefault directive is displayed as it is.

The elements are not hidden but removed from the DOM.

Angular uses loose equality checks to compare the ngSwitchCase expression with the ngSwitch expression. This means that the empty string "" matches 0.

You can share the template between multiple ngSwitchCase using the ngTemplateOutlet

ngSwitch Example

[Create a new Angular project](https://www.tektutorialshub.com/angular/angular-create-first-application/). Add the bootstrap CSS to the project by adding the following in the index.html

|  |  |
| --- | --- |
| 1  2  3 | <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm" crossorigin="anonymous"> |

Component Class

Create a variable num in your [Angular Component](https://www.tektutorialshub.com/angular/angular-component/) class

|  |  |
| --- | --- |
| 1  2  3 | num: number= 0; |

Template

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | <div class='card'>    <div class='card-header'>      ngSwitch Example    </div>    <div class="card-body">      Input string : <input type='text' [(ngModel)]="num" />        <div [ngSwitch]="num">        <div \*ngSwitchCase="'1'">One</div>        <div \*ngSwitchCase="'2'">Two</div>        <div \*ngSwitchCase="'3'">Three</div>        <div \*ngSwitchCase="'4'">Four</div>        <div \*ngSwitchCase="'5'">Five</div>        <div \*ngSwitchDefault>This is Default</div>      </div>    </div>  </div> |

Now let us examine the code in detail

|  |  |
| --- | --- |
| 1  2  3 | Input string : <input type='text' [(ngModel)] ="num"/> |

We bind the num variable to the input box.

|  |  |
| --- | --- |
| 1  2  3 | <div [ngSwitch]="num"> |

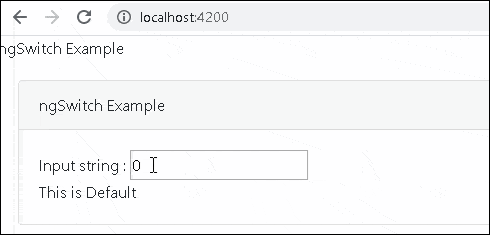
We attach the ngSwitch directive to the div element, then bind it to the expression num.

|  |  |
| --- | --- |
| 1  2  3 | <div \*ngSwitchCase="'1'">One</div> |

Next, we have a few ngSwitchCase directives attached to the div element with matching expressions “1”,”2” etc. Whenever the num matches these expressions, the ngSwitchCase displays the element attached to it else it removes it from DOM.

|  |  |
| --- | --- |
| 1  2  3 | <div \*ngSwitchDefault>This is Default</div> |

The ngSwithcDefault does not take any expression, but it is displays only when all other ngSwitchCase match expressions fail.



More Examples

The following uses the array of objects instead of a variable.

Component class

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class item {      name: string;      val: number;  }    export class AppComponent  {      items: item[] = [{name: 'One', val: 1}, {name: 'Two', val: 2}, {name: 'Three', val: 3}, {name: 'Four', val: 3}, {name: 'Five', val: 3}];      selectedValue1: string= 'One';      selectedValue2: string= 'One';      selectedValue3: string= 'One';      selectedValue4: string= 'One';  } |

Template

Note that we have two matches for \*ngSwitchCase="'Two'". ngSwitchcase renders both.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | <div class='card'>    <div class='card-header'>      Multiple ngSwitchCase with same condition    </div>      <div class="card-body">      <select [(ngModel)]="selectedValue1">        <option \*ngFor="let item of items;" [value]="item.name">{{item.name}}</option>      </select>        <div [ngSwitch]="selectedValue1">        <p \*ngSwitchCase="'One'">One is Selected</p>        <p \*ngSwitchCase="'Two'">Two is Selected</p>        <p \*ngSwitchCase="'Two'">Two Again used in another element</p>        <p \*ngSwitchDefault>This is Default</p>      </div>    </div>    </div> |

You can also make use of ng-template directly instead of \*ngSwitchCase. In fact \*ngSwitchCase="'One'" is a shortcut to ng-template [ngSwitchCase]="'One'".

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | <div class='card'>    <div class='card-header'>      ngSwitch using ng-template    </div>      <div class="card-body">        <select [(ngModel)]="selectedValue2">        <option \*ngFor="let item of items;" [value]="item.name">{{item.name}}</option>      </select>        <div [ngSwitch]="selectedValue2">        <ng-template [ngSwitchCase]="'One'">One is Selected</ng-template>        <ng-template [ngSwitchCase]="'Two'">Two is Selected</ng-template>        <ng-template ngSwitchDefault>This is Default</ng-template>      </div>      </div>  </div> |

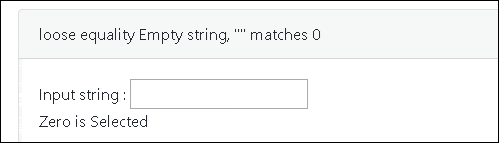
More than one ngSwitchDefault. Works perfectly ok.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | <div class='card'>    <div class='card-header'>      Multiple ngSwitchDefault Directives    </div>      <div class="card-body">      <select [(ngModel)]="selectedValue3">        <option \*ngFor="let item of items;" [value]="item.name">{{item.name}}</option>      </select>          <div [ngSwitch]="selectedValue3">        <div \*ngSwitchCase="'One'">One is Selected</div>        <div \*ngSwitchDefault>This is Default 1</div>        <div \*ngSwitchCase="'Two'">Two is Selected</div>        <div \*ngSwitchDefault>This is Default 2</div>      </div>        </div>  </div> |

Loose Equality Checks

Angular uses loose equality checks to compare the ngSwitchCase expression with the ngSwitch expression. This means that the empty string "" matches 0.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | <div class='card'>    <div class='card-header'>      loose equality Empty string, "" matches 0    </div>      <div class="card-body">        Input string : <input type='text' [(ngModel)]="num" />        <div [ngSwitch]="num">        <div \*ngSwitchCase="0">Zero is Selected</div>        <div \*ngSwitchCase="1">One is Selected</div>        <div \*ngSwitchCase="2">Two is Selected</div>        <div \*ngSwitchDefault>This is Default 2</div>      </div>        </div>  </div> |



Multiple / Sharing ngSwitchCase

You may also want to share the template between two values. For Example One Template for the values One & Two & another template for values Three & Four. One option is to repeat the template under each switch case. Or you can also make use of the ngTemplateOutlet to share the template as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | <div class='card'>    <div class='card-header'>      Sharing Templates    </div>      <div class="card-body">        <select [(ngModel)]="selectedValue4">        <option \*ngFor="let item of items;" [value]="item.name">{{item.name}}</option>      </select>          <ng-container [ngSwitch]="selectedValue4">          <ng-container \*ngSwitchCase="'One'">          <ng-container \*ngTemplateOutlet="sharedTemplate12"></ng-container>        </ng-container>        <ng-container \*ngSwitchCase="'Two'">          <ng-container \*ngTemplateOutlet="sharedTemplate12"></ng-container>        </ng-container>          <ng-container \*ngSwitchCase="'Three'">          <ng-container \*ngTemplateOutlet="sharedTemplate34"></ng-container>        </ng-container>          <ng-container \*ngSwitchCase="'Four'">          <ng-container \*ngTemplateOutlet="sharedTemplate34"></ng-container>        </ng-container>          <ng-template #sharedTemplate12>Shared between 1,2</ng-template>        <ng-template #sharedTemplate34>Shared between 3,4</ng-template>        <ng-container \*ngSwitchDefault>Default Template</ng-container>        </ng-container>      </div>  </div> |

ngIf Syntax

|  |  |
| --- | --- |
| 1  2  3  4  5 | <p \*ngIf="condition">      content to render, when the condition is true  </p> |

The ngIf is attached to a DOM element ( p element in the above example). ngIf is a [structural directive](https://www.tektutorialshub.com/angular/angular-directives/#structural-directives), which means that you can add it to any element like div, p, h1, component selector, etc. Like all [structural directive](https://www.tektutorialshub.com/angular/angular-directives/#structural-directives), it is prefixed with \* asterisk

We bind the \*ngIf to an expression (a condition in the above example). The expression is then evaluated by the ngIf directive. The expression must return either true or false.

If the expression evaluates to false then the Angular removes the entire element from the DOM. If true it will insert the element into the DOM.

Hidden attribute Vs ngIf

|  |  |
| --- | --- |
| 1  2  3  4  5 | <p [hidden]="condition">      content to render, when the condition is true  </p> |

The above achieves the same thing, with one vital difference.

ngIf does not hide the DOM element. It removes the entire element along with its subtree from the DOM. It also removes the corresponding state freeing up the resources attached to the element.  
  
hidden attribute does not remove the element from the DOM. But just hides it.

The difference between [hidden]='false' and \*ngIf='false' is that the first method simply hides the element. The second method with ngIf removes the element completely from the DOM.

By using the **Logical NOT** (!), you can mimic the else condition as shown here.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <p \*ngIf="!condition">      content to render, when the condition is false  </p> |

The better solution is to use the optional else block as shown in the next paragraph.

Condition

The condition can be anything. It can be a property of the component class. It can be a method in the component class. But it must evaluate to true/false. The ngIf directive tries to coerce the value to Boolean.

ngIf else

The ngIf allows us to define optional else block using the ng-template

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <div \*ngIf="condition; else elseBlock">      content to render, when the condition is true  </div>    <ng-template #elseBlock>      content to render, when the condition is false  </ng-template> |

The expression starts with a condition followed by a semicolon.

Next, we have else clause bound to a template named elseBlock. The template can be defined anywhere using the ng-template. Place it right after ngIf for readability.

When the condition evaluates to false, then the ng-template with the name #elseBlock is rendered by the ngIf Directive.

ngIf then else

You can also define then else block using the ng-template

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <div \*ngIf="condition; then thenBlock else elseBlock">      This content is not shown  </div>    <ng-template #thenBlock>      content to render when the condition is true.  </ng-template>    <ng-template #elseBlock>      content to render when condition is false.  </ng-template> |

Here, we have then clause followed by a template named thenBlock.

When the condition is true, the template thenBlock is rendered. If false, then the template elseBlock is rendered

ngIf Example

Create a new Angular project by running the command ng new ngIf

Component Class

Create a boolean variable showMe in your app.component.ts class as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    title: string = 'ngIf Example' ;    showMe: boolean;  } |

Template

Copy the following code to the app.component.html.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | <h1>Simple example of ngIf </h1>      <div class="row">    Show <input type="checkbox" [(ngModel)]="showMe" />  </div>    <h1>ngIf </h1>    <p \*ngIf="showMe">    ShowMe is checked  </p>  <p \*ngIf="!showMe">    ShowMe is unchecked  </p>    <h1>ngIf Else</h1>    <p \*ngIf="showMe; else elseBlock1">    ShowMe is checked  </p>    <ng-template #elseBlock1>    <p>ShowMe is unchecked Using elseBlock</p>  </ng-template>    <h1>ngIf then else</h1>    <p \*ngIf="showMe; then thenBlock2 else elseBlock2">    This is not rendered  </p>    <ng-template #thenBlock2>    <p>ShowMe is checked Using thenblock</p>  </ng-template>    <ng-template #elseBlock2>    <p>ShowMe is unchecked Using elseBlock</p>  </ng-template>    <h1>using hidden </h1>    <p [hidden]="showMe">      content to render, when the condition is true  using hidden property binding  </p>    <p [hidden]="!showMe">      content to render, when the condition is false. using hidden property binding  </p> |

Now let us examine the code in detail

|  |  |
| --- | --- |
| 1  2  3 | Show <input type="checkbox" [(ngModel)] ="showMe"/> |

This is a simple checkbox bound to showMe variable in the component

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div \*ngIf="showMe">      ShowMe is checked  </div> |

The ngIf directive is attached to the div element. It is then bound to the expression “showMe”. The expression is evaluated and if it is true, then the div element is added to the DOM else it is removed from the DOM.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <p \*ngIf="showMe; else elseBlock1">    ShowMe is checked  </p>    <ng-template #elseBlock1>    <p>ShowMe is checked Using elseBlock</p>  </ng-template> |

If else example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <p \*ngIf="showMe; then thenBlock2 else elseBlock2">    This is not rendered  </p>    <ng-template #thenBlock2>    <p>ShowMe is checked Using thenblock</p>  </ng-template>    <ng-template #elseBlock2>    <p>ShowMe is unchecked Using elseBlock</p>  </ng-template> |

If then else example. Note that the content of p element, to which ngIf is attached is never rendered

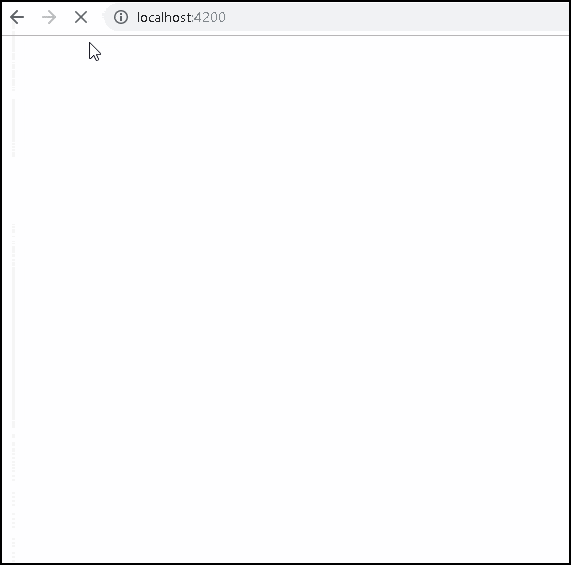
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <h1>using hidden </h1>    <p [hidden]="showMe">      content to render, when the condition is true  using hidden property binding  </p>    <p [hidden]="!showMe">      content to render, when the condition is false. using hidden property binding  </p> |

The property binding on the hidden attribute. You can open the developer console and see that both the Angular renders both the elements. But mark one of them as visible and the other one as hidden.

Module

Import [FormsModule](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/) in app.module.ts as we are using [ngModal](https://www.tektutorialshub.com/angular/angular-data-binding/#two-way-binding) directive

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule } from '@angular/forms';    import { AppRoutingModule } from './app-routing.module';  import { AppComponent } from './app.component';    @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule,      AppRoutingModule,      FormsModule    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |



NgClass

The ngClass directive adds and removes CSS classes on an HTML element. The syntax of the ngClass is as shown below.

|  |  |
| --- | --- |
| 1  2  3 | <element [ngClass]="expression">...</element> |

Where

**element** is the DOM element to which class is being applied

**expression** is evaluated and the resulting classes are added/removed from the element. The expression can be in various formats like string, array or an object. Let us explore all of them with example

NgClass with a String

You can use the String as expression and bind it to directly to the ngClass attribute. If you want to assign multiple classes, then separate each class with space as shown below.

|  |  |
| --- | --- |
| 1  2  3 | <element [ngClass]="'cssClass1 cssClass2'">...</element> |

Example

Add the following classes to the app.component.css

|  |  |
| --- | --- |
| 1  2  3  4 | .red { color: red; }  .size20 { font-size: 20px; } |

Add the following to the app.template.html

|  |  |
| --- | --- |
| 1  2  3 | <div [ngClass]="'red size20'"> Red Text with Size 20px </div> |

The above example code adds the two CSS Classes red & size20 to the div element.

You can also use the ngClass without a square bracket. In that case, the expression is not evaluated but assigned directly to the class attribute. We also need to remove the double quote around the expression as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div class="row">      <div ngClass='red size20'>Red Text with Size 20px </div>  </div> |

NgClass with Array

You can achieve the same result by using an array instead of a string as shown below. The syntax for ngClass array syntax is as shown below

|  |  |
| --- | --- |
| 1  2  3 | <element [ngClass]="['cssClass1', 'cssClass2']">...</element> |

Example

All you need to change the template as shown below

|  |  |
| --- | --- |
| 1  2  3 | <div [ngClass]="['red','size20']">Red Text with Size 20px </div> |

NgClass with Object

You can also bind the ngClass to an object. Each property name of the object acts as a class name and is applied to the element if it is true. The syntax is as shown below

|  |  |
| --- | --- |
| 1  2  3 | <element [ngClass]="{'cssClass1': true, 'cssClass2': true}">...</element> |

Example of objects as CSS Classes

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div class="row">    <div [ngClass]="{'red':true,'size20':true}">Red Text with Size 20px</div>  </div> |

In the above example, an object is bound to the ngClass. The object has two properties red and size20. The property name is assigned to the div element as a class name.

Dynamically updating Class names

We can dynamically change the CSS Classes from the component.

Using strings

To do that first create a string variable cssStringVar in your component code and assign the class names to it as shown below.

|  |  |
| --- | --- |
| 1  2  3 | cssStringVar: string= 'red size20'; |

You can refer to the cssStringVar in your template as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div class="row">     <div [ngClass]="cssStringVar">Red Text with Size 20px : from component     </div>  </div> |

Using arrays

Instead of string variable, you can create a **array of string** as shown below.

|  |  |
| --- | --- |
| 1  2  3 | cssArray:string[]=['red','size20']; |

And, then use it in ngClass directive

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <div class="row">    <div [ngClass]="cssArray">      Red Text with Size 20px  : from CSS Array    </div>  </div> |

Using JavaScript object

Create a class as shown below in your component

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class CssClass {    red: boolean= true;    size20: boolean= true;  } |

Next, create the instance of the CssClass in the component as shown below. You can change the value of the property true as false dynamically

|  |  |
| --- | --- |
| 1  2  3 | cssClass: CssClass = new CssClass(); |

And then refer to the cssClass in your template.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div class="row">    <div [ngClass]="cssClass"> Red Text with Size 20px : from component as object</div>  </div> |

Trackby in ngFor

We usengFor to display a iterable items like array in a list or tabular format. For Example the following code iterates over the movies collection and displays each movie inside an ul

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <ul>      <li \*ngFor="let movie of movies">        {{ movie.title }} - {{movie.director}}      </li>    </ul> |

The Angular creates a li element for each movie. So if there are n number of movies, the angular inserts the n number of li nodes into the DOM

But the data will not remain constant. The user will add a new movie, delete a movie, sort the list in a different order, or simply refresh the movie from the back end. This will force the angular to render the template again.

The easiest way to achieve that is to remove the entire list and render the DOM again. But this is inefficient and if the list is large it is a very expensive process.

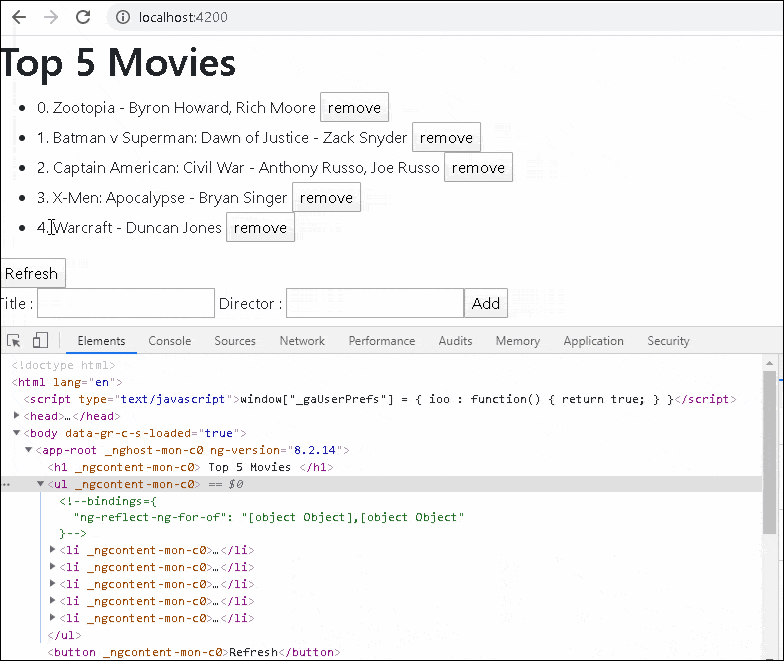
To avoid that the Angular uses the object identity to track the elements in the collection to the DOM nodes. Hence when you add an item or remove an item, the Angular will track it and update only the modified items in the DOM.

But if you refresh the entire list from the back end, it will replace the objects in the movie collection with the new objects. Even if the movies are the same, Angular will not be able to detect as the object references have changed. Hence it considers them new and renders them again after destroying the old ones.

The following example shows what happens when we refresh the entire list. The App displays the list of movies. it has option to add a movie, remove a movie and refresh the entire movie.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | import { Component, OnInit } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css'],  })  export class AppComponent implements OnInit {    title: string = "Top 5 Movies";      movies=[];      mTitle:string="";    mDirector:string="";      ngOnInit() {      this.Refresh();    }      remove(i) {      this.movies.splice(i,1);    }      addMovie() {      this.movies.push({ title: this.mTitle, director: this.mDirector})      this.mTitle=""      this.mDirector=""    }      Refresh() {      console.log("refresh")      this.movies = [        { title: 'Zootopia', director: 'Byron Howard, Rich Moore'},        { title: 'Batman v Superman: Dawn of Justice', director: 'Zack Snyder'},        { title: 'Captain American: Civil War', director: 'Anthony Russo, Joe Russo'},        { title: 'X-Men: Apocalypse', director: 'Bryan Singer'},        { title: 'Warcraft', director: 'Duncan Jones'},      ]    }  }    class Movie {    title: string;    director: string;  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | <h1> {{title}} </h1>    <ul>    <li \*ngFor="let movie of movies; let i=index;trackBy: trackByFn;">      {{i}}. {{ movie.title }} - {{movie.director}} <button (click)="remove(i)">remove</button>    </li>  </ul>    <button (click)="Refresh()">Refresh</button> <br>    Title     : <input type="text" [(ngModel)]="mTitle">  Director  : <input type="text" [(ngModel)]="mDirector">  <button  (click)="addMovie()">Add</button> |



You can see from the above example, that Angular renders the entire DOM every time we click on refresh.

Trackby

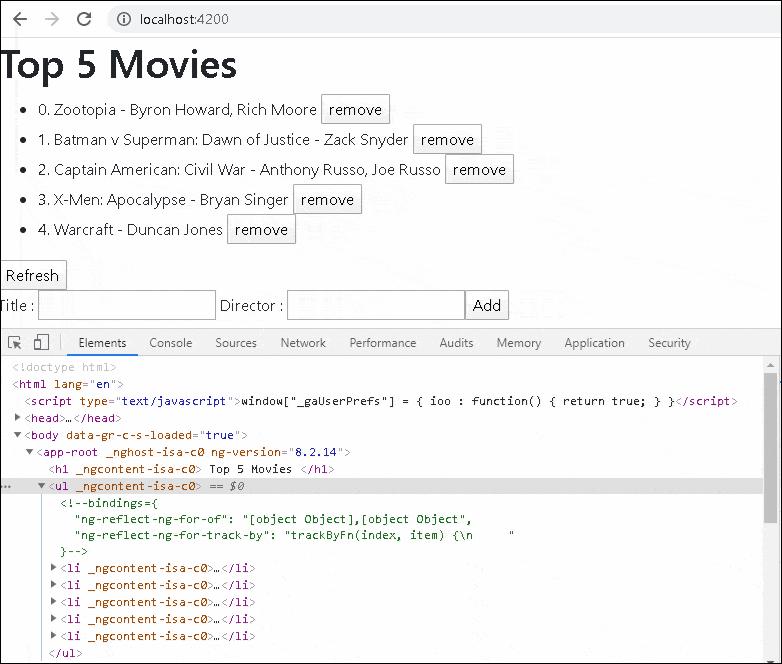
We can solve this problem by providing a function to the trackBy option that returns a unique id for each item. The ngFor will use the unique id returned by the trackBy function to track the items. Hence even if we refresh the data from the back end, the unique id will remain the same and the list will not be rendered again.

The trackBy takes a function that has two arguments: index and the current item. It must return a id that uniquely identifies the item. The following example returns the title as the unique id.

|  |  |
| --- | --- |
| 1  2  3  4  5 | trackByFn(index, item) {      return item.title;    } |

In the template assign the newly created trackByFn to trackBy option in the ngFor statement.

|  |  |
| --- | --- |
| 1  2  3 | <li \*ngFor="let movie of movies; let i=index;trackBy: trackByFn;"> |



Trackby multiple fields

You can also trackby multiple fields as shown below

|  |  |
| --- | --- |
| 1  2  3 | <li \*ngFor="let movie of movies; let i=index;trackBy: trackByFnMultipleFields;"> |

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | trackByFnMultipleFields(index, item) {      return item.title + item.director;    } |

The Attribute directives can change the appearance or behavior of an element.

Creating Custom Attribute Directive

The Angular has several built-in attribute directives. Let us create a ttClass directive, which allows us to add class to an element. Similar to the [Angular ngClass](https://www.tektutorialshub.com/angular/angular-ngclass-directive/) directive.

Create a new file and name it as tt-class.directive.ts. import the necessary libraries that we need.

|  |  |
| --- | --- |
| 1  2  3 | import { Directive, ElementRef, Input, OnInit } from '@angular/core' |

Decorate the class with @Directive. Here we need to choose a selector (ttClass) for our directive. We name our directive as ttClassDirective.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Directive({    selector: '[ttClass]',  })  export class ttClassDirective implements OnInit { |

Our directive needs to take the class name as the input. The [Input](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) decorator marks the property ttClass as the input property. It can receive the class name from the parent component.

We use the same name same as the select name ttClass. This will enable us to use the [property binding](https://www.tektutorialshub.com/angular/property-binding-in-angular/) syntax <button [ttClass]="'blue'"> in the component.

You can also create more than [@Input](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) properties.

|  |  |
| --- | --- |
| 1  2  3 | @Input() ttClass: string; |

We attach the attribute directive to an element, which we call the parent element. To change the properties of the parent element, we need to get the reference. Angular injects the parent element when we ask for the instance of the [ElementRef](https://www.tektutorialshub.com/angular/elementref-in-angular/) in its constructor.

|  |  |
| --- | --- |
| 1  2  3  4 | constructor(private el: ElementRef) {  } |

[ElementRef](https://www.tektutorialshub.com/angular/elementref-in-angular/) is a wrapper for the Parent DOM element. We can access the DOM element via the property nativeElement. The classList method allows us to add the class to the element.

|  |  |
| --- | --- |
| 1  2  3  4  5 | ngOnInit() {    this.el.nativeElement.classList.add(this.ttClass);  } |

The complete code is as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import { Directive, ElementRef, Input, OnInit } from '@angular/core'    @Directive({    selector: '[ttClass]',  })  export class ttClassDirective implements OnInit {      @Input() ttClass: string;      constructor(private el: ElementRef) {    }      ngOnInit() {      this.el.nativeElement.classList.add(this.ttClass);    }    } |

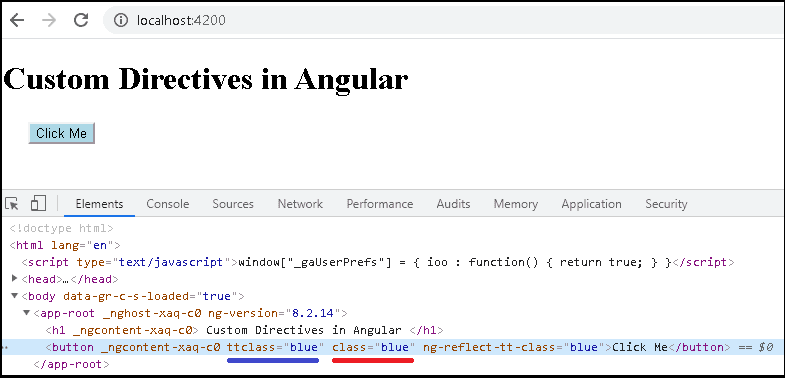
In the app.component.css and the CSS class blue

|  |  |
| --- | --- |
| 1  2  3  4  5 | .blue {    background-color: lightblue;  } |

Finally in the component template attach our customer directive ttClass to the button element.

|  |  |
| --- | --- |
| 1  2  3 | <button [ttClass]="'blue'">Click Me</button> |

You can see from the image below, that class='blue' is inserted by Our Custom Directive.

Custom Directive Example in Angular

The above is a simple imitation of [ngClass](https://www.tektutorialshub.com/angular/angular-ngclass-directive/). Have a look at the source code of [ngClass](https://github.com/angular/angular/blob/master/packages/common/src/directives/ng_class.ts)

Creating Custom Structural Directive

Now, let us build a Custom Structural directive. Let us mimic the [ngIf](https://www.tektutorialshub.com/angular/angular-ngif-directive/) and create a custom directive, which we name it as ttIf. There is hardly any difference in creating a Attribute or structural directive.

We start of with creating a tt-if.directive.ts file and import the relevant modules.

|  |  |
| --- | --- |
| 1  2  3 | import { Directive, ViewContainerRef, TemplateRef, Input } from '@angular/core'; |

Decorate the class with @Directive with the selector as (ttIf). We name our directive as ttIfDirective.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Directive({    selector: '[ttIf]'  })  export class ttIfDirective  { |

A variable to hold our if condition.

|  |  |
| --- | --- |
| 1  2  3 | \_ttif: boolean; |

Since, we are manipulating the DOM, we need ViewContainerRef and TemplateRef instances.

|  |  |
| --- | --- |
| 1  2  3  4  5 | constructor(private \_viewContainer: ViewContainerRef,              private templateRef: TemplateRef<any>) {    } |

Our directive needs to take the if condition as the input. The [Input](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) decorator marks the property ttIf as the input property. Note that we are using [setter](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/#intercept-input-property-changes-with-a-setter) function, because we want the add or remove the content whenever the if condition changes.

We use the same name same as the select name ttIf. This will enable us to use the [property binding](https://www.tektutorialshub.com/angular/property-binding-in-angular/) syntax <div \*ttIf="show"> in the template.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Input()    set ttIf(condition) {      this.\_ttif = condition      this.\_updateView();    } |

This is where all the magic happens. We use the createEmbeddedView method of the ViewContainerRef to insert the template if the condition is true. The clear removes the template from the DOM.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | \_updateView() {      if (this.\_ttif) {        this.\_viewContainer.createEmbeddedView(this.templateRef);      }      else {        this.\_viewContainer.clear();      } |

That it. Remember to ttIfDirective in the declaration array of the app.module.ts. The complete code is as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | import { Directive, ViewContainerRef, TemplateRef, Input } from '@angular/core';    @Directive({    selector: '[ttIf]'  })  export class ttIfDirective  {      \_ttif: boolean;      constructor(private \_viewContainer: ViewContainerRef,              private templateRef: TemplateRef<any>) {    }        @Input()    set ttIf(condition) {      this.\_ttif = condition      this.\_updateView();    }      \_updateView() {      if (this.\_ttif) {        this.\_viewContainer.createEmbeddedView(this.templateRef);      }      else {        this.\_viewContainer.clear();      }    }    } |

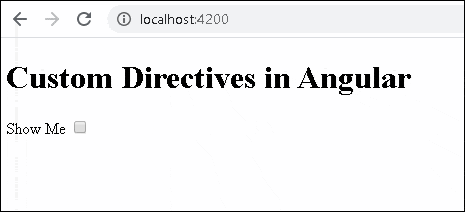
Component class

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    title: string = "Custom Directives in Angular";    show=true;  } |

Template

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | <h1> {{title}} </h1>    Show Me  <input type="checkbox" [(ngModel)]="show">    <div \*ttIf="show">    Using the ttIf directive  </div>    <div \*ngIf="show">    Using the ngIf directive  </div> |

Run the app and compare the ngIf & our custom directive ttIf side by side.



Why you need to specify \*

Remove the \* from our newly created ttIf directive. And you will get the error message

**ERROR NullInjectorError: StaticInjectorError(AppModule)[NgIf -> TemplateRef]:  
StaticInjectorError(Platform: core)[NgIf -> TemplateRef]:  
NullInjectorError: No provider for TemplateRef!**

We use the \*notation to tell Angular that we have a structural directive and we will be manipulating the DOM. It basically tells angular to inject the TemplateRef. To inject the templateRef, the Angular needs to locate the template. The \* tells the Angular to locate the template and inject its reference as templateRef

Custom Directive Examples

The following two more Custom Directive Examples. Toggle & Tooltip directives

Toggle Directive

The following directive adds or removes the CSS class toggle from the Parent element. We do that by listening to the click event on the host element or parent element.

Angular makes this easy to listen to the events from the parent or host element using the[@HostListener](https://www.tektutorialshub.com/angular/hostbinding-and-hostlistener-in-angular/) function decorator. We use it to decorate the function (onClick method in the example). It accepts the name of the event as the argument and invokes the decorated method whenever the user raises the event.

|  |  |
| --- | --- |
| 1  2  3  4 | @HostListener('click')  private onClick() { |

The complete code of the ttToggleDirective is as follows.

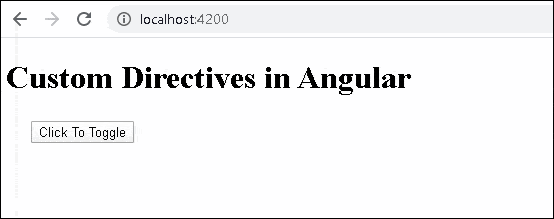
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | import { Directive, ElementRef, Renderer2, Input, HostListener, HostBinding } from '@angular/core'    @Directive({    selector: '[ttToggle]',  })  export class ttToggleDirective {      private elementSelected = false;      constructor(private el: ElementRef) {    }      ngOnInit() {    }      @HostListener('click')    private onClick() {      this.elementSelected = !this.elementSelected;      if (this.elementSelected) {        this.el.nativeElement.classList.add('toggle')      } else {        this.el.nativeElement.classList.remove('toggle')      }    }    } |

Add the following CSS Class

|  |  |
| --- | --- |
| 1  2  3  4  5 | .toggle {    background-color: yellow  } |

Use it as follows.

|  |  |
| --- | --- |
| 1  2  3 | <button ttToggle>Click To Toggle</button> |



Tooltip Directive

The tooltip directive shows the tip whenever the user hovers over it. The directive uses the [HostListener](https://www.tektutorialshub.com/angular/hostbinding-and-hostlistener-in-angular/) to listen to the mouseenter and mouseleave events.

The showHint method adds a span element into the DOM and sets its top & left position just below the host element. The removeHint removes it from the DOM.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | import { Directive, ElementRef, Renderer2, Input, HostListener } from '@angular/core'    @Directive({    selector: '[ttToolTip]',  })  export class ttTooltipDirective {      @Input() toolTip: string;      elToolTip: any;      constructor(private elementRef: ElementRef,              private renderer: Renderer2) {    }      @HostListener('mouseenter')    onMouseEnter() {      if (!this.elToolTip) { this.showHint(); }    }      @HostListener('mouseleave')    onMouseLeave() {      if (this.elToolTip) { this.removeHint(); }    }      ngOnInit() {    }      removeHint() {      this.renderer.removeClass(this.elToolTip, 'tooltip');      this.renderer.removeChild(document.body, this.elToolTip);      this.elToolTip = null;    }      showHint() {        this.elToolTip = this.renderer.createElement('span');      const text = this.renderer.createText(this.toolTip);      this.renderer.appendChild(this.elToolTip, text);        this.renderer.appendChild(document.body, this.elToolTip);      this.renderer.addClass(this.elToolTip, 'tooltip');        let hostPos = this.elementRef.nativeElement.getBoundingClientRect();      let tooltipPos= this.elToolTip.getBoundingClientRect();        let top = hostPos.bottom+10 ;      let left = hostPos.left;        this.renderer.setStyle(this.elToolTip, 'top', `${top}px`);      this.renderer.setStyle(this.elToolTip, 'left', `${left}px`);    }  } |

Add the following CSS Class

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | .tooltip {    display: inline-block;    border-bottom: 1px dotted black;    position: absolute;  } |

Use it as follows.

|  |  |
| --- | --- |
| 1  2  3 | <button ttToolTip toolTip="Tip of the day">Show Tip</button> |

# PIPES

Angular Pipes Syntax

The syntax of the pipe is as follows

|  |  |
| --- | --- |
| 1  2  3 | Expression | pipeOperator[:pipeArguments] |

Where

Expression: is the expression, which you want to transform  
| : is the Pipe Character  
pipeOperator : name of the Pipe  
pipeArguments: arguments to the Pipe

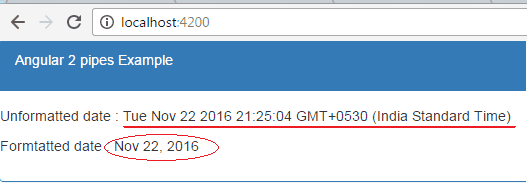
Pipes Example

In this example let use Angular built in date pipe to transform the date

Component class

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import { Component } from '@angular/core';  import { FormsModule } from '@angular/forms';    @Component({      selector: 'app-root',      templateUrl: `<p> Unformatted date : {{toDate }} </p>                    <p> Formatted date : {{toDate | date}} </p>`  })  export class AppComponent  {      title: string = 'pipe Example' ;      toDate: Date = new Date();  } |

In the above example, we are taking current date and transforming it into the easily readable format using the date pipe. We have included the unformatted date format for comparison. The output is as shown below

Example of Angular Pipes

Passing arguments to pipes

We can also pass optional arguments to the pipe. The arguments are added to the pipe using acolon (:) sign followed by the value of the argument. If there are multiple arguments separate each of them with the colon (:). For example, we can pass the format as the argument to the [date pipe](https://www.tektutorialshub.com/angular/formatting-dates-with-angular-date-pipe/), which is Optional. The medium is one of the valid value of the format argument, which displays the date in yMMMdjms format. The example code is as shown below.

|  |  |
| --- | --- |
| 1  2  3 | {{toDate | date:'medium'}} |

The parameter medium displays the date as **Nov 22, 2016, 10:04:10 PM**

Chaining Pipes

Pipes can be chained together to make use of multiple pipes in one expression. For example in the following code, the toDate is passed to the Date Pipe. The output of the Date pipe is then passed to the uppercase pipe.

|  |  |
| --- | --- |
| 1  2  3 | toDate | date | uppercase |

The Angular Built-in pipes

The Angular has several built-in pipes, which you can use in your application. You can read about them from this [link](https://angular.io/api?query=pipe)

Some of the important pipes are [Date Pipe](https://www.tektutorialshub.com/angular/formatting-dates-with-angular-date-pipe/), Uppercase Pipe, Lowercase Pipe, Number Pipe/ Decimal Pipe, Currency Pipe, and Percent Pipe**,** etc

DatePipe

The [Date pipe](https://www.tektutorialshub.com/angular/formatting-dates-with-angular-date-pipe/) formats the date according to locale rules. The syntax of the date pipe is as shown below

|  |  |
| --- | --- |
| 1  2  3 | date\_expression | date[:format] |

Where

**date\_expression** is a date object or a number

**date** is the name of the pipe

**format** is the date and time format string which indicates the format in which date/time components are displayed.

Some of the common format strings are

| Component | format | Example |
| --- | --- | --- |
| Year | y | 2016 |
| Year | yy | 16 |
| Month | M | 9 |
| Month | M | 99 |
| Month | MMM | Nov |
| Month | MMMM | November |
| Day | d | 9 |
| Day | dd | 09 |
| hour | j | 9 |
| hour | jj | 09 |
| hour | h | 9 AM |
| hour | hh | 09 AM |
| hour24 | H | 13 |
| hour24 | HH | 13 |
| minute | m | 9 |
| minute | mm | 09 |
| second | s | 9 |
| second | ss | 99 |
| Time zone | z | Pacific Standard time |
| Time zone | Z | GMT-8:00 |
| Time zone | a | PM |
| Era | G | AD |
| Era | GGGG | Anno Domini |

Format argument also supports some predefined commonly used formats

| **Format Name** | **Equivalent Format strng** | **Example (for en-US)** |
| --- | --- | --- |
| medium | yMMMdjms | Sep 3, 2010, 12:05:08 PM |
| short | yMdjm | 9/3/2010, 12:05 PM |
| fullDate | yMMMMEEEEd | Friday, September 3, 2010 |
| longDate | yMMMMd | September 3, 2010 |
| mediumDate | yMMMd | Sep 3, 2010 |
| shortDate | yMd | 9/3/2010 |
| mediumTime | jms | 12:05:08 PM |
| shortTime | jm | 12:05 PM |

You can read about the complete list from [link](https://angular.io/api/common/DatePipe)

Example of Datepipe

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import { Component } from '@angular/core';  import { FormsModule } from '@angular/forms';  @Component({      selector: 'app-root',      template:`<p>medium : {{toDate | date:'medium'}} </p>                <p>short : {{toDate | date:'short'}} </p>                <p>fullDate : {{toDate | date:'fullDate'}} </p>                <p>longDate : {{toDate | date:'longDate'}} </p>                <p>mediumDate : {{toDate | date:'mediumDate'}} </p>                <p>shortDate : {{toDate | date:'shortDate'}} </p>                <p>mediumTime : {{toDate | date:'mediumTime'}} </p>                <p>dd-MM-y : {{toDate | date:'dd-MM-y'}} </p>                <p>dd-MM-yy HH:mm : {{toDate | date:'dd-MM-yy HH:mm'}} </p>`  })  export class AppComponent  {      title: string = 'Angular pipes Example' ;      toDate: Date = new Date();  } |

UpperCasePipe & LowerCasePipe

As the name suggests, these pipes transform the string to Uppercase or lowercase

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import { Component } from '@angular/core';  import { FormsModule } from '@angular/forms';    @Component({      selector: 'app-root',      template:`<p>Unformatted :{{msg}} </p>                <p>Uppercase :{{msg | uppercase}} </p>                <p>Lowercase :{{msg | lowercase}} </p>`  })  export class AppComponent  {      title: string = 'Angular pipes Example' ;      msg: string= 'Welcome to Angular';  } |

Read more about [uppercasepipe](https://angular.io/api/common/UpperCasePipe) & [lowercasepipe](https://angular.io/api/common/LowerCasePipe)

SlicePipe

Creates a new List or String containing a subset (slice) of the string or array. This Pipe uses the JavaScript API [Array.prototype.slice()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice) and [String.prototype.slice()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/slice).

Syntax

|  |  |
| --- | --- |
| 1  2  3 | array\_or\_string\_expression | slice:start[:end] |

Where

**array\_or\_string\_expression** is the string to slice

**slice** is the name of the pipe

**start** is the start position/index from where the slicing will start

**end**is the ending index/position in the array/string

The slice pipes take two arguments. The first argument**start** is the starting index of the string/array. The second argument **end** is the ending index of the string/array. If the start or end index is negative then the index is counted from end of the string/array

Example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import { Component } from '@angular/core';  import { FormsModule } from '@angular/forms';  @Component({      selector: 'app-root',      template:`<p>Complete String :{{msg}} </p>                <p>Example 1 :{{msg | slice:11:20}} </p>                <p>Example 2 :{{msg | slice:-9}} </p>`  })    export class AppComponent  {      title: string = 'Angular pipes Example' ;      msg: string= 'Welcome to Angular ';  } |

Both the above examples will display Angular. You can read more about slice from this [link](https://angular.io/api/common/SlicePipe)

DecimalPipe / NumberPipe

The Decimal Pipe is used to Format a number as Text. This pipe will format the number according to locale rules.

Syntax

|  |  |
| --- | --- |
| 1  2  3 | number\_expression | number[:digitInfo] |

Where

**number\_expression** is the number you want to format

**number** is the name of the pipe

**digitInfo** is a string which has the following format

{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}

Where

**minIntegerDigits** is the minimum number of integer digits to use. Defaults to 1.

**minFractionDigits** is the minimum number of digits after fraction. Defaults to 0.

**maxFractionDigits** is the maximum number of digits after fraction. Defaults to 3.

Example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | import { Component } from '@angular/core';  import { FormsModule } from '@angular/forms';  @Component({      selector: 'app-root',      template: `<p> Unformatted :{{num}}</p>                 <p> Formatted :{{num | number}}</p>                 <p> Formatted :{{num | number:'3.1-2'}}</p>                 <p> Formatted :{{num | number:'7.1-5'}} </p>`  })    export class AppComponent  {      title: string = 'Angular pipes Example' ;      num: number= 9542.14554;  } |

PercentePipe

Formats the given number as a percentage according to locale rules.

|  |  |
| --- | --- |
| 1  2  3 | number\_expression | percent[:digitInfo] |

|  |  |
| --- | --- |
| 1  2  3 |  |

Where

**number\_expression** is the number you want to format

**percent** is the name of the pipe

**digitInfo** is a string which has the following format. It is similar to used in decimal pipe

Example code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import { Component } from '@angular/core';  import { FormsModule } from '@angular/forms';    @Component({      selector: 'app-root',      template:`<p>Unformatted :{{per}} </p>                <p>Example 1 :{{per | percent }} </p>                <p>Example 2 :{{per | percent:'1.2-2'}} </p>`  })  export class AppComponent  {      title: string = 'Angular pipes Example' ;      per: number= .7414;2';  } |

More about Percent pipe from the [link](https://angular.io/api/common/PercentPipe)

CurrencyPipe

Formats a number as currency using locale rules.

|  |  |
| --- | --- |
| 1  2  3 | number\_expression | currency[:currencyCode[:symbolDisplay[:digitInfo]]] |

Where

**number\_expression** currency to format a number as currency.

**Currency** is the name of the pipe

**currencyCode** is the ISO 4217 currency code, such as USD for the US dollar and EUR for the euro.

**symbolDisplay** is a boolean indicating whether to use the currency symbol or code. Use true to display symbol and false to use code

**digitInfo** is similar to the one used in decimal pipe

Example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import { Component } from '@angular/core';  import { FormsModule } from '@angular/forms';  @Component({      selector: 'app-root',      template: `<p>Unformatted :{{cur}} </p>                 <p>Example 1 :{{cur | currency }} </p>                 <p>Example 2 :{{cur | currency:'INR':true:'4.2-2'}} </p>`  })    export class AppComponent  {      title: string = 'Angular pipes Example' ;      cur: number= 175;  } |

How to Create Custom Pipes

To create a Custom Pipe, first, You need to follow these steps

Create a pipe class

Decorate the class with @pipe decorator.

Give a name to the pipe in the name meta data of the @pipe decorator. We will use this name in the template.

The pipe class must implement the PipeTransform interface. The interfaces contain only one method transform.

The first parameter to the transform method is the value to be transferred. The transform method must transform the value and return the result. You can add any number of additional arguments to the transform method.

Declare the pipe class in the Angular Module (app.module.ts)

Use the custom pipe just as you use other pipes.

Now let us create a Temperature converter pipe, which converts temperature from Celsius to Fahrenheit and vice versa.

Temparature Convertor Custom Pipe Example

Create a new Angular application. If you are new to Angular you can refer to the tutorial [Create Angular Application](https://www.tektutorialshub.com/angular/angular-create-first-application/).

We are using bootstrap 4 for styling. Hence open the index.html and add the following

|  |  |
| --- | --- |
| 1  2  3 | <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm" crossorigin="anonymous"> |

Create a new file temp-convertor.pipe.ts. Under the folder src/app. Copy the following code and paste it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import {Pipe, PipeTransform} from '@angular/core';    @pipe({      name: 'tempConverter'  })  export class TempConverterPipe implements PipeTransform {      transform(value: number, unit: string) {          if(value && !isNaN(value)) {              if (unit === 'C') {                  var temperature = (value - 32) /1.8 ;                  return temperature.toFixed(2);              } else if (unit === 'F'){                  var temperature = (value \* 1.8 ) + 32                  return temperature.toFixed(2);              }          }          return;      }  } |

Let us look at the code in details

We need to import the Pipe & PipeTransform libraries from Angular. These libraries are part of the Angular Core

|  |  |
| --- | --- |
| 1  2  3 | import {Pipe, PipeTransform} from '@angular/core'; |

We decorate TempConverterPipe class with @pipe decorator. The[@pipe decorator](https://angular.io/api/core/Pipe) is what tells Angular that the class is a pipe. the decorator expects us to provide a name to the pipe. We have given it as tempConverter. This is the name we must use in the template to make use of this pipe.

Our class must implement the PipeTransform interface.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @pipe({      name: 'tempConverter'  })  export class TempConverterPipe implements PipeTransform {      } |

The PipeTransform interface defines only one method transform. The interface definition is as follows.

|  |  |
| --- | --- |
| 1  2  3  4  5 | interface PipeTransform {    transform(value: any, ...args: any[]): any  } |

The first argument value is the value, that pipe needs to transform. We can also include any number of arguments. The method must return the final transformed data.

The following is Our implementation of the transform method. The first is Value and the second is the Unit. The unit expects either C (Convert to Celsius) or F ( convert to Fahrenheit). It converts the value received to either to Celsius or to Fahrenheit based on the Unit.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | export class TempConverterPipe implements PipeTransform {        transform(value: number, unit: string) {          if(value && !isNaN(value)) {              if (unit === 'C') {                 var temperature = (value - 32) /1.8 ;                 return temperature.toFixed(2);              } else if (unit === 'F'){                 var temperature = (value \* 1.8 ) + 32                 return temperature.toFixed(2);              }          }          return;      }    } |

Declare the Pipe

Before using our pipe, we need to tell our component, where to find it. This is done by first by importing it and then including it in declarations array of the AppModule.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule } from '@angular/forms';  import { HttpModule } from '@angular/http';    import { AppComponent } from './app.component';    import {TempConverterPipe} from './temp-convertor.pipe';    @NgModule({      declarations: [AppComponent,TempConverterPipe],      imports: [BrowserModule,FormsModule,HttpModule],      bootstrap: [AppComponent]  })  export class AppModule { } |

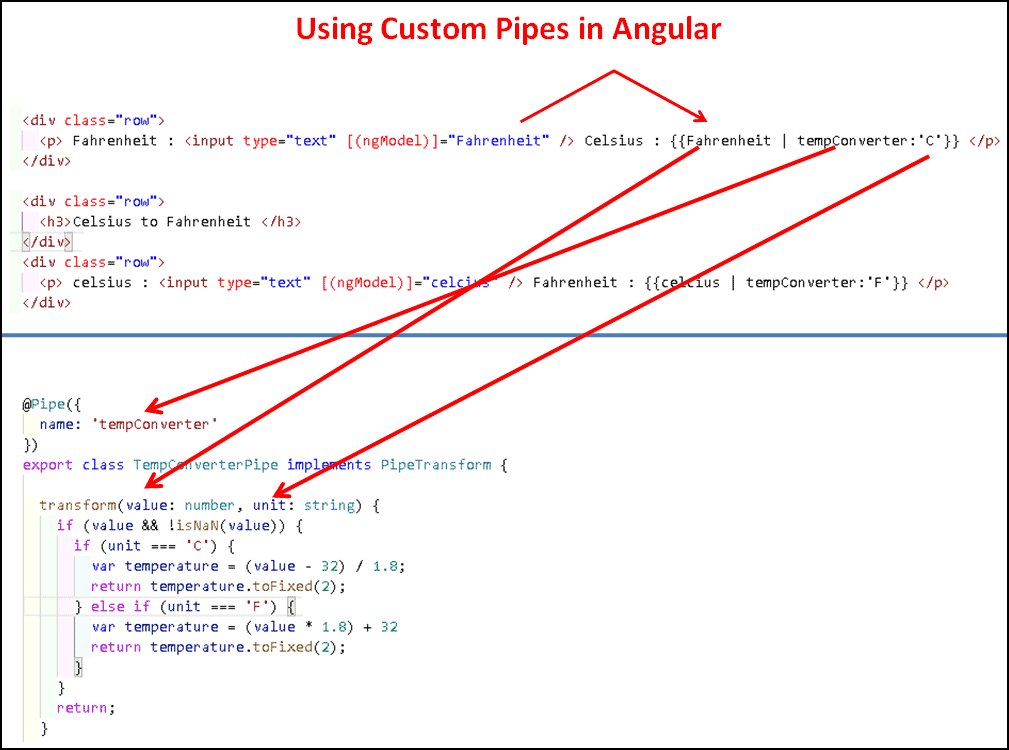
Using the Custom Pipe

The custom pipes are used in the same as the Angular built-in pipes are used. Add the following HTML code to your app.component.html file

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | <div class='card'>    <div class='card-header'>      <p>{{title}} </p>    </div>    <div class="card-body">        <div class="row">        <h3>Fahrenheit to Celsius </h3>      </div>      <div class="row">        <p> Fahrenheit : <input type="text" [(ngModel)]="Fahrenheit" />        Celsius : {{Fahrenheit | tempConverter:'C'}} </p>      </div>        <div class="row">        <h3>Celsius to Fahrenheit </h3>      </div>      <div class="row">        <p> celsius : <input type="text" [(ngModel)]="celcius" />         Fahrenheit : {{celcius | tempConverter:'F'}} </p>      </div>    </div>  </div> |

We use our pipe as follows. Fahrenheit is sent to the tempConverter as the first argument value. We use the | to indicate that the tempConverter is a pipe to angular. The C after the colon is the first argument. You can pass more than argument to the pipe by separating each argument by a : colon.

|  |  |
| --- | --- |
| 1  2  3 | {{Fahrenheit | tempConverter:'C'}} |

Using the TempConverter Pipe in Template

app.component code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import { Component } from '@angular/core';  import { FormsModule } from '@angular/forms';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })    export class AppComponent  {      title: string = 'Angular Custom Pipe Example' ;      celcius: number;      Fahrenheit: number;  } |

Using Date Pipe

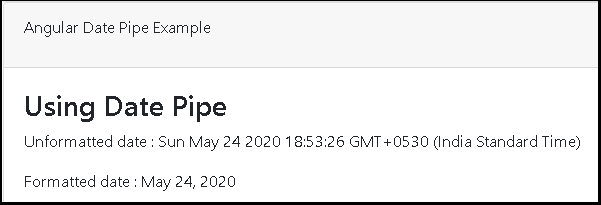
The Date uses the pipe operator i.e |. Specify the date\_expression, which you want to format in the left side of the |. On the right side specify date followed by the arguments. It accepts three arguments format, timezone & locale

|  |  |
| --- | --- |
| 1  2  3 | {{ date\_expression | date [ : format [ : timezone [ : locale ] ] ] }} |

Date Pipe Example

The following is the example of a date pipe it its simplest form.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <h3>Using Date Pipe </h3>  <p>Unformatted date : {{toDate }} </p>     //Without pipe  <p>Formatted date : {{toDate | date}} </p>   //With Date Pipe |

Date Pipe Example

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Ad by Valueimpression

Date Expression

The Date Expression can be anything that evaluates to date. For example it can be a Date object, a number (milliseconds since UTC epoch), or an [ISO string](https://www.w3.org/TR/NOTE-datetime)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | //Component  toDate: Date = new Date();  numDate=1590319189931;  strDate="Sun May 24 2020 19:16:23"; |

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <h3>Date Expression </h3>  <p>Date Object : {{toDate | date}} </p>       //May 24, 2020  <p>Number Date : {{numDate | date}} </p>      //May 24, 2020  <p>ISO Date : {{strDate | date}} </p>         //May 24, 2020 |

Parameters to Date Pipe

The Date pipe accepts three arguments format, timezone & locale

| **Parameter** | **Data Type** | **Particulars** |
| --- | --- | --- |
| format | string | There are two types of format. 1. predefined formats 2. Custom format string  Default: mediumDate. |
| timezone | string | Use  A timezone offset (such as '+0430')  Standard UTC/GMT  continental timezone abbreviation.  Default: End-users local system timezone |
| locale | string | A locale code for the locale format rules to use. Default: The value of LOCALE\_ID ( which is en-US) [How to Change LOCALE\_ID](https://angular.io/api/core/LOCALE_ID) |

Date Format

There are two types formats options available

Pre defined Format

Custom Format string

Pre defined formats

The following are the Pre defined formats that you can use. We have also mentioned the corresponding custom formats next to it.

| **Format** | **Equivalent Custom Format** | **Example** | **Result** |
| --- | --- | --- | --- |
| short | 'M/d/yy, h:mm a' | {{toDate | date:'short'}} | 5/24/20, 3:40 PM |
| medium | 'MMM d, y, h:mm:ss a' | {{toDate | date:'medium'}} | May 24, 2020, 3:42:17 PM |
| long | 'MMMM d, y, h:mm:ss a z' | {{toDate | date:'long'}} | May 24, 2020 at 3:42:17 PM GMT+5 |
| full | 'EEEE, MMMM d, y, h:mm:ss a zzzz' | {{toDate | date:'full'}} | Sunday, May 24, 2020 at 3:42:17 PM GMT+05:30 |
| shortDate | 'M/d/yy' | {{toDate | date:'shortDate'}} | 5/24/20 |
| mediumDate | 'MMM d, y' | {{toDate | date:'mediumDate'}} | May 24, 2020 |
| longDate | 'MMMM d, y' | {{toDate | date:'longDate'}} | May 24, 2020 |
| fullDate | 'EEEE, MMMM d, y' | {{toDate | date:'fullDate'}} | Sunday, May 24, 2020 |
| shortTime | 'h:mm a' | {{toDate | date:'shortTime'}} | 3:42 PM |
| mediumTime | 'h:mm:ss a' | {{toDate | date:'mediumTime'}} | 3:42:17 PM |
| longTime | 'h:mm:ss a z' | {{toDate | date:'longTime'}} | 3:42:17 PM GMT+5 |
| fullTime | 'h:mm:ss a zzzz' | {{toDate | date:'fullTime'}} | 3:42:17 PM GMT+05:30 |

Custom Format string

The following is the complete list of custom formats that are available

| **Field type** | **Format** | **Description** | **Example Value** |
| --- | --- | --- | --- |
| Era | G, GG & GGG | Abbreviated | AD |
| GGGG | Wide | Anno Domini |
| GGGGG | Narrow | A |
| Year | y | Numeric: minimum digits | 2, 20, 201, 2017, 20173 |
| yy | umeric: 2 digits + zero padded | 02, 20, 01, 17, 73 |
| yyy | Numeric: 3 digits + zero padded | 002, 020, 201, 2017, 20173 |
| yyyy | Numeric: 4 digits or more + zero padded | 0002, 0020, 0201, 2017, 20173 |
| Month | M | Numeric: 1 digit | 9, 12 |
| MM | Numeric: 2 digits + zero padded | 09, 12 |
| MMM | Abbreviated | Sep |
| MMMM | Wide | September |
| MMMMM | Narrow | S |
| Month standalone | L | Numeric: 1 digit | 9, 12 |
| LL | Numeric: 2 digits + zero padded | 09, 12 |
| LLL | Abbreviated | Sep |
| LLLL | Wide | September |
| LLLLL | Narrow | S |
| Week of year | w | Numeric: minimum digits | 1... 53 |
| ww | Numeric: 2 digits + zero padded | 01... 53 |
| Week of month | W | Numeric: 1 digit | 1... 5 |
| Day of month | d | Numeric: minimum digits | 1 |
| dd | Numeric: 2 digits + zero padded | 01 |
| Week day | E, EE & EEE | Abbreviated | Tue |
| EEEE | Wide | Tuesday |
| EEEEE | Narrow | T |
| EEEEEE | Short | Tu |
| Period | a, aa & aaa | Abbreviated | am/pm or AM/PM |
| aaaa | Wide (fallback to a when missing) | ante meridiem/post meridiem |
| aaaaa | Narrow | a/p |
| Period\* | B, BB & BBB | Abbreviated | mid |
| BBBB | Wide | am, pm, midnight, noon, morning, afternoon, evening, night |
| BBBBB | Narrow | md. |
| Period standalone\* | b, bb & bbb | Abbreviated | mid. |
| bbbb | Wide | am, pm, midnight, noon, morning, afternoon, evening, night |
| bbbbb | Narrow | md |
| Hour 1-12 | h | Numeric: minimum digits | 1, 12 |
| hh | Numeric: 2 digits + zero padded | 01, 12 |
| Hour 0-23 | H | Numeric: minimum digits | 0, 23 |
| HH | Numeric: 2 digits + zero padded | 00, 23 |
| Minute | m | Numeric: minimum digits | 8, 59 |
| mm | Numeric: 2 digits + zero padded | 08, 59 |
| Second | s | Numeric: minimum digits | 0... 59 |
| ss | Numeric: 2 digits + zero padded | 00... 59 |
| Fractional seconds | S | Numeric: 1 digit | 0... 9 |
| SS | Numeric: 2 digits + zero padded | 00... 99 |
| SSS | Numeric: 3 digits + zero padded (= milliseconds) | 000... 999 |
| Zone | z, zz & zzz | Short specific non location format (fallback to O) | GMT-8 |
| zzzz | Long specific non location format (fallback to OOOO) | GMT-08:00 |
| Z, ZZ & ZZZ | ISO8601 basic format | -0800 |
| ZZZZ | Long localized GMT format | GMT-8:00 |
| ZZZZZ | ISO8601 extended format + Z indicator for offset 0 (= XXXXX) | -08:00 |
| O, OO & OOO | Short localized GMT format | GMT-8 |
| OOOO | Long localized GMT format | GMT-08:00 |

Custom Format example

|  |  |
| --- | --- |
| 1  2  3  4  5 | {{toDate | date:'dd/MM/y'}}                //24/05/2020    {{toDate | date:'dd/MM/yy HH:mm'}}         //May 24, 2020, 7:17:26 PM |

Timezone Example

The following examples, shows how to use time zones

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Date in India (IST Time Zone)  : {{toDate | date:'short':'IST'}}     //Date in India (IST Time Zone) : 5/24/20, 7:32 PM    Date  in USA (CDT Time Zone)   : {{toDate | date:'short':'CDT'}}    //Date in USA (CDT Time Zone) : 5/24/20, 9:02 AM    Date in India (+0530)     : {{toDate | date:'short':'+0530'}}     //Date in India (+0530) : 5/24/20, 7:32 PM    Date in USA (-0700)     : {{toDate | date:'short':'-0500'}}    //Date in USA (-0700) : 5/24/20, 9:02 AM |

Country Locale Example

The Country Locale is the third argument.

|  |  |
| --- | --- |
| 1  2  3 | British date time is {{toDate | date:'dd/MM/yy HH:mm':'GMT':'en-GB'}}       //British date time is 24/05/20 14:26 |

Syntax of Async Pipe

The following is the syntax of the async pipe. expression must return an observable or promise. It is followed by | (pipe character) and the keyword async. We are using the async pipe with [interpolation](https://www.tektutorialshub.com/angular/interpolation-in-angular/) syntax.

|  |  |
| --- | --- |
| 1  2  3 | {{expression | async}} |

Async Pipe Example with Observables

The following example [creates an observable](https://www.tektutorialshub.com/angular/rxjs-observable-using-create-of-from-in-angular/). It returns 1000 after an delay. The obsValue variable stores the observable.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | obsValue = new Observable((observer) => {      console.log("Observable starts")      setTimeout(() => { observer.next("90000") }, 1000);    }) |

We can use it in the template as shown below.

|  |  |
| --- | --- |
| 1  2  3 | {{ obsValue | async}} |

When the components load, the angular automatically subscribes to the obsValue observable.

The observable returns the value 1000 after a delay. When the value arrives, async pipe automatically triggers change detection. Hence you will see the return value on the screen.

The observable is automatically unsubscribed when the component is destroyed. Thus avoiding any potential memory leaks

Use the async pipe with ngIf

We above example uses the async pipe with [interpolation](https://www.tektutorialshub.com/angular/interpolation-in-angular/). We can also use it with the [ngIf](https://www.tektutorialshub.com/angular/angular-ngif-directive/) or [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/) etc.

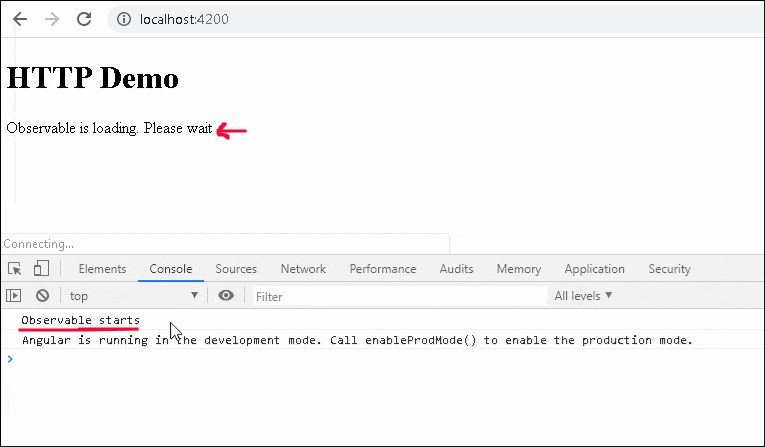
The following example shows how **NOT** to use the observable with [ngIf](https://www.tektutorialshub.com/angular/angular-ngif-directive/) directive.

The condition (obsValue | async) becomes true, when the observable returns a value. Until then the [elseBlock](https://www.tektutorialshub.com/angular/angular-ngif-directive/) is shown, which we use to display the loading indicator. In the example, it displays the message Observable is loading. Please wait.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <div \*ngIf="(obsValue | async); else elseBlock">    {{ obsValue | async}}  </div>    <ng-template #elseBlock>    Observable is loading. Please wait  </ng-template> |

When the observable returns with a value the [ngIf](https://www.tektutorialshub.com/angular/angular-ngif-directive/) condition becomes true and the pipe displays the returned value.

You can see it from the following image.



As you can see from the above image, you can see that the observable fires twice.

i.e because we are using the async pipe twice. one in if condition and the other inside the if block

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div \*ngIf="(obsValue | async); else elseBlock">         //obsValue  Subscribed here    {{ obsValue | async}}                          //obsValue  Subscribed here again  </div> |

There are two ways in whcih you can solve this problem. One is using the ShareReplay rxjs operator

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

ShareReplay

We use the shareReplay when you want subscribers to share the observable and access previously emitted values. i.e. the observable is subscribed only once and for every subsequent subscription, the previously received value is used.

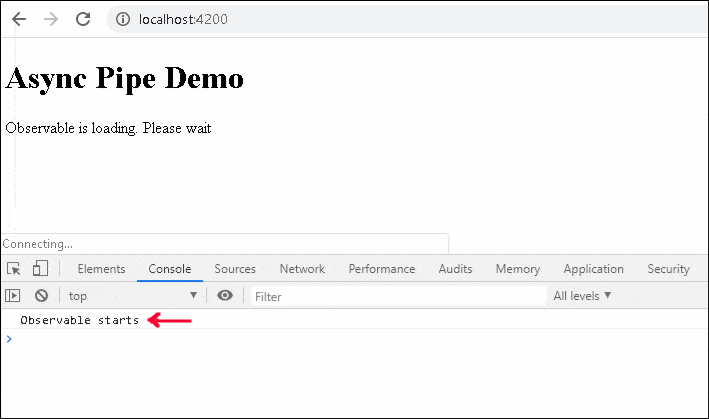
The updated observable, with shareReplay is as shown below. We need to use the pipe operator

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | obsValue = new Observable((observer) => {      console.log("Observable starts")      setTimeout(() => {        console.log("Returns value")        observer.next("1000")      }, 5000);    }).pipe(shareReplay()); |

There is no need to make any changes in component code. But for this example, we have one more [if](https://www.tektutorialshub.com/angular/angular-ngif-directive/) block. making the total async pipe to three

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <div \*ngIf="(obsValue | async); else elseBlock">    {{ obsValue | async}}  </div>    <ng-template #elseBlock>    Observable is loading. Please wait  </ng-template>    <div \*ngIf="(obsValue | async);">    observable has recevied data  </div> |

As you can see from the following, in spite of having three subscriptions, the observable is subscribed only once.



Using ngIf “as” syntax

We can use the as keyword to store the result in a [template local variable](https://www.tektutorialshub.com/angular/template-reference-variable-in-angular/). Once we assign the result to a variable, then we can use it anywhere inside the ngIf block as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <div \*ngIf="(obsValue | async) as value; else elseBlock">    {{ value}}      //works only inside the If Block  </div>    <ng-template #elseBlock>    Observable is loading. Please wait  </ng-template>    {{ value}}   // will not work |

Remove the shareReplay from the observable and check it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | obsValue = new Observable((observer) => {      console.log("Observable starts")      setTimeout(() => {        console.log("Returns value")        observer.next("1000")      }, 5000);    }); |

Use the async pipe with ngfor

Now we will see how to use the async pipe with [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/). For this example, we will make use of [httpclient](https://www.tektutorialshub.com/angular/angular-httpclient/) library to make [HTTP get](https://www.tektutorialshub.com/angular/angular-http-get-example-using-httpclient/)request and display the results using the [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/)

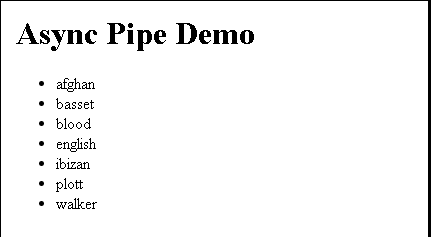
For this example, let use the free HTTP end point https://dog.ceo/dog-api/documentation/. It returns the array of hound breeds as shown below (in the message array)

|  |  |
| --- | --- |
| 1  2  3 | {"message":["afghan","basset","blood","english","ibizan","plott","walker"],"status":"success"} |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | hounds: Observable<any> = this.getHoundList();    getHoundList(): Observable<any> {    return this.http.get<any>("https://dog.ceo/api/breed/hound/list")  } |

In the template use the (hounds | async) to subscribe to the hounds observable. We are using a safe navigation operator ? before the property name message. i.e because initially, it is null until the result arrives and without ? you will see errors in your console

|  |  |
| --- | --- |
| 1  2  3  4  5 | <ul>    <li \*ngFor="let breed of (hounds | async)?.message">{{breed}}</li>  </ul> |

async pipe example using ngFor

You can also make use of combination of [ngIf](https://www.tektutorialshub.com/angular/angular-ngif-directive/)& [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/) and using the as to store the result in breeds.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <div \*ngIf="(hounds | async) as breeds">    <ul>      <li \*ngFor="let breed of breeds.message">{{breed}}</li>    </ul>  </div> |

The following code displays the a random image of the dog using ngIf

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | //component    randomPic: Observable<any> = this.getRandom();    getRandom(): Observable<any> {    return this.http.get<any>("https://dog.ceo/api/breeds/image/random")  } |

|  |  |
| --- | --- |
| 1  2  3  4  5 | //Template    <img src="{{ (randomPic | async)?.message}}"> |

How it works

Consider, that you have the following object and a map object. It has property a,b & c. We cannot use ngFor to iterate over it as it requires an array. This is where the KeyValue pipe comes into play. It will convert them to an array of key-value pair

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | obj = {    c: 123,    b: 456,    a: 789,  };      mapObj = new Map([     ['c', 123],     ['b', 446],     ['a', 789],  ]); |

We use keyvalue just like any other [pipes in Angular](https://www.tektutorialshub.com/angular/angular-pipes/) and as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5 | obj | keyvalue    mapObj | keyvalue |

The keyValue converts them and returns in the following format. each property of the object a: 789 is converted to an object with name as key and value as value { key:a, value:789 }. It creates array of such objects and returns it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | obj = [      { key:a, value:789 },      { key:b, value:446 },      { key:c, value:123 },    ]; |

Now we can use the ngFor to loop through it and display the content.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <ul>    <li \*ngFor="let item of obj | keyvalue">      {{item.key}} ---> {{item.value}}</li>  </ul>    //output    a ---> 789  b ---> 456  c ---> 123 |

Default Sorting

KeyValue pipe uses the key to sort the results array. You can see it from the above example. Even though our object was c,b & a it was sorts it as a,b,c. The keyValue pipe uses the defaultComparator to sort the result. It uses

Ascending Order if the keys are number

Alphabetical Order if keys are strings

if keys are are of different types. then covert them to to their string values and use Alphabetical Order

If key is a either Null or undefined, put then at the end of the sort.

Custom Sorting

You can customize it by providing a custom sort function (compareFn) as the first argument to the keyValue pipe

The syntax for the compareFn as shown below. It accepts first & second keyValue and must return a number. The number must be a zero if values are equivalent else either a negative number or positive number

|  |  |
| --- | --- |
| 1  2  3 | compareFn (a: KeyValue, b: KeyValue) => number |

The following are three compareFn

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | orderOriginal = (a: KeyValue<number,string>, b: KeyValue<number,string>): number => {    return 0  }    orderbyValueAsc = (a: KeyValue<number,string>, b: KeyValue<number,string>): number => {     return a.value > b.value ? -1 : (a.value > b.value) ? 0 : 1  }    orderbyValueDsc = (a: KeyValue<number,string>, b: KeyValue<number,string>): number => {    return a.value > b.value ? 1 : (a.value > b.value) ? 0 : -1  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | <ul>    <li \*ngFor="let item of obj | keyvalue">      {{item.key}} ---> {{item.value}}</li>  </ul>    //Output  a ---> 789  b ---> 456  c ---> 123      <ul>    <li \*ngFor="let item of obj | keyvalue : orderOriginal">      {{item.key}} ---> {{item.value}}</li>  </ul>    //Output  b ---> 456  c ---> 123  a ---> 78      <ul>    <li \*ngFor="let item of obj | keyvalue : orderbyValueAsc ">      {{item.key}} ---> {{item.value}}</li>  </ul>    //Output  a ---> 789  b ---> 456  c ---> 123      <ul>    <li \*ngFor="let item of obj | keyvalue : orderbyValueDsc ">      {{item.key}} ---> {{item.value}}</li>  </ul>    //Output  c ---> 123  b ---> 456  a ---> 789 |

KeyValue Pipe Example

Consider the following breeds of dogs. The example sorts the list based on the number of sub breeds. The final code is as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | breeds=      {        "corgi": ["cardigan"],        "deerhound": ["scottish"],        "bulldog": ["boston", "english", "french"],        "mastiff": ["bull", "english", "tibetan"],        "australian": ["shepherd"],        "greyhound": ["italian"],        "buhund": ["norwegian"],        "hound": ["afghan", "basset", "blood", "english", "ibizan", "plott", "walker"],        "bullterrier": ["staffordshire"],      } |

CompareFn

|  |  |
| --- | --- |
| 1  2  3  4  5 | orderClause = (a: KeyValue<number,[string]>, b: KeyValue<number,[string]>): number => {      return a.value.length > b.value.length ? -1 : (a.value.length > b.value.length) ? 0 : 1    } |

Template

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <ul>    <li \*ngFor="let item of breeds | keyvalue : orderClause ">      {{item.key}} ---> {{item.value}}</li>  </ul> |

The output

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | hound ---> afghan,basset,blood,english,ibizan,plott,walker  bulldog ---> boston,english,french  mastiff ---> bull,english,tibetan  corgi ---> cardigan  deerhound ---> scottish  australian ---> shepherd  greyhound ---> italian  buhund ---> norwegian  bullterrier ---> staffordshire |

Using Pipes in Components & Services

Using pipes in your code involves three simple steps

Import the pipe in Module

Inject pipe in the constructor

Use the transform method of the pipe

Using Date pipe in Components & Services

First import the [DatePipe](https://www.tektutorialshub.com/angular/formatting-dates-with-angular-date-pipe/) from @angular/common. Add it in the [Angular Provider](https://www.tektutorialshub.com/angular/angular-providers/) metadata providers: [ DatePipe ],.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';    import { DatePipe } from '@angular/common';    import { AppRoutingModule } from './app-routing.module';  import { AppComponent } from './app.component';    @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule,      AppRoutingModule    ],    providers: [      DatePipe    ],    bootstrap: [AppComponent]  })  export class AppModule { } |

Open the app.component.html and [inject](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) the [DatePipe](https://www.tektutorialshub.com/angular/formatting-dates-with-angular-date-pipe/) in the constructor.

|  |  |
| --- | --- |
| 1  2  3  4 | constructor(private datePipe:DatePipe) {    } |

You can use it in component as shown below.

|  |  |
| --- | --- |
| 1  2  3 | this.toDate = this.datePipe.transform(new Date()); |

The transform method accepts the date as the first argument. You can supply additional [Parameters to DatePipe](https://www.tektutorialshub.com/angular/formatting-dates-with-angular-date-pipe/#parameters-to-date-pipe) like format, timezone & locale.

|  |  |
| --- | --- |
| 1  2  3 | this.toDate = this.datePipe.transform(new Date(),'dd/MM/yy HH:mm'); |

The complete component code is as below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import { Component, OnInit } from '@angular/core';  import { DatePipe } from '@angular/common';    @Component({    selector: 'app-root',    template: `      {{toDate}}    `,    styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {    title = 'pipesInService';      toDate      constructor(private datePipe:DatePipe) {    }      ngOnInit() {        this.toDate = this.datePipe.transform(new Date());    }  } |

# Component Communication

There are few ways in which components can communicate or share data between them. And methods depend on whether the components have a Parent-child relationship between them are not.

Here are the three Possible scenarios

Parent to Child Communication

Child to Parent Communication

Interaction when there is no parent-child relation

Parent to Child Communication

If the Components have a parent-child relationship then, then the parent component can pass the data to the child using the [@input](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/#input) Property.

Using [@Input Decorator](https://www.tektutorialshub.com/angular/angular-passing-data-child-component/#the-child-component-with-input-decorator) to Pass Data

Create a property (someProperty) in the Child Component and decorate it with @Input(). This will mark the property as input property

|  |  |
| --- | --- |
| 1  2  3  4  5 | export class ChildComponent {      @Input() someProperty: number;  } |

And in the Parent Component Instantiate the Child Component. Pass the value to the someProperty using the [Property Bind](https://www.tektutorialshub.com/angular/property-binding-in-angular/) Syntax

|  |  |
| --- | --- |
| 1  2  3 | <child-component [someProperty]=value></child-component>` |

In this way, Child Component will receive the data from the parent

You can refer to the tutorial [pass data from parent to child in Angular](https://www.tektutorialshub.com/angular/angular-passing-data-child-component/).

Listen for Input Changes

The Child Component can get the values from the someProperty. But it also important for the child component to get notification when the values changes.

There are two ways in which we can achieve that.

Using OnChanges life Cycle hook or

Using a Property Setter on Input Property

Refer to the following tutorial

[Pass data from parent to child in Angular](https://www.tektutorialshub.com/angular/angular-passing-data-child-component/).

[OnChanges Life Cycle hook](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/)

[Angular @input, @output & EventEmitter](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/)

Child to Parent Communication

The Child to Parent communication can happen in three ways.

Listens to Events from Child

Uses [Local Variable](https://www.tektutorialshub.com/angular/template-reference-variable-in-angular/) to access the child in the Template

Uses a [@ViewChild](https://www.tektutorialshub.com/angular/understanding-viewchild-viewchildren-querylist-in-angular/) to get a reference to the child component

Listens to Child Event

This is done by the child component by exposing an [EventEmitter](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) Property. We also decorate this Property with [@Output](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) decorator. When Child Component needs to communicate with the parent it raises the emit event of the [EventEmitter](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) Property. The Parent Component listens to that event and reacts to it.

For Example, refer to the tutorial [Parent Listens to Child Event](https://www.tektutorialshub.com/angular/angular-pass-data-to-parent-component/#parent-listens-for-child-event)

Uses [Local Variable](https://www.tektutorialshub.com/angular/template-reference-variable-in-angular/) to access the child

Using [Local Variable](https://www.tektutorialshub.com/angular/template-reference-variable-in-angular/) is to refer to the child component is another technique.

For Example, Create a reference variable #child to the Child Component.

|  |  |
| --- | --- |
| 1  2  3 | <child-component #child></child-component> |

You can use the child (note without #) to access a property of the Child Component. The Code below displays count of the Child Component and displays it on screen

|  |  |
| --- | --- |
| 1  2  3 | <p> current count is {{child.count}} </p> |

For Example, refer to the tutorial [local variable to access the Child in Template](https://www.tektutorialshub.com/angular/angular-pass-data-to-parent-component/#parent-listens-for-child-event)

Uses a [@ViewChild](https://www.tektutorialshub.com/angular/understanding-viewchild-viewchildren-querylist-in-angular/) to get the reference to the child component

|  |  |
| --- | --- |
| 1  2  3 | <child-component></child-component> |

Another way to get the reference of the child component is using the [ViewChild](https://www.tektutorialshub.com/angular/understanding-viewchild-viewchildren-querylist-in-angular/) query in the component class

|  |  |
| --- | --- |
| 1  2  3 | @ViewChild(ChildComponent) child: ChildComponent; |

You can call any method in the Child component.

|  |  |
| --- | --- |
| 1  2  3  4  5 | increment() {      this.child.increment();    } |

For Complete explanation please refer to [Pass data from child to Parent Component](https://www.tektutorialshub.com/angular/angular-pass-data-to-parent-component/) & [ViewChild, ViewChildren & QueryList](https://www.tektutorialshub.com/angular/understanding-viewchild-viewchildren-querylist-in-angular/)

Communication when there is no relation

If the Components do not share the Parent-child relationship, then the only way they can share data is by using the services and observable.

The advantageous of using service is that

You can share data between multiple components.

Using observable, you can notify each component, when the data changes.

Create a Service and create an [Angular Observable](https://www.tektutorialshub.com/angular/angular-subject-example/) in that service using either [BehaviorSubject](https://www.tektutorialshub.com/angular/replaysubject-behaviorsubject-asyncsubject-in-angular/) or [Subject](https://www.tektutorialshub.com/angular/subjects-in-angular/).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | export class TodoService {      private \_todo = new BehaviorSubject<Todo[]>([]);    readonly todos$ = this.\_todo.asObservable();    ...  } |

The \_todo observable will emit data, whenever it is available or changes using the next method of the Subject.

|  |  |
| --- | --- |
| 1  2  3 | this.\_todo.next(Object.assign([], this.todos)); |

In the component class, you can listen to the changes just by subscribing to the observable

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | this.todoService.todos$.subscribe(val=> {        this.data=val;        //do whatever you want to with it.      }) |

For the complete explanation of the code, you can refer to

Passing data to a child/nested component

In the previous tutorials, we [built an Angular Application](https://www.tektutorialshub.com/angular/angular-create-first-application/) and then [added a child component](https://www.tektutorialshub.com/angular/angular-adding-child-component/) to it. We also looked at how Angular Component communicates with its View (templates) using the [data binding](https://www.tektutorialshub.com/angular/angular-data-binding/).

These Components are useless if they are not able to communicate with each other. They need to communicate with each other if they want to serve any useful purpose.

How to Pass data to a child component

In Angular, the Parent Component can communicate with the child component by setting its Property. To do that the Child component must expose its properties to the parent component. The Child Component does this by using the [**@Input**](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/)**decorator**

In the Child Component

Import the [@Input](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) module from @angular/Core Library

Mark those property, which you need data from the parent as input property using [@Input](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) decorator

In the Parent Component

Bind the Child component property in the Parent Component when instantiating the Child

Ad by Valueimpression

@Input Decorator

The @Input Decorator is used to configure the input properties of the component. This decorator as also supports change tracking.

When you mark a property as input property, then the Angular injects values into the component property using [Property Binding](https://www.tektutorialshub.com/angular/angular-data-binding/#property-binding). The Property Binding uses the [] brackets. The Binding Target (Property of the child component) is placed inside the square brackets. The Binding source is enclosed in quotes. [Property binding](https://www.tektutorialshub.com/angular/angular-data-binding/#property-binding) is one way from Component to the Target in the template

@Input example

Now let us build a simple component to demonstrate the use of @Input.

Our application will have a counter which is incremented by the Parent Component. The Parent Component then passes this counter to the child component for display in its template

You can download the source of this tutorial from the [Github](https://github.com/tekTutorialsHub/Angular-Advanced-Components). The initial code is available in GettingStarted Folder and Final Code in InputDecorator folder

The Child Component with @Input Decorator

Create the ChildComponent.ts under the src/app folder. And copy the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | import { Component, Input  } from '@angular/core';    @Component({      selector: 'child-component',      template: `<h2>Child Component</h2>                 current count is {{ count }}      `  })  export class ChildComponent {      @Input() count: number;  } |

Now, let us look at the code in detail

First, we import the @Input decorator from @angular/core

|  |  |
| --- | --- |
| 1  2  3 | import { Component, Input  } from '@angular/core'; |

We have defined the inline template for the child component, where it displays the current count.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Component({      selector: 'child-component',      template: `<h2>Child Component</h2>                 current count is {{ count }}     `  }) |

The Child Component expects the count to come from the Parent Component. Hence in ChildComponent decorate the count property with @Input decorator

|  |  |
| --- | --- |
| 1  2  3  4  5 | export class ChildComponent {      @Input() count: number;  } |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Bind to Child Property in Parent Component

Now, time to pass the Count values to the Child Component from the Parent

Open the app.component.ts and copy the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | import { Component} from '@angular/core';    @Component({    selector: 'app-root',    template: `          <h1>Welcome to {{title}}!</h1>          <button (click)="increment()">Increment</button>          <button (click)="decrement()">decrement</button>          <child-component [count]=Counter></child-component>` ,    styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'Component Interaction';    Counter = 5;      increment() {      this.Counter++;    }    decrement() {      this.Counter--;    }  } |

The inline template in the Parent Component has two buttons. The Buttons Increments/decrements the counter.

|  |  |
| --- | --- |
| 1  2  3  4 | <button (click)="increment()">Increment</button>      <button (click)="decrement()">decrement</button> |

In the next line, we are invoking the child component inside

|  |  |
| --- | --- |
| 1  2  3 | <child-component [count]=Counter></child-component> |

Here, we are using count property, which is a property of the child Component inside the square bracket. We bind it to Counter property of the Parent Component.

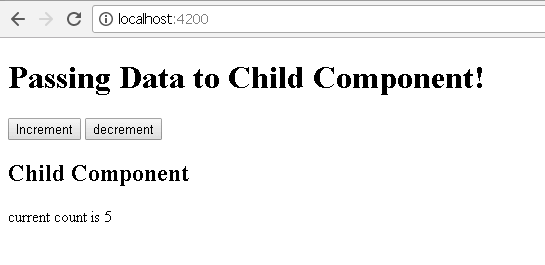
Remember square bracket represents the [Property Binding in Angular](https://www.tektutorialshub.com/angular/angular-data-binding/#property-binding).

Finally, we will add counter in Parent component and set it to 5 as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | export class AppComponent {    title = 'Component Interaction';    Counter = 5;      increment() {      this.Counter++;    }    decrement() {      this.Counter--;    }  } |

That’s it.

Now run the Code and you should see the following displayed in the browser



Click on Increment & Decrement buttons to see that the changes are propagated to the child component.

Various ways to use @Input Decorator

We used input @Input Decorator to mark the property in child component as input property. There are two ways you can do it Angular.

Using the @Input decorator to decorate the class property

Using the input array meta data of the component decorator

Using the @Input decorator to decorate the class property

We saw this in our above example.

|  |  |
| --- | --- |
| 1  2  3  4  5 | export class ChildComponent {       @Input() count: number;  } |

Using the input array metadata of the component decorator

The same result can be achieved by using Input array of the @Component decorator as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Component({      selector: 'child-component',      inputs: ['count'],      template: `<h2>Child Component</h2>      current count is {{ count }}  `  })  export class ChildComponent {} |

We have moved the count property to inputs array of the component metadata.

Aliasing input Property

You can Alias the input property and use the aliased name the parent component as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5 | export class ChildComponent {       @Input('MyCount') count: number;  } |

Here, we are aliasing count property with MyCount alias

In the parent component, we can use the MyCount as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5 | template: `     <h1>Welcome to {{title}}!</h1>     <child-component [MyCount]=ClickCounter></child-component> |

Detecting the Input changes

We looked at how to pass the data from parent to the child using @Input decorator and property binding.

Passing the data to child component is not sufficient, the child Component needs to know when the input changes so that it can act upon it.

There are two ways of detecting when input changes in the child component in Angular

Using [OnChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/) LifeCycle Hook

Using Input Setter

Let us look at both the methods in detail

Using OnChanges LifeCycle Hook

[ngOnChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/) is a lifecycle hook, which angular fires when it detects changes to data-bound input property. This method receives a [SimpeChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/#simplechanges) object, which contains the current and previous property values. We can Intercept input property changes in the child component using this hook.

How to use ngOnChanges for Change Detection

Import the OnChanges interface, SimpleChanges, SimpleChange from @angule/core library.

Implement the ngOnChanges() method. The method receives the SimpleChanges object containing the changes in each input property.

Let us update our Child Component to use the OnChanges hook

Open the child.component.ts and make the following changes

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | import { Component, Input, OnChanges, SimpleChanges, SimpleChange  } from '@angular/core';    @Component({      selector: 'child-component',      template: `<h2>Child Component</h2>                 current count is {{ count }}      `  })  export class ChildComponent implements OnChanges {      @Input() count: number;        ngOnChanges(changes: SimpleChanges) {            for (let property in changes) {              if (property === 'count') {                console.log('Previous:', changes[property].previousValue);                console.log('Current:', changes[property].currentValue);                console.log('firstChange:', changes[property].firstChange);              }          }      }  } |

First, we are Importing the required libraries

|  |  |
| --- | --- |
| 1  2  3 | import { Component, Input, OnChanges, SimpleChanges, SimpleChange  } from '@angular/core'; |

Next, Modify the class to implement the Onchanges interface

|  |  |
| --- | --- |
| 1  2  3 | export class ChildComponent implements OnChanges { |

ngOnChanges method

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | ngOnChanges(changes: SimpleChanges) {            for (let property in changes) {              if (property === 'count') {                console.log('Previous:', changes[property].previousValue);                console.log('Current:', changes[property].currentValue);                console.log('firstChange:', changes[property].firstChange);              }          }      } |

This method receives all the changes made to the input properties as SimpleChanges object. The SimpleChanges object whose keys are property names and values are instances of SimpleChange.

SimpleChange class Represents a basic change from a previous to a new value. It has three class members.

| **Property Name** | **Description** |
| --- | --- |
| previousValue:any | Previous value of the input property. |
| currentValue:any | New or current value of the input property. |
| FirstChange():boolean | Boolean value, which tells us whether it was the first time the change has taken place |

And we loop through the SimpleChanges to get our property count

Run the code and open the console log to watch the logs as you click on increment and decrement buttons in the parent component.

Using Input Setter

We can use the property getter and setter to detect the changes made to the input property as shown below

In the Child Component create a private property called \_count

|  |  |
| --- | --- |
| 1  2  3 | private \_count = 0; |

Create getter & setter on property count and attach @Input Decorator. We intercept the input changes from the setter function and log them to console

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Input()  set count(count: number) {      this.\_count = count;      console.log(count);  }  get count(): number { return this.\_count; } |

Summary

In this tutorial, we looked at how to pass data from parent to child Component. The Child Component decorates the property using the @Input Decorator. In the Parent Component, we use property binding to bind it to the Property or method of Parent Component.

We can also track changes made to the Input Property either by Using hooking to ngOnChanges life cycle hook. Or using the Property setter

Pass data from Child to parent component

There are three ways in which the parent component can interact with the child component

Listens to Child Event

Uses [Local Variable](https://www.tektutorialshub.com/angular/template-reference-variable-in-angular/) to access the child

Uses a [@ViewChild](https://www.tektutorialshub.com/angular/understanding-viewchild-viewchildren-querylist-in-angular/) to get the reference to the child component

Let us look at each of those scenarios in detail

Parent listens for child event

The Child Component exposes an [EventEmitter](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) Property. This Property is adorned with the [@Output](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) decorator. When Child Component needs to communicate with the parent it raises the event. The Parent Component listens to that event and reacts to it.

EventEmitter Property

To Raise an event, the component must declare an EventEmmitter Property. The Event can be emitted by calling the .emit() method

For Example

|  |  |
| --- | --- |
| 1  2  3 | countChanged: EventEmitter<number> = new EventEmitter() |

And then call emit method passing the whatever the data you want to send as shown below

|  |  |
| --- | --- |
| 1  2  3 | this.countChanged.emit(this.count); |

@Output Decorator

Using the EventEmitter Property gives the components ability to raise an event. But to make that event accessible from parent component, you must decorate the property with @Output decorator

How to Pass data to parent component using @Output

In the child component

Declare a property of type EventEmitter and instantiate it

Mark it with a @Output Decorator

Raise the event passing it with the desired data

In the Parent Component

Bind to the Child Component using [Event Binding](https://www.tektutorialshub.com/angular/angular-data-binding/#Event-Binding) and listen to the child events

Define the event handler function

Passing data to parent component Via Events (Example)

Now let us build an application to demonstrate this

In the last [passing data to child component](https://www.tektutorialshub.com/angular/angular-passing-data-child-component/) tutorial, we built a counter in the parent component. We assigned the initial value to the counter and added increment/decrement methods. In the child Component, we used the @Input decorator to bind count property to the parent component. Whenever parent count is changed in the parent the child component is updated and displayed the count.

In this tutorial, we will move the counter to the child component. We will raise an event in the child component whenever the count is increased or decreased. We then bind to that event in the parent component and display the count in the parent component.

Download the source code for this from the [GitHub](https://github.com/tekTutorialsHub/Angular-Advanced-Components) from the folder inputdecorator. The The final code is available in outputdecorator folder.

Child Component

Open the child.component.ts and copy the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import { Component, Input, Output, EventEmitter  } from '@angular/core';    @Component({      selector: 'child-component',      template: `<h2>Child Component</h2>                 <button (click)="increment()">Increment</button>                 <button (click)="decrement()">decrement</button>                 current count is {{ count }}      `  })  export class ChildComponent {      @Input() count: number;        @Output() countChanged: EventEmitter<number> =   new EventEmitter();        increment() {          this.count++;          this.countChanged.emit(this.count);        }      decrement() {          this.count--;          this.countChanged.emit(this.count);      }  } |

Now, let us look at the code in detail

First, as usual, we need to import output & EventEmitter from @angular/core

|  |  |
| --- | --- |
| 1  2  3 | import { Component, Input, Output, EventEmitter } from '@angular/core'; |

In the inline template, we have two buttons increment and decrement.  We also displaying the current count

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Component({      selector: 'child-component',      template: `<h2>Child Component</h2>                 <button (click)="increment()">Increment</button>                 <button (click)="decrement()">decrement</button>                 current count is {{ count }}      `  }) |

In the child component, define the countChanged event of type [EventEmiiter](https://angular.io/api/core/EventEmitter).  Decorate the property with @Output decorator to make it accessible from the parent component

|  |  |
| --- | --- |
| 1  2  3 | @Output() countChanged: EventEmitter<number> = new EventEmitter(); |

Finally, we raise the event in increment & decrement methods using emit

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | increment() {          this.count++;          this.countChanged.emit(this.count);        }      decrement() {          this.count--;          this.countChanged.emit(this.count);      } |

Parent Component

In the parent component , we need to listen to the “countChanged” event

The “countChanged” event is enclosed in Parentheses. It is then assigned to the method “countChangedHandler” in the component class. The syntax is similar to [Event Binding](https://www.tektutorialshub.com/angular/angular-data-binding/#Event-Binding)

|  |  |
| --- | --- |
| 1  2  3 | <child-component [count]=ClickCounter (countChanged)="countChangedHandler($event)"></child-component>` |

The countChangedHandler($event) method accepts the $event argument. The data associated with event is now available to in the $event property

Our CountChangedHandler is as follows. It just updates the clickCounter and also logs the count to console

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | countChangedHandler(count: number) {      this.ClickCounter = count;      console.log(count);    } |

The complete code is as follows

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import { Component} from '@angular/core';    @Component({    selector: 'app-root',    template: `          <h1>Welcome to {{title}}!</h1>          <p> current count is {{ClickCounter}} </p>          <child-component [count]=Counter (countChanged)="countChangedHandler($event)"></child-component>` ,    styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'Component Interaction';    Counter = 5;      countChangedHandler(count: number) {      this.Counter = count;      console.log(count);    }  } |

Run the code. Whenever the increment/decrement buttons clicked, The child raises the event. The Parent component gets notified of the this and updates the counter with the latest value.

Parent uses local variable to access the Child in Template

Parent Template can access the child component properties and methods by creating the template reference variable

Child Component

Let us update the child component

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import { Component} from '@angular/core';    @Component({      selector: 'child-component',      template: `<h2>Child Component</h2>                 current count is {{ count }}      `  })  export class ChildComponent {      count = 0;         increment() {          this.count++;        }      decrement() {          this.count--;      }  } |

We have removed the input, output & eventemiitter.

Our component is now have property count and two methods to increment and decrement it

Parent component

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | import { Component} from '@angular/core';    @Component({    selector: 'app-root',    template: `          <h1>{{title}}!</h1>          <p> current count is {{child.count}} </p>          <button (click)="child.increment()">Increment</button>          <button (click)="child.decrement()">decrement</button>          <child-component #child></child-component>` ,    styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'Parent interacts with child via local variable';  } |

We have created a local variable, #child, on the tag <child-component>. The “child” is called template reference variable, which now represents the child component

The Template Reference variable is created, when you use #<varibaleName> and attach it to a DOM element. You can then, use the variable to reference the DOM element in your Template

|  |  |
| --- | --- |
| 1  2  3 | <child-component #child></child-component>` , |

Now you can use the local variable elsewhere in the template to refer to the child component methods and properties as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5 | <p> current count is {{child.count}} </p>          <button (click)="child.increment()">Increment</button>          <button (click)="child.decrement()">decrement</button> |

The code above wires child components increment & decrement methods from the parent component

The local variable approach is simple and easy. But it is limited because the parent-child wiring must be done entirely within the parent template. The parent component itself has no access to the child.

You can’t use the local variable technique if an instance of the parent component class must read or write child component values or must call child component methods.

Parent uses a @ViewChild() to get reference to the Child Component

Injecting an instance of the child component into the parent as a @ViewChild is the another technique used by the parent to access the property and method of the child component

The @ViewChild decorator takes the name of the component/directive as its input. It is then used to decorate a property. The Angular then injects the reference of the component to the Property

For Example

In the Parent component, declare a property child which is of type ChildComponent

|  |  |
| --- | --- |
| 1  2  3 | child: ChildComponent; |

Next, decorate it with @ViewChild decorator passing it the name of the component to inject

|  |  |
| --- | --- |
| 1  2  3 | @ViewChild(ChildComponent) child: ChildComponent; |

Now, when angular creates the child component, the reference to the child component is assigned to the child property.

We now update the code from previous section

Child Component

There is no change in the child component

Parent Component

In the parent component, we need to import the viewChild Decorator. We also need to import the child component

|  |  |
| --- | --- |
| 1  2  3  4 | import { Component, ViewChild } from '@angular/core';  import { ChildComponent } from './child.component'; |

Next, create a property child which is an instance of type ChildComponent. Apply the viewChild Decorator on the child component as shown below

|  |  |
| --- | --- |
| 1  2  3 | @ViewChild(ChildComponent) child: ChildComponent; |

Finally, add increment and decrement method, which invokes the methods in the child component

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | increment() {      this.child.increment();    }      decrement() {      this.child.decrement();    } |

Now, the parent can access the properties and methods of child component

And in the template make necessary changes

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <h1>{{title}}</h1>          <p> current count is {{child.count}} </p>          <button (click)="increment()">Increment</button>          <button (click)="decrement()">decrement</button>          <child-component></child-component>` |

The complete app.component.ts is as follows

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | import { Component, ViewChild } from '@angular/core';  import { ChildComponent } from './child.component';    @Component({    selector: 'app-root',    template: `          <h1>{{title}}</h1>          <p> current count is {{child.count}} </p>          <button (click)="increment()">Increment</button>          <button (click)="decrement()">decrement</button>          <child-component></child-component>` ,    styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'Parent calls an @ViewChild()';      @ViewChild(ChildComponent) child: ChildComponent;      increment() {      this.child.increment();    }      decrement() {      this.child.decrement();    }  } |

Conclusion

In this tutorial, we looked at how the parent can communicate with the child component. The Parent can subscribe to the events of the child component. It can use the Template local variable to access the properties and methods. We can also use @ViewChild decorator to inject the child component instance to the parent

# Component Life Cycle Hook

What is Angular Component lifecycle hooks

When the angular application starts it creates and renders the root component. It then creates and renders its Childrens & their children. It forms a [tree of components](https://www.tektutorialshub.com/angular/angular-architecture-overview-concepts/#a-typical-angular-application).

Once Angular loads the components, it starts the process of rendering the view.  To do that it needs to check the input properties, evaluate the data bindings & expressions, render the projected content etc. Angular also removes the component from the DOM, when it is no longer needs it.

Angular lets us know when these events happen using lifecycle hooks

The Angular life cycle hooks are nothing but callback function, which angular invokes when a certain event occurs during the component’s life cycle.

For example,

[ngOnInit](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/) when Angular initializes the component for the first time.

When a component’s input property change, Angular invokes [ngOnChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/)

If the component is destroyed, Angular invokes [ngOnDestroy](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/)

Angular lifecycle hooks

Here is the complete list of life cycle hooks, which angular invokes during the component life cycle. Angular invokes them when a certain event occurs.

ngOnChanges

ngOnInit

ngDoCheck

ngAfterContentInit

ngAfterContentChecked

ngAfterViewInit

ngAfterViewChecked

ngOnDestroy

Ad by Valueimpression

Change detection Cycle

Before diving into the lifecycle hooks, we need to understand the change detection cycle.

Change detection is the mechanism by which angular keeps the template in sync with the component

Consider the following code.

|  |  |
| --- | --- |
| 1  2  3  4 | <div>Hello {{name}}</div> |

Angular updates the DOM, whenever the value of the name changes. And it does it instantly.

How does angular know when the value of name changes?. It does so by running a change detection cycle on every event that may result in a change. It runs it on every input changes, DOM events, timer events like setTimeout() and setInterval() , http requests etc.

During the change detection cycle angular checks each and every bound property in the template, with that of the component class. If it detects any changes it updates the DOM.

Angular raises the life cycle hooks during the important stages of the change detection  mechanism.

Constructor

Life Cycle of a component begins, when Angular creates the component class. First method that gets invoked is class Constructor.

Constructor is neither a life cycle hook nor it is specific to Angular.  It is a Javascript feature. It is a method which is invoked, when a class is created.

Angular makes use of a constructor to [inject dependencies](https://www.tektutorialshub.com/angular/angular-dependency-injection/).

At this point, none of the components input properties are available to use. Neither its child components are constructed. Projected contents are also not available.

Hence there is not much you can do in this method. And also it is recommend not to use it

Once Angular instantiates the class, It kick-start the first change detection cycle of the component.

ngOnChanges

The Angular invokes [ngOnChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/) life cycle hook whenever any data-bound input property of the component or directive changes. Initializing the Input properties is the first task that angular carries during the change detection cycle. And if it detects any change in property, then it raises the [ngOnChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/) hook. It does so during every change detection cycle. This hook is not raised if change detection does not detect any changes.

[Input](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) properties are those properties, which we define using the [@Input decorator](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/). It is one of the ways by which a parent communicates with the child component.

In the following example, the child component declares the property message as the [input property](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/)

|  |  |
| --- | --- |
| 1  2  3 | @Input() message:string |

The parent can send the data to the child using the [property binding](https://www.tektutorialshub.com/angular/property-binding-in-angular/) as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <app-child [message]="message">  </app-child> |

The change detector checks if such input properties of a component are changed by the parent component. If it is then it raises the [ngOnChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/) hook.

We used this life cycle hook in the tutorial [Passing data to child component](https://www.tektutorialshub.com/angular/angular-passing-data-child-component/).

The change detector uses the === [strict equality operator](https://www.tektutorialshub.com/typescript/strict-equality-loose-equality-in-typescript/) for detecting changes. Hence for objects, the hook is fired only if the references are changed. You can read more about it from [Why ngOnChanges does not fire](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/#onchanges-does-not-fire-always).

ngOnInit

The Angular raises the [ngOnInit](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/) hook, after it creates the component and updates its input properties. It raises it after the [ngOnChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/) hook.

This hook is fired **only once** and immediately after its creation (during the first change detection).

This is a perfect place where you want to add any initialisation logic for your component.  Here you have access to every input property of the component. You can use them in  http get requests to get the data from the back end server or run some initialization logic etc.

But note that none of child components or projected content are available at this point. Hence any properties we decorate with [@ViewChild](https://www.tektutorialshub.com/angular/understanding-viewchild-viewchildren-querylist-in-angular/), [@ViewChildren](https://www.tektutorialshub.com/angular/understanding-viewchild-viewchildren-querylist-in-angular/) , [@ContentChild](https://www.tektutorialshub.com/angular/contentchild-and-contentchildren-in-angular/) & [@ContentChildren](https://www.tektutorialshub.com/angular/contentchild-and-contentchildren-in-angular/) will not be available to use.

ngDoCheck

The Angular invokes the [ngDoCheck](https://www.tektutorialshub.com/angular/angular-ngdocheck-life-cycle-hook/) hook event during every change detection cycle. This hook is invoked even if there is no change in any of the properties.

Angular invokes it after the [ngOnChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/) & [ngOnInit](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/) hooks.

Use this hook to Implement a custom change detection, whenever Angular fails to detect the changes made to Input properties. This hook is particularly useful when you opt for the Onpush change detection strategy.

The Angular [ngOnChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/) hook [does not detect all the changes made to the input properties](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/#onchanges-does-not-fire-always).

ngAfterContentInit

ngAfterContentInit Life cycle hook is called after the Component’s [projected content](https://www.tektutorialshub.com/angular/ng-content-content-projection-in-angular/) has been fully initialized. Angular also updates the properties decorated with the [ContentChild and ContentChildren](https://www.tektutorialshub.com/angular/contentchild-and-contentchildren-in-angular/) before raising this hook. This hook is also raised, even if there is no content to project.

The content here refers to the external content injected from the parent component via [Content Projection](https://www.tektutorialshub.com/angular/ng-content-content-projection-in-angular/).

The [Angular Component](https://www.tektutorialshub.com/angular/angular-component/)s can include the [ng-content](https://www.tektutorialshub.com/angular/ng-content-content-projection-in-angular/) element, which acts as a placeholder for the content from the parent as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5 | <h2>Child Component</h2>  <ng-content></ng-content>   <!-- placehodler for content from parent --> |

Parent injects the content between the opening & closing element.  Angular passes this content to the child component

|  |  |
| --- | --- |
| 1  2  3  4  5 | <h1>Parent Component</h1>  <app-child> This <b>content</b> is injected from parent</app-child> |

During the change detection cycle, Angular checks if the injected content has changed and updates the DOM.

This is a component only hook.

ngAfterContentChecked

ngAfterContentChecked Life cycle hook is called during every change detection cycle after Angular finishes checking of component’s projected content. Angular also updates the properties decorated with the [ContentChild and ContentChildren](https://www.tektutorialshub.com/angular/contentchild-and-contentchildren-in-angular/) before raising this hook. Angular calls this hook even if there is no projected content in the component

This hook is very similar to the ngAfterContentInit hook. Both are called after the external content is initialized, checked & updated. Only difference is that ngAfterContentChecked is raised after every change detection cycle. While ngAfterContentInit during the first change detection cycle.

This is a component only hook.

ngAfterViewInit

ngAfterViewInit hook is called after the Component’s View & all its child views are fully initialized. Angular also updates the properties decorated with the [ViewChild](https://www.tektutorialshub.com/angular/understanding-viewchild-viewchildren-querylist-in-angular/) & [ViewChildren](https://www.tektutorialshub.com/angular/understanding-viewchild-viewchildren-querylist-in-angular/) properties before raising this hook.

The View here refers to the template of the current component and all its child components & directives.

This hook is called during the first change detection cycle, where angular initializes the view for the first time

At this point all the lifecycle hook methods & change detection  of all child components & directives are processed & Component is completely ready

This is a component only hook.

ngAfterViewChecked

The Angular fires this hook after it checks & updates the component’s views and child views. This event is fired after the ngAfterViewInit and after that during every change detection cycle

This hook is very similar to the ngAfterViewInit hook. Both are called after all the child components & directives are initialized and updated. Only difference is that ngAfterViewChecked is raised during every change detection cycle. While ngAfterViewInit during the first change detection cycle.

This is a component only hook.

ngOnDestroy

This hook is called just before the Component/Directive instance is [destroyed by Angular](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/)

You can Perform any cleanup logic for the Component here. This is the correct place where you would like to Unsubscribe Observables and detach event handlers to avoid memory leaks.

How to Use Lifecycle Hooks

Import Hook interfaces

Declare that Component/directive Implements lifecycle hook interface

Create the hook method

Let us build a simple component, which implements the **ngOnInit hook**

Create a [Angular Project using Angular Cli](https://www.tektutorialshub.com/angular/angular-create-first-application/). Open the app.component.ts

Import Hook interfaces

Import hook interfaces from the core module. The name of the Interface is hook name without ng. For example interface of ngOnInit hook is OnInit

|  |  |
| --- | --- |
| 1  2  3 | import { Component,OnInit } from '@angular/core' |

Component Implements lifecycle hook interface

Next, define the AppComponent to implement OnInit interface

|  |  |
| --- | --- |
| 1  2  3 | export class AppComponent implements OnInit { |

Create the hook method

The life cycle hook methods must use the same name as the hook.

|  |  |
| --- | --- |
| 1  2  3  4  5 | ngOnInit() {      console.log("AppComponent:OnInit");    } |

The complete code for the app.component.ts.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import { Component,OnInit } from '@angular/core';    @Component({    selector: 'app-root',    template: `        <h2>Life Cycle Hook</h2>` ,    styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {      constructor() {      console.log("AppComponent:Constructor");    }      ngOnInit() {      console.log("AppComponent:OnInit");    }    } |

Now, run the code and open the developer console. you will see the following

|  |  |
| --- | --- |
| 1  2  3  4 | AppComponent:Constructor  AppComponent:OnInit |

Note that the constructor event is fired before the OnInit hook.

The Order of Execution of Life Cycle Hooks

The Angular executes the hooks in the following order

On Component Creation

OnChanges

OnInit

DoCheck

AfterContentInit

AfterContentChecked

AfterViewInit

AfterViewChecked

When the Component with Child Component is created

OnChanges

OnInit

DoCheck

AfterContentInit

AfterContentChecked

Child Component -> OnChanges

Child Component -> OnInit

Child Component -> DoCheck

Child Component -> AfterContentInit

Child Component -> AfterContentChecked

Child Component -> AfterViewInit

Child Component -> AfterViewChecked

AfterViewInit

AfterViewChecked

After The Component is Created

OnChanges

DoCheck

AfterContentChecked

AfterViewChecked

The OnChanges hook is fired only if there is an input property defined in the component and it changes. Otherwise, it will never fire

Angular Lifecycle hook Example

[Source Code](https://stackblitz.com/edit/angular-component-life-cycle-hooks-example-1)

**app.component.ts**

import { ChangeDetectionStrategy, Component, VERSION } from "@angular/core";

@Component({

selector: "my-app",

changeDetection:ChangeDetectionStrategy.Default,

template: `

<h1>Angular Life Cycle Hooks</h1>

Reference :

<a

href="https://www.tektutorialshub.com/angular/angular-component-life-cycle-hooks/#create-the-hook-method"

>Angular Life Cycle Hooks</a

>

<h1>Root Component</h1>

<br />

<input

type="text"

name="message"

[(ngModel)]="message"

autocomplete="off"

/>

<br />

<input

type="text"

name="content"

[(ngModel)]="content"

autocomplete="off"

/>

<br />

hide child :

<input

type="checkbox"

name="hideChild"

[(ngModel)]="hideChild"

autocomplete="off"

/>

<br />

<br />

<app-child [message]="message" \*ngIf="!hideChild">

<!-- Injected Content -->

<b> {{ content }} </b>

</app-child>

`

})

export class AppComponent {

name = "Angular " + VERSION.major;

message = "Hello";

content = "Hello";

hideChild=false;

constructor() {

console.log("AppComponent:Contructed");

}

ngOnChanges() {

console.log("AppComponent:ngOnChanges");

}

ngOnInit() {

console.log("AppComponent:ngOnInit");

}

ngDoCheck() {

console.log("AppComponent:DoCheck");

}

ngAfterContentInit() {

console.log("AppComponent:ngAfterContentInit");

}

ngAfterContentChecked() {

console.log("AppComponent:AfterContentChecked");

}

ngAfterViewInit() {

console.log("AppComponent:AfterViewInit");

}

ngAfterViewChecked() {

console.log("AppComponent:AfterViewChecked");

}

ngOnDestroy() {

console.log("AppComponent:ngOnDestroy");

}

}

We are listening to all the hooks and logging them to the console.

There are two form fields message & content. We pass both to the child component. One as input property & the other via content projection

Using the hideChild form field we an add or remove the ChildComponent from the DOM. We are making use of the [ngIf](https://www.tektutorialshub.com/angular/angular-ngif-directive/) directive.

We pass the message property to ChildComponent using Property binding.

The content property is passed as projected content.

**child.component.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77 | import { ChangeDetectionStrategy, Component, Input,  OnInit } from '@angular/core';  import { Customer } from './customer';    @Component({    selector: 'app-child',    changeDetection:ChangeDetectionStrategy.Default,    template: `          <h2>child component</h2>          <br>        <!-- Data as a input -->        Message from Parent via @input {{message}}        <br><br>        <!-- Injected Content -->        Message from Parent via content injection        <ng-content></ng-content>          <br><br><br>        Code :        <input type="text" name="code" [(ngModel)]="customer.code" autocomplete="off">        <br><br>        Name:        <input type="text" name="name" [(ngModel)]="customer.name" autocomplete="off">          <app-grand-child [customer]="customer"></app-grand-child>      `    })  export class ChildComponent {      @Input() message:string      customer:Customer = new Customer()      constructor() {      console.log("  ChildComponent:Contructed");    }      ngOnChanges() {      console.log("  ChildComponent:ngOnChanges");    }      ngOnInit() {      console.log("  ChildComponent:ngOnInit");    }      ngDoCheck() {      console.log("  ChildComponent:DoCheck");    }      ngAfterContentInit() {      console.log("  ChildComponent:ngAfterContentInit");    }      ngAfterContentChecked() {      console.log("  ChildComponent:AfterContentChecked");    }      ngAfterViewInit() {      console.log("  ChildComponent:AfterViewInit");    }      ngAfterViewChecked() {      console.log("  ChildComponent:AfterViewChecked");    }      ngOnDestroy() {      console.log("  ChildComponent:ngOnDestroy");    }    } |

We are listening to all the hooks

[@Input](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) decorator marks the message as input property. It will receive the data from the parent

<ng-content></ng-content> is the place holder to receive the projected content from the parent.

Two forms fields for customer object, which we pass it to the GrandChildComponent

**grandchild.component.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | import { ChangeDetectionStrategy, Component, Input,  OnInit } from '@angular/core';  import { Customer } from './customer';    @Component({    selector: 'app-grand-child',    changeDetection:ChangeDetectionStrategy.Default,    template: `          <h3>grand child component </h3>          <br>        Name {{customer.name}}      `,  })  export class GrandChildComponent {        @Input() customer:Customer      constructor() {      console.log("    GrandChildComponent:Contructed");    }      ngOnChanges() {      console.log("    GrandChildComponent:ngOnChanges");    }      ngOnInit() {      console.log("    GrandChildComponent:ngOnInit");    }        ngDoCheck() {      console.log("    GrandChildComponent:DoCheck");    }      ngAfterContentInit() {      console.log("    GrandChildComponent:ngAfterContentInit");    }      ngAfterContentChecked() {      console.log("    GrandChildComponent:AfterContentChecked");    }      ngAfterViewInit() {      console.log("    GrandChildComponent:AfterViewInit");    }      ngAfterViewChecked() {      console.log("    GrandChildComponent:AfterViewChecked");    }      ngOnDestroy() {      console.log("    GrandChildComponent:ngOnDestroy");    }        } |

We are listening to all the hooks

[@Input](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) decorator marks the customer as input property. It will receive the data from the parent

Run the code and check the console for the log messages

Conclusion

We learned about Component life cycle hooks in Angular.  The Angular generates following hooks OnChanges, OnInit, DoCheck, AfterContentInit, AfterContentChecked, AfterViewInit, AfterViewChecked & OnDestroy. We then learned how to build an Application using OnInit life cycle hook. Finally, we looked at the Order of execution of these life cycle hooks

ngOnInit

The ngOnInit or OnInit hook is called when the component is created for the first time. This hook is called after the constructor and first ngOnChanges hook is fired.

This is a perfect place where you want to add any initialization logic for your component.

Note that ngOnChanges hook is fired before ngOnInit. Which means all the input properties are available to use when the ngOnInit is hook is called

This hook is fired only once

This hook is fired before any of the child directive properties are initialized.

ngOnDestroy

The ngOnDestroy or OnDestroy hook is called just before the Component/Directive instance is destroyed by Angular

Use this hook to Perform any cleanup logic for the Component. This is the correct place where you would like to Unsubscribe Observables and detach event handlers to avoid memory leaks.

Example of ngOnInit

Let us build a Component that illustrates the use on OnInit and OnDestroy hook

Let us build a Child component, which is conditionally displayed or destroyed based on flag from the Parent Component

Child Component

Create the child.component.ts

First Import the OnDestroy and OnInit from the angular/core library

|  |  |
| --- | --- |
| 1  2  3 | import { Component, OnDestroy, OnInit } from '@angular/core'; |

The Component template just displays the title “Child Component”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Component({    selector: 'child-component',    template: `        <h2>Child Component</h2>        ` ,    styleUrls: ['./app.component.css']  }) |

Declare child Component implements OnInint and OnDestroy Hooks

|  |  |
| --- | --- |
| 1  2  3 | export class ChildComponent implements OnInit, OnDestroy { |

Add the constructor and add to log when the constructor is called

|  |  |
| --- | --- |
| 1  2  3  4  5 | constructor() {      console.log('ChildComponent:Constructor');    } |

Finally. Create the hook method. The method writes to the console log

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | ngOnInit() {      console.log('ChildComponent:OnInit');    }      ngOnDestroy() {      console.log('ChildComponent:OnDestroy');    } |

The complete code for child component

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import { Component, OnDestroy, OnInit } from '@angular/core';    @Component({    selector: 'child-component',    template: `        <h2>Child Component</h2>        ` ,    styleUrls: ['./app.component.css']  })  export class ChildComponent implements OnInit, OnDestroy {      constructor() {      console.log('ChildComponent:Constructor');    }      ngOnInit() {      console.log('ChildComponent:OnInit');    }      ngOnDestroy() {      console.log('ChildComponent:OnDestroy');    }    } |

Parent Component

Here’s how our App Component looks like

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | import { Component, OnInit, OnDestroy } from '@angular/core';    @Component({    selector: 'app-root',    template: `        <h2>Life Cycle Hook</h2>        <button (click)="toggle()">Hide/Show Child </button>        <child-component \*ngIf="displayChild"></child-component>        ` ,    styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit, OnDestroy {      displayChild = true;      constructor() {      console.log('AppComponent:Constructor');    }      toggle() {      this.displayChild = !this.displayChild;    }      ngOnInit() {      console.log('AppComponent:OnInit');    }        ngOnDestroy() {      console.log('AppComponent:OnDestroy');    }  } |

Note that the parent template

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | template: `        <h2>Life Cycle Hook</h2>        <button (click)="toggle()">Hide/Show Child </button>        <child-component \*ngIf="displayChild"></child-component>        ` , |

We have used \*[ngIf directive](https://www.tektutorialshub.com/angular/angular-ngif-directive/), which hides/shows the child component based on the displaychild value. The toggle function toggle the status of the displaychild.

We have added OnInit and OnDestroy hook to parent component also.

Run the Code

When you run the code for the first time you will see the following in the console window

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | AppComponent: Constructor  AppComponent: OnInit  ChildComponent:Constructor  ChildComponent:OnInit |

Click on the toggle button. The Child Component is destroyed and you will see the following logs

|  |  |
| --- | --- |
| 1  2  3 | ChildComponent:OnDestroy |

Click on the toggle button again. The Child Component is created again and you will see that the constructor of the child component is called again and then the OnInit is invoked

|  |  |
| --- | --- |
| 1  2  3  4 | ChildComponent:Constructor  ChildComponent:OnInit |

The above code demonstrates how the OnInit & OnDestroy works

Difference Between Constructor and ngOnInit

The Constructor is executed when the class is instantiated. It has nothing do with the angular. It is the feature of Javascript and Angular does not have the control over it

The ngOnInit is Angular specific and is called when the Angular has initialized the component with all its input properties

The @Input properties are available under the ngOnInit lifecycle hook. This will help you to do some initialization stuff like getting data from the back-end server etc to display in the view

@Input properties are shows up as undefined inside the constructor

Conclusion

OnInit Hook is useful for Initialising the Component like getting data from back-end server, while OnDestroy hook must be used to perform clean-up operation in the Component.

What is nOnChanges Life cycle hook

The ngOnChnages is a [life cycle hook](https://www.tektutorialshub.com/angular/angular-component-life-cycle-hooks/), which angular fires when it detects changes to [data-bound input property](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/). This method receives a SimpeChanges object, which contains the current and previous property values.

There are several ways the parent component can communicate with the child component. One of the ways is to use the [@Input decorator](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/). We looked at this in our tutorial [passing data to Child Components](https://www.tektutorialshub.com/angular/angular-passing-data-child-component/)

Let us just recap what we have done in that tutorial

The child Component decorates the property using the [@Input](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) decorator.

|  |  |
| --- | --- |
| 1  2  3 | @Input() message: string; |

And then parent passes the data to the child component using [property binding](https://www.tektutorialshub.com/angular/property-binding-in-angular/) as shown below

|  |  |
| --- | --- |
| 1  2  3 | <child-component [message]=message></child-component>` |

Whenever the parent changes the value of the message property,  the Angular raises the OnChanges hook event in the child component, so that it can act upon it.

How does it work

The ngOnChanges() method takes an object that maps each changed property name to a SimpleChange object, which holds the current and previous property values. You can iterate over the changed properties and act upon it.

SimpleChange

SimpleChange is a simple class, which has three properties

| **Property Name** | **Description** |
| --- | --- |
| previousValue:any | Previous value of the input property. |
| currentValue:any | New or current value of the input property. |
| FirstChange():boolean | Boolean value, which tells us whether it was the first time the change has taken place |

SimpleChanges

Every [@Input](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) property of our component gets a SimpleChange object (if Property is changed)

SimpleChanges is the object that contains the instance of all those SimpleChange objects. You can access those SimpleChange objects using the name of the [@Input](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) property as the key

For Example, if the two [Input properties](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) message1 & message2 are changed, then the SimpleChanges object looks like

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | {    "message1": { "previousValue":"oldvalue",                  "currentValue":"newvalue",                  "firstChange":false }    },    "message2": { "previousValue":"oldvalue",                  "currentValue":"newvalue",                  "firstChange":false }    }  } |

And if the input property is an object (customer object with name & code property) then the SimpleChanges would be

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | {  "Customer":      {"previousValue":{"name":"Angular","code":"1"},       "currentValue":{"name":"Angular2","code":"1"},       "firstChange":false}  } |

ngOnChanges example

Create a class customer.ts under src/app folder.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | export class Customer {    code: number;    name: string;  } |

Parent Component

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | import { Component} from '@angular/core';  import { Customer } from './customer';    @Component({    selector: 'app-root',    template: `          <h1>{{title}}!</h1>          <p> Message : <input type='text' [(ngModel)]='message'> </p>          <p> Code : <input type='text' [(ngModel)]='code'></p>          <p> Name : <input type='text' [(ngModel)]='name'></p>          <p><button (click)="updateCustomer()">Update </button>          <child-component [message]=message [customer]=customer></child-component>          ` ,            styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'ngOnChanges';    message = '';    customer: Customer = new Customer();    name= '';    code= 0;      updateCustomer() {      this.customer.name = this.name;      this.customer.code = this.code;    }    } |

Lets us look at the code

We have 3 user input fields for the message, code and name. The UpdateCustomer button updates the Customer object.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <p> Message : <input type='text' [(ngModel)]='message'> </p>  <p> Code : <input type='text' [(ngModel)]='code'></p>  <p> Name : <input type='text' [(ngModel)]='name'></p>  <p><button (click)="updateCustomer()">Update </button> |

The message and Customer is bound to the child component using the [property binding](https://www.tektutorialshub.com/angular/property-binding-in-angular/)

|  |  |
| --- | --- |
| 1  2  3 | <child-component [message]=message [customer]=customer></child-component> |

The AppComponent class has a message & customer property. We update customer object with new code & name when user clicks the updateCustomer button.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | export class AppComponent {    title = 'ngOnChanges';    message = '';    customer: Customer = new Customer();    name= '';    code= 0;      updateCustomer() {      this.customer.name = this.name;      this.customer.code = this.code;    }  } |

Child Component

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import { Component, Input, OnInit, OnChanges, SimpleChanges, SimpleChange,ChangeDetectionStrategy  } from '@angular/core';  import { Customer } from './customer';    @Component({      selector: 'child-component',      template: `<h2>Child  Component</h2>                 <p>Message {{ message }} </p>                 <p>Customer Name {{ customer.name }} </p>                 <ul><li \*ngFor="let log of changelog;"> {{ log }}</li></ul> `  })  export class ChildComponent implements OnChanges, OnInit {      @Input() message: string;      @Input() customer: Customer;      changelog: string[] = [];        ngOnInit() {          console.log('OnInit');      }        ngOnChanges(changes: SimpleChanges) {          console.log('OnChanges');          console.log(JSON.stringify(changes));            // tslint:disable-next-line:forin          for (const propName in changes) {               const change = changes[propName];               const to  = JSON.stringify(change.currentValue);               const from = JSON.stringify(change.previousValue);               const changeLog = `${propName}: changed from ${from} to ${to} `;               this.changelog.push(changeLog);          }      }  } |

Let us look at each line of code in detail

First, We import the Input, OnInit, OnChanges, SimpleChanges, SimpleChange from Angular Core

|  |  |
| --- | --- |
| 1  2  3 | import { Component, Input, OnInit, OnChanges, SimpleChanges, SimpleChange } from '@angular/core'; |

The Template displays the message & name property from the customer object. Both these properties are updated from the parent component.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | template: `<h2>Child  Component</h2>                 <p>Message {{ message }} </p>                 <p>Customer Name {{ customer.name }} </p> |

We also display the changelog using [ngFor Directive](https://www.tektutorialshub.com/angular/angular-ngfor-directive/).

|  |  |
| --- | --- |
| 1  2  3 | <ul><li \*ngFor="let log of changelog;"> {{ log }}</li></ul> ` |

The child Component implements the OnChanges & [OnInit life cycle hooks](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/).

|  |  |
| --- | --- |
| 1  2  3  4  5 | export class ChildComponent implements OnChanges, OnInit { |

We also define message & customer property, which we decorate with the [@Input decorator](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/). The parent component updates these properties via [Property Binding](https://www.tektutorialshub.com/angular/property-binding-in-angular/).

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Input() message: string;     @Input() customer: Customer;     changelog: string[] = []; |

The [OnInit hook](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/)

|  |  |
| --- | --- |
| 1  2  3  4  5 | ngOnInit() {          console.log('OnInit');      } |

The ngOnChnages hook gets all the changes as an instance of SimpleChanges. This object contains the instance of SimpleChange for each property

|  |  |
| --- | --- |
| 1  2  3  4  5 | ngOnChanges(changes: SimpleChanges) {          console.log('OnChanges');          console.log(JSON.stringify(changes)); |

We, then loop through each property of the SimpleChanges object and get a reference to the SimpleChange object.

|  |  |
| --- | --- |
| 1  2  3  4 | for (const propName in changes) {               const change = changes[propName]; |

Next, we will take the current & previous value of each property and add it to change log

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | const to  = JSON.stringify(change.currentValue);               const from = JSON.stringify(change.previousValue);               const changeLog = `${propName}: changed from ${from} to ${to} `;               this.changelog.push(changeLog);          }      }  } |

That’s it.

Now our OnChanges hook is ready to use.

Now, run the code and type the Hello and you will see the following log

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | message: changed from undefined to ""  customer: changed from undefined to {}  message: changed from "" to "H"  message: changed from "H" to "He"  message: changed from "He" to "Hel"  message: changed from "Hel" to "Hell"  message: changed from "Hell" to "Hello" |

Open the developer console and you should see the changes object

Note that the first OnChanges fired before the [OnInit hook](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/). This ensures that initial values bound to inputs are available when ngOnInit() is called

OnChanges does not fire always

Now, change the customer code and name and click UpdateCustomer button.

The Child Components displays customer Name, but OnChanges event does not fire.

This behavior is by design.

Template is Updated

Updating the DOM is part of Angular’s change detection mechanism The change detector checks each and every bound property for changes and updates the DOM if it finds any changes.

In the child component template, we have two bound properties. {{ message }} & {{ customer.name }}. Hence the change detector checks only these two properties and updates the DOM. The customer object also has code property. The change detector will never check it.

Why onChanges does not fire?

The Change detector also raises the OnChanges hook. But it uses a different techniques for comparison.

The change detector uses the [=== strict equality operator](https://www.tektutorialshub.com/typescript/strict-equality-loose-equality-in-typescript/) for detecting changes to the [input properties](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/). For [primitive data types](https://www.tektutorialshub.com/typescript/typescript-data-types/) like string, the above comparison works perfectly

But in the case of an object like a customer, this fails. For Arrays/objects, the strict checking means that only the references are checked. Since the reference to the customer stays the same the Angular does not raise the OnChanges hook.

That leaves us two possible solutions

Create a new customer and copy the old data to new customer

We can Perform our own change detection using the ngDoCheck lifecycle hook

Update the updateCustomer method and create a new instance of customer every time

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | updateCustomer() {      this.customer= new Customer();    //Add this      this.customer.name = this.name;      this.customer.code = this.code;    } |

Now, run the code, you will see onChanges event fired when customer is updated

What is ngDoCheck lifecycle hook

We looked at how [OnChanges](https://www.tektutorialshub.com/angular/angular-ngonchanges-life-cycle-hook/) hook works in the Previous chapter. It is triggered every time when the Angular detected a change to the data-bound input property

We also looked at how OnChanges does not fire when the input property is an array/object because Angular uses dirty checking to compare the properties.

In such a scenario, where Angular fails to detect the changes to the input property, the DoCheck allows us to implement our custom change detection.

The Angular Fires the DoCheck hook after each change detection

ngDoCheck example

Let us build on the code, we built in the previous tutorial on Onchanges

customer.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | export class Customer {      code: number;      name: string;  } |

There is no change in the app.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | import { Component} from '@angular/core';  import { Customer } from './customer';    @Component({    selector: 'app-root',    template: `          <h1>{{title}}!</h1>          <p> Message : <input type='text' [(ngModel)]='message'> </p>          <p> Code : <input type='text' [(ngModel)]='code'></p>          <p> Name : <input type='text' [(ngModel)]='name'></p>          <p><button (click)="updateCustomer()">Update </button>          <child-component [message]=message [customer]=customer></child-component>          ` ,            styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'ngOnChanges';    message = '';    customer: Customer = new Customer();    name= '';    code= 0;      updateCustomer() {      this.customer.name = this.name;      this.customer.code = this.code;    }    } |

Child Component

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | import { Component, Input, OnChanges, OnInit, SimpleChanges, SimpleChange, DoCheck  } from '@angular/core';  import { Customer } from './customer';    @Component({      selector: 'child-component',      template: `<h2>Child  Component</h2>                 <p>Message {{ message }} </p>                 <p>Customer Name {{ customer.name }} </p>                 <p>Customer Code {{ customer.code }} </p>                 <p>Do Check count {{ DocheckCount }} </p>                 <ul><li \*ngFor="let log of changelog;"> {{ log }}</li></ul> `  })  export class ChildComponent implements OnChanges, DoCheck, OnInit {      @Input() message: string;      @Input() customer: Customer;      changelog: string[] = [];      oldCustomer: Customer= new Customer();      DocheckCount = 0;        ngOnInit() {          console.log('OnInit');          this.oldCustomer = Object.assign({}, this.customer);      }        ngDoCheck() {          console.log('Docheck');          this.DocheckCount++;          if (this.oldCustomer.name !== this.customer.name || this.oldCustomer.code !== this.customer.code ) {              const to  = JSON.stringify(this.customer);              const from = JSON.stringify(this.oldCustomer);              const changeLog = `DoCheck customer: changed from ${from} to ${to} `;              this.changelog.push(changeLog);                this.oldCustomer = Object.assign({}, this.customer);          }      }        ngOnChanges(changes: SimpleChanges) {          console.log('OnChanges');          console.log(JSON.stringify(changes));            // tslint:disable-next-line:forin          for (const propName in changes) {               const change = changes[propName];               const to  = JSON.stringify(change.currentValue);               const from = JSON.stringify(change.previousValue);               const changeLog = `${propName}: changed from ${from} to ${to} `;               this.changelog.push(changeLog);          }      }  } |

First, we imported the DoCheck from the @angular/core library

|  |  |
| --- | --- |
| 1  2  3  4 | import { Component, Input, OnChanges, OnInit, SimpleChanges, SimpleChange, DoCheck  } from '@angular/core';  import { Customer } from './customer'; |

Implement the DoCheck Interface

|  |  |
| --- | --- |
| 1  2  3 | export class ChildComponent implements OnChanges, DoCheck, OnInit { |

We have created a new property oldCustomer to store the old value of the customer. We also have DoCheckcount property, which keeps track of no of times this hook is fired

|  |  |
| --- | --- |
| 1  2  3  4 | oldCustomer: Customer= new Customer();      DocheckCount = 0; |

We are cloning the customer object into the oldCustomer in the OnInit hook. The old customer values are compared with new customer to check whether the customer object has changed

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ngOnInit() {          console.log('OnInit');          this.oldCustomer = Object.assign({}, this.customer);      } |

Finally, in the ngDoChek hook, we compare the new values of customer to oldCustomer values to detect any changes

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | ngDoCheck() {          console.log('Docheck');          this.DocheckCount++;          if (this.oldCustomer.name !== this.customer.name || this.oldCustomer.code !== this.customer.code ) {              const to  = JSON.stringify(this.customer);              const from = JSON.stringify(this.oldCustomer);              const changeLog = `DoCheck customer: changed from ${from} to ${to} `;              this.changelog.push(changeLog);                this.oldCustomer = Object.assign({}, this.customer);          }        } |

That’s it

Run the code and you will notice that whenever the customer is added, the our code detects the change logs it into our changeLog

When ngDoCheck is called

Notice that DoCheckCount keeps incrementing for every keystroke, mouse movements

Angular calls this hook very frequently. This hook is called after every change detection cycle no matter where the change has occurred

It is advisable to keep the implementation of Docheck simple and lightweight. Otherwise, it will result in bad user experience

Checking for changes

In the example above, we cloned our customer object and checked each property for a change. But what if we have large object or array.

The Angular provides a service called differs, which evaluate the given object/array and determines what changed

There are two types of differs, that angular provides

key-value differs

iterable differs

key-value differs

The KeyValueDiffers service is a differ that tracks changes made to an object over time and also expose an API to react to these changes.

Key-value differs should be used for dictionary-like structures, and it works at the key level. This differ will identify changes when a new key is added, when a key removed and when the value of a key changed.

Iterable differs

Iterable differs service is used when we have a list-like structure and we’re only interested in  
knowing things that were added or removed from that list.

It will detect if the elements are added/removed from the array. This will not detect if the changes are done to the elements of array

To do that, you need to create a separate key value differ for the each element

Example of key-value differs

Import KeyValueDiffers from @angular/core

Inject it into the constructor

|  |  |
| --- | --- |
| 1  2  3  4 | constructor(private differs: KeyValueDiffers) {     } |

Create a differ property for customer object

|  |  |
| --- | --- |
| 1  2  3 | differ: any; |

Initaisle the differ object with initial value.

The find() method searches for a key value differ in differs collection. If not found creates the differ and returns an instance of DefaultKeyValueDiffer

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ngOnInit() {          console.log('OnInit');          this.differ = this.differs.find(this.customer).create(null);      } |

Next, using the diff method of the differ, we are checking if our object is changed. The object returns null if there is no change. It returns an object, which contains the changes made to the object

|  |  |
| --- | --- |
| 1  2  3 | const customerChanges = this.differ.diff(this.customer); |

We, can then use the returned object to find out what was added, changed or removed properties using the forEachChangedItem, forEachAddedItem, forEachRemovedItem as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | if (customerChanges) {              console.log(customerChanges);              customerChanges.forEachChangedItem(r =>  this.changelog.push('changed ' + r.key + ' ' + JSON.stringify( r.currentValue)));              customerChanges.forEachAddedItem(r =>  this.changelog.push('added ' + r.key + ' ' + JSON.stringify( r.currentValue)));              customerChanges.forEachRemovedItem(r =>  this.changelog.push('removed ' + r.key + ' ' + JSON.stringify( r.currentValue)));          } |

The Complete child component as follows

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | import { Component, Input, OnChanges, OnInit, SimpleChanges, SimpleChange, DoCheck, KeyValueDiffers } from '@angular/core';  import { Customer } from './customer';    @Component({      selector: 'child-component',      template: `<h2>Child  Component</h2>                 <p>Message {{ message }} </p>                 <p>Customer Name {{ customer.name }} </p>                 <p>Customer Code {{ customer.code }} </p>                 <p>Do Check count {{ DocheckCount }} </p>                 <ul><li \*ngFor="let log of changelog;"> {{ log }}</li></ul> `  })  export class ChildComponent implements OnChanges, DoCheck, OnInit {      @Input() message: string;      @Input() customer: Customer;      changelog: string[] = [];      oldCustomer: Customer= new Customer();      DocheckCount = 0;      differ: any;        constructor(private differs: KeyValueDiffers) {        }      ngOnInit() {          console.log('OnInit');          this.differ = this.differs.find(this.customer).create(null);      }        ngDoCheck() {          console.log('Docheck');          this.DocheckCount++;            const customerChanges = this.differ.diff(this.customer);            if (customerChanges) {              console.log(customerChanges);              customerChanges.forEachChangedItem(r =>  this.changelog.push('changed ' + r.key + ' ' + JSON.stringify( r.currentValue)));              customerChanges.forEachAddedItem(r =>  this.changelog.push('added ' + r.key + ' ' + JSON.stringify( r.currentValue)));              customerChanges.forEachRemovedItem(r =>  this.changelog.push('removed ' + r.key + ' ' + JSON.stringify( r.currentValue)));          }        }        ngOnChanges(changes: SimpleChanges) {          console.log('OnChanges');          console.log(JSON.stringify(changes));            // tslint:disable-next-line:forin          for (const propName in changes) {               const change = changes[propName];               const to  = JSON.stringify(change.currentValue);               const from = JSON.stringify(change.previousValue);               const changeLog = `${propName}: changed from ${from} to ${to} `;               this.changelog.push(changeLog);          }      }  } |

Iterable differs

The iterable differ behaves the same way the key-value differ but it only provides methods for items that were added or removed.

The iterable differs works on arrays. Using iterable differs is no different for key value differs. Just import the IterableDiffers and inject it into the constructor. Rest of the code stays same (except forEachChangedItem)

Conclusion

In this tutorial, we looked at how to use ngDoCheck hook to built custom change detection for input properties. We also looked at how to use key-value differs and Iterable differs.

# Angular Forms

The data entry forms can be very simple to very complex. It can contain large no of input fields, Spanning multiple tabs. Forms may also contain complex validation logic interdependent on multiple fields.

Some things forms are expected to do

* Initialize the forms fields and present it to the user
* Capture the data from the user
* Track changes made to the fields
* Validate the inputs
* Display helpful errors to the user

Link to all article on Angular Forms

Angular Forms Module

Angular forms module provides all the above [services](https://www.tektutorialshub.com/angular/angular-services/) out of the box. It binds the form field to the [Angular component](https://www.tektutorialshub.com/angular/angular-component/) class. It tracks changes made to the form fields so that we can respond accordingly. The Angular forms provide the [built-in validators](https://www.tektutorialshub.com/angular/angular-reactive-forms-validation/) to validate the inputs. You can create your own [custom validator.](https://www.tektutorialshub.com/angular/custom-validator-in-angular-reactive-form/) It presents the validation errors to the user. Finally, it encapsulates all the input fields into an object structure when the user submits the form.

Angular takes two approaches to build the forms. One is [Template-driven forms](https://www.tektutorialshub.com/angular/angular-template-driven-forms/) approach and another one is [Reactive forms](https://www.tektutorialshub.com/angular/angular-reactive-forms/) or model-driven forms approach

Template-driven forms approach

In [Template-driven approach](https://www.tektutorialshub.com/angular/angular-template-driven-forms/) is the easiest way to build the Angular forms. The logic of the form is placed in the template. The approach here is similar to what we did in AngularJs.

Model-driven forms approach

In [Reactive Forms](https://www.tektutorialshub.com/angular/angular-reactive-forms/) or [Model-driven](https://www.tektutorialshub.com/angular/angular-reactive-forms/) approach, the logic of the form is defined in the component as an object. The Model-driven approach has more benefits as it makes the testing of the component easier.

In this approach, the representation of the form is created in the component class. This form model is then bound to the HTML elements. it is done using the special markups.

[Template-driven forms](https://www.tektutorialshub.com/angular/angular-template-driven-forms/) in Angular allows us to create sophisticated looking forms easily without writing any javascript code. The model-driven forms are created in component class, where Form fields are created as properties of our component class.  This makes it easier to test.

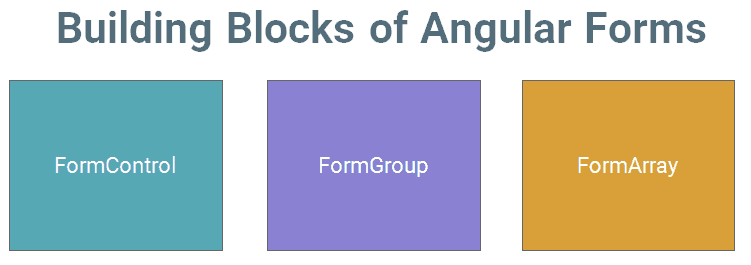
[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Here is the list of tutorials on how to build a form using the template-driven approach & reactive or model-driven approach

Ad by Valueimpression

Building Blocks of Angular Forms

The Angular Forms module consists of three Building blocks, irrespective of whether you are using [Template-driven](https://www.tektutorialshub.com/angular/angular-template-driven-forms/) or [Reactive forms](https://www.tektutorialshub.com/angular/angular-reactive-forms/) approach.



FormControl

A [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) represents a single input field in an Angular form.

Consider a simple Text input box

|  |  |
| --- | --- |
| 1  2  3 | First Name : <input type="text" name="firstname" /> |

As a developer, you would like to know the current value in the Text box. You would also be like to know if the value is valid or not. If the user has changed the value(dirty) or is it unchanged. You would like to be notified when the user changes value.

The [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) is an object that encapsulates all this information related to the single input element. It Tracks the value and validation status of each of these control

The [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) is just a class. A [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) is created for each form field. We can refer them in our component class and inspect its properties and methods

You can use [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) to set the value of the Form field, find the status of form field like (valid/invalid, pristine/dirty, touched/untouched ) etc & add validation rules to it.

The above input field is created using the [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) as shown below

|  |  |
| --- | --- |
| 1  2  3 | let firstname= new FormControl(); //Creating a FormControl in a Reactive forms |

Then, you can retrieve the current value in the input field using the value property

|  |  |
| --- | --- |
| 1  2  3 | firstname.value   //Returns the value of the first name field |

You can check the validation status of the First Name element as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | firstname.errors      // returns the list of errors  firstname.dirty       // true if the value has changed (dirty)  firstname.touched     // true if input field is touched  firstname.valid       // true if the input value has passed all the validation |

FormGroup

[FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) is a collection of [FormControls](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) . Each [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) is a property in a [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/). with the control name as the key.

Often forms have more than one field. It is helpful to have a simple way to manage the Form controls together.

Consider the following Form. we have three input fields street, city & Pincode.

|  |  |
| --- | --- |
| 1  2  3  4  5 | city : <input type="text" name="city" >  Street : <input type="text" name="street" >  PinCode : <input type="text" name="pincode" > |

All of the above input fields are represented as the separate [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/). If we wanted to check the validity of our form, we have to check the validity of each and every [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) for validity. Imagine Form having large no of fields. It is cumbersome to loop over large no of [FormControls](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) and check for validity.

[FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) solve’s this issue by providing a wrapper interface around a collection of [FormControls](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) A [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) tracks the status of each child [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) and aggregates the values into one object. with each control name as the key

We can group these input fields under the group address as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | let address= new FormGroup({      street : new FormControl(""),      city : new FormControl(""),      pinCode : new FormControl("")  }) |

In the above example, the address is our FormGroup, consisting of 3 Form Controls city, street, and Pincode. Now we can check the validity of the entire group together. For example, if the state is invalid, then the address [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) returns the invalid state.

You can read the value of an address using the value method, which returns the JSON object as shown below

|  |  |
| --- | --- |
| 1  2  3 | address.value |

The Return value

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | address {      street :"",      city:"",      Pincode:""  } |

You can access child control as

|  |  |
| --- | --- |
| 1  2  3 | address.get("street") |

Check the Validation status as follows

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | address.errors     // returns the list of errors  address.dirty      // true if the value of one of the child control has changed (dirty)  address.touched    // true if one of the child control is touched  address.valid      // true if all the child controls passed the validation |

A typical Angular Form can have more than one [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/). A [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) can also contain another [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/).

The Angular form is itself a [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/)

FormArray

[FormArray](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/) is an array of form controls. It is similar to [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) except for one difference. In [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) each [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) is a property with the control name as the key. In [FormArray](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/)is an array of form controls.

We define the [FormArray](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/)as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | contactForm = new FormGroup( {      name: new FormControl(''),      cities:new FormArray([        new FormControl('Mumbai'),        new FormControl('Delhi')      ])    }); |

You can get the reference to the cities from the contactForm.get method

|  |  |
| --- | --- |
| 1  2  3  4  5 | cities() :FormArray {      return this.contactForm.get("cities") as FormArray    } |

Check the Validation status as follows

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | cities.errors     // returns the list of errors  cities.dirty      // true if the value of one of the child control has changed (dirty)  cities.touched    // true if one of the child control is touched  cities.valid      // true if all the child controls passed the validation |

Summary

In this tutorial, we learned what is Angular Forms all about. We looked at the basic building blocks of Angular Forms i.e. [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/), [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) & [FormArray](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/). The Angular allows us to build Forms using two different approaches. One is [Template Driven](https://www.tektutorialshub.com/angular/angular-template-driven-forms/) & another one is Reactive Forms or Model-driven. In the next few tutorials, we look at how to build Angular Forms using both of these options.

What is Template-driven form?

In Template Driven Forms we specify behaviors/validations using directives and attributes in our template and let it work behind the scenes. All things happen in Templates hence very little code is required in the component class. This is different from the reactive forms, where we define the logic and controls in the component class.

The Template-driven forms

The form is set up using ngForm directive

controls are set up using the ngModel directive

ngModel also provides the two-way data binding

The Validations are configured in the template via directives

Template-driven forms are

Contains little code in the component class

Easier to set up

While they are

Difficult to add controls dynamically

Unit testing is a challenge

Create the Example Application

Use ng new to create a new application

|  |  |
| --- | --- |
| 1  2  3 | ng new tdf  --routing=true --style=css |

Run ng serve and verify if everything is installed correctly.

Import FormsModule

To work with Template-driven forms, we must import the FormsModule. We usually import it in root module or in a [shared module](https://www.tektutorialshub.com/angular/angular-folder-structure-best-practices/#shared-module). The FormsModule contains all the form directives and constructs for working with forms

Open the app.module.ts and add the import { FormsModule } from '@angular/forms'; to it.

And also add the FormsModule to the imports metadata property array

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule } from '@angular/forms';        //import FormsModule    import { AppRoutingModule } from './app-routing.module';  import { AppComponent } from './app.component';    @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule,      AppRoutingModule,      FormsModule                    //Add in Imports Array    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

HTML Form

The first task is to build the template. The following is a regular HTML form. We enclose it in a <form> tag. We have included two text input (FirstName & LastName), a email (email), a radio button (gender), a checkbox (isMarried), and a select list (country). These are form elements.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | <form>      <p>      <label for="firstname">First Name</label>      <input type="text" id="firstname" name="firstname">    </p>      <p>      <label for="lastname">Last Name</label>      <input type="text" id="lastname" name="lastname">    </p>      <p>      <label for="email">Email </label>      <input type="text" id="email" name="email">    </p>      <p>      <label for="gender">Geneder</label>      <input type="radio" value="male" id="gender" name="gender"> Male      <input type="radio" value="female" id="gender" name="gender"> Female    </p>      <p>      <label for="isMarried">Married</label>      <input type="checkbox" id="isMarried" name="isMarried">    </p>      <p>    <label for="country">country </label>    <select name="country" id="country">      <option selected="" value=""></option>      <option [ngValue]="c.id" \*ngFor="let c of countryList">        {{c.name}}      </option>    </select>    </p>      <p>      <button type="submit">Submit</button>    </p>    </form> |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

**Component Class**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'Template driven forms';      countryList:country[] = [      new country("1", "India"),      new country('2', 'USA'),      new country('3', 'England')    ];  }    export class country {    id:string;    name:string;      constructor(id:string, name:string) {      this.id=id;      this.name=name;    }  } |

ngForm

Once, we have a form with few form elements, the angular automatically converts it into a Template-driven form. This is done by the ngForm directive.

The ngForm directive is what makes the Angular template-driven forms work. But we do not need to add it explicitly. Angular adds it automatically

When we include FormsModule, the Angular is going to look out for any <form> tag in our HTML template. Angular does this via ngForm [directive](https://www.tektutorialshub.com/angular/angular-directives/). ngForm directive automatically detects the <form> tag and automatically binds to it. You do not have to do anything on your part to invoke and bind the ngForm directive.

The ngForm does the following

Binds itself to the <Form> directive

Creates a top-level FormGroup instance

CreatesFormControl instance for each of child control, which has ngModel directive.

CreatesFormGroup instance for each of the  NgModelGroup directive.

We can export the ngForm instance into a local template variable using ngForm as the key (ex: #contactForm="ngForm"). This allows us to access the many properties and methods of ngForm using the template variable contactForm

Hence, update the form element as shown below.

|  |  |
| --- | --- |
| 1  2  3 | <form #contactForm="ngForm"> |

FormControl

The FormControl is the basic building block of the [Angular Forms](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/). It represents a single input field in an [Angular form](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/)**.** The [Angular Forms Module](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/#angular-forms-module) binds the input element to a FormControl. We use the FormControl instance to track the value, user interaction and validation status of an individual form element. Each individual Form element is a FormControl

We have six form elements in our HTML template. They are firstName, lastname, email, gender, isMarried & country. We need to bind them to FormControl instance. We do this by using the ngModel directive. Add the ngModel directive to each control as shown below.

|  |  |
| --- | --- |
| 1  2  3 | <input type="text" name="firstname" ngModel> |

ngModel will use the name attribute to create the FormControl instance for each of the Form field it is attached.

Submit Form

Now have the template ready, except for the final piece i.e submitting data to the component.

We use the ngSubmit event, to submit the form data to the component class. We use the [event binding](https://www.tektutorialshub.com/angular/angular-data-binding/) (parentheses) to bind ngSubmit to OnSubmit method in the component class. When the user clicks on the submit button, the ngSubmit event will fire

|  |  |
| --- | --- |
| 1  2  3 | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)"> |

We are passing the local template variable contactForm in onSubmit method. contactForm holds the reference to the ngForm directive. We can use this in our component class to extract the data from the form fields.

Final Template

Our final template is as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)">      <p>      <label for="firstname">First Name</label>      <input type="text" name="firstname" ngModel>    </p>      <p>      <label for="lastname">Last Name</label>      <input type="text" name="lastname" ngModel>    </p>      <p>      <label for="email">Email </label>      <input type="text" id="email" name="email" ngModel>    </p>      <p>      <label for="gender">Geneder</label>      <input type="radio" value="male" name="gender" ngModel> Male      <input type="radio" value="female" name="gender" ngModel> Female    </p>      <p>      <label for="isMarried">Married</label>      <input type="checkbox" name="isMarried" ngModel>    </p>      <select name="country" ngModel>      <option [ngValue]="c.id" \*ngFor="let c of countryList">        {{c.name}}      </option>    </select>      <p>      <button type="submit">Submit</button>    </p>    </form> |

Receive Form Data

We need to receive the data in component class from our form. To do this we need to create the onSubmit method in our component class. The submit method receives the reference to the ngForm directive, which we named is as contactForm. The contactForm exposes the value method which returns the form fields as a Json object.

|  |  |
| --- | --- |
| 1  2  3  4  5 | onSubmit(contactForm) {      console.log(contactForm.value);    } |

You can print the value to the console using the console.log(contactForm.value)

Run the code now and enter some data into the form. Open the Developer Console in your browser and check the output, when you submit the data.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | country: "1"  firstname: "Sachin"  email:"sachin@gmail.com"  gender: "male"  isMarried: true  lastname: "Tendulkar" |

Angular template-driven forms in Action

Local Variable

We can assign the ngForm,FormControl or FormGroup instance to a template local variable. This allows us to check the status of the form like whether the form is valid, submitted, and value of the form elements, etc

ngForm

We have access to the ngForm instance via the local template variable #contactForm.

|  |  |
| --- | --- |
| 1  2  3 | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)"> |

Now, we can make use of some of the properties & methods to know the status of form. For Example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <p>    <button type="submit">Submit</button>  </p>    <pre>Value : {{contactForm.value | json }} </pre>  <pre>Valid : {{contactForm.valid}} </pre>  <pre>Touched : {{contactForm.touched  }} </pre>  <pre>Submitted : {{contactForm.submitted  }} </pre> |

value: The value property returns the object containing the value of every FormControl  
valid: Returns true if the form is Valid else returns false.  
touched: True if the user has entered a value in at least in one field.  
submitted: Returns true if the form is submitted. else false.

FormControl

Similarly, we can also get access to the FormControl instance by assigning the ngModel to a local variable as shown below

|  |  |
| --- | --- |
| 1  2  3 | <input type="text" name="firstname" #fname="ngModel" ngModel> |

Now, the variable #fname holds the reference to the firstname FormControl. We can then access the properties of FormControl like value, valid, isvalid, tocuhed etc

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <p>    <label for="firstname">First Name </label>    <input type="text" name="firstname" #fname="ngModel" ngModel>  </p>    <pre>Value    : {{fname.value}} </pre>  <pre>valid    : {{fname.valid}} </pre>  <pre>invalid  : {{fname.invalid}} </pre>  <pre>touched  : {{fname.touched}} </pre |

value: Returns the current value of the control  
valid: Returns true if the value is Valid else false  
invalid: True if the value is invalid else false  
touched: Returns true if the value is entered in the element

Nested FormGroup

The FormGroup is a collection of FormControl. It can also contain other FormGroup's.

The ngForm directive creates the top Level FormGroup behind the scene, when we use the <Form> directive.

|  |  |
| --- | --- |
| 1  2  3 | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)"> |

We can add new [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) using the ngModelGroup directive. Let us add street, city & Pincode form controls and group them under the address FormGroup

All you need to do is to enclose the fields inside a div element with ngModelGroup directive applied on it as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | <div ngModelGroup="address">        <p>        <label for="city">City</label>        <input type="text" name="city" ngModel>      </p>        <p>        <label for="street">Street</label>        <input type="text" name="street" ngModel>      </p>      <p>        <label for="pincode">Pin Code</label>        <input type="text" name="pincode" ngModel>      </p>    </div> |

Run the App and submit. The resultant object is as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | Value : {    "firstname": "Sachin",    "lastname": "Tendulkar",    "email":"sachin@gmail.com"    "gender": "male",    "isMarried": true,    "country": "1",    "address": {      "city": "Mumbai",      "street": "Fashin Street",      "pincode": "400600"    }  } |

Setting the Initial Value

The form is usually pre-filled with some default data. In the case of editing, we have to show the user the current data. You can refer to the next tutorial on [How to set value in the template-driven form](https://www.tektutorialshub.com/angular/how-to-set-value-in-template-driven-forms-in-angular/).

Validating the Form

Validating the form is another important task. We have covered it in Validation in template-driven form tutorial.

Summary

**Angular Template-driven Forms** is simpler compared to the reactive forms. The FormsModule is imported first. Then we create the HTML form. The Angular detects the <form> tag and converts the form to the Angular Form. ngModel directive added to each form element, which converts them to FormControl. Finally, submit event is subscribed via event binding.

Template

The following is the app.component.html from the [angular template-driven forms tutorial](https://www.tektutorialshub.com/angular/angular-template-driven-forms/).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)">      <p>      <label for="firstname">First Name </label>      <input type="text" id="firstname" name="firstname" ngModel>    </p>      <p>      <label for="lastname">Last Name </label>      <input type="text" id="lastname" name="lastname" ngModel>    </p>      <p>      <label for="email">Email </label>      <input type="text" id="email" name="email"  ngModel>    </p>      <p>      <label for="gender">Geneder </label>      <input type="radio" value="male" id="gender" name="gender" ngModel> Male      <input type="radio" value="female" id="gender" name="gender" ngModel> Female    </p>      <p>      <label for="isMarried">Married </label>      <input type="checkbox" id="isMarried" name="isMarried" ngModel>    </p>      <p>      <label for="country">country </label>      <select id="country" name="country" ngModel>        <option [ngValue]="c.id" \*ngFor="let c of countryList">          {{c.name}}        </option>      </select>    </p>      <div ngModelGroup="address">        <p>        <label for="city">City</label>        <input type="text" id="city" name="city" ngModel>      </p>        <p>        <label for="street">Street</label>        <input type="text" id="street" name="street" ngModel>      </p>      <p>        <label for="pincode">Pin Code</label>        <input type="text" id="pincode" name="pincode" ngModel>      </p>      </div>      <p>      <button type="submit">Submit</button>    </p>    </form> |

Before we set the default value, it is better to create a model class for the above form. Open the app.component.ts and add the following class

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | export class contact {    firstname:string;    lastname:string;    email:string;    gender:string;    isMarried:boolean;    country:string;    address: {      city:string;      street:string;      pincode:string;    }  } |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Set value in template-driven forms

There are two ways you can set the value of the form elements

Two-way data binding

Use the template reference variable

Two-way data binding

The [two-way data binding](https://www.tektutorialshub.com/angular/angular-data-binding/#two-way-binding).is the recommended way to set the value in the template-driven forms.

The following code uses the [(ngModel)]="contact.firstname" to bind the firstname HTML element to the contact.firstname field in the component class. The advantageous here is that any changes made in the form are automatically propagated to the component class and changes made in component class are immediately shown in the form.

|  |  |
| --- | --- |
| 1  2  3  4 | <label for="firstname">First Name </label>  <input type="text" id="firstname" name="firstname" [(ngModel)]="contact.firstname"> |

Set the default/initial value

To set the initial or default value all you need to populate the contact model in the [ngOnInit](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/) method as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | ngOnInit() {        this.contact = {        firstname: "Sachin",        lastname: "Tendulkar",        email: "sachin@gmail.com",        gender: "male",        isMarried: true,        country: "2",        address: { city: "Mumbai", street: "Perry Cross Rd", pincode: "400050" }      };      } |

Set the value individually or dynamically

|  |  |
| --- | --- |
| 1  2  3  4  5 | changeCountry() {    this.contact.country = "1";  } |

Reset form

|  |  |
| --- | --- |
| 1  2  3 | <button type="button" (click)="reset(contactForm)">Reset</button> |

|  |  |
| --- | --- |
| 1  2  3  4  5 | reset(contactForm :NgForm) {    contactForm.resetForm();  } |

app.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86 | import { Component, ViewChild, ElementRef, OnInit } from '@angular/core';  import { NgForm } from '@angular/forms';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {    title = 'Template driven forms';        countryList: country[] = [      new country("1", "India"),      new country('2', 'USA'),      new country('3', 'England')    ];      contact: contact;      ngOnInit() {        this.contact = {        firstname: "Sachin",        lastname: "Tendulkar",        email: "sachin@gmail.com",        gender: "male",        isMarried: true,        country: "2",        address: { city: "Mumbai", street: "Perry Cross Rd", pincode: "400050" }      };      }      onSubmit() {      console.log(this.contact);    }      setDefaults() {      this.contact = {        firstname: "Sachin",        lastname: "Tendulkar",        email: "sachin@gmail.com",        gender: "male",        isMarried: true,        country: "2",        address: { city: "Mumbai", street: "Perry Cross Rd", pincode: "400050" }      };    }      changeCountry() {      this.contact.country = "1";    }      reset(contactForm :NgForm) {      contactForm.resetForm();    }    }    export class contact {    firstname: string;    lastname: string;    email: string;    gender: string;    isMarried: boolean;    country: string;    address: {      city: string;      street: string;      pincode: string;    }  }      export class country {    id: string;    name: string;      constructor(id: string, name: string) {      this.id = id;      this.name = name;    }  } |

app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)">      <p>      <label for="firstname">First Name </label>      <input type="text" id="firstname" name="firstname" [(ngModel)]="contact.firstname">    </p>      <p>      <label for="lastname">Last Name </label>      <input type="text" id="lastname" name="lastname" [(ngModel)]="contact.lastname">    </p>      <p>      <label for="email">Email </label>      <input type="text" id="email" name="email"  [(ngModel)]="contact.email">    </p>      <p>      <label for="gender">Geneder </label>      <input type="radio" value="male" id="gender" name="gender" [(ngModel)]="contact.gender"> Male      <input type="radio" value="female" id="gender" name="gender" [(ngModel)]="contact.gender"> Female      </p>      <p>      <label for="isMarried">Married </label>      <input type="checkbox" id="isMarried" name="isMarried" [(ngModel)]="contact.isMarried">    </p>      <p>      <label for="country">country </label>      <select id="country" name="country" [(ngModel)]="contact.country">        <option [ngValue]="c.id" \*ngFor="let c of countryList">          {{c.name}}        </option>      </select>    </p>      <div ngModelGroup="address">        <p>        <label for="city">City</label>        <input type="text" id="city" name="city" [(ngModel)]="contact.address.city">      </p>        <p>        <label for="street">Street</label>        <input type="text" id="street" name="street" [(ngModel)]="contact.address.street">      </p>        <p>        <label for="pincode">Pin Code</label>        <input type="text" id="pincode" name="pincode"  [(ngModel)]="contact.address.pincode">      </p>      </div>      <p>      <button type="submit">Submit</button>    </p>      <p>      <button type="button" (click)="changeCountry()">Change Country</button>      <button type="button" (click)="setDefaults()">Set Defaults</button>      <button type="button" (click)="reset(contactForm)">Reset</button>    </p>      <b>valid</b> {{contactForm.valid}}    <b>touched</b> {{contactForm.touched}}    <b>pristine</b> {{contactForm.pristine}}    <b>dirty</b> {{contactForm.dirty}}    </form> |

Template reference variable

We have a #contactForm reference variable, which is an instance of ngForm.

|  |  |
| --- | --- |
| 1  2  3 | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)"> |

We can get the reference to the #contactForm in the app.component.ts, using the viewchild

|  |  |
| --- | --- |
| 1  2  3 | @ViewChild('contactForm',null) contactForm: NgForm; |

Once we have the reference, we can use the [setValue](https://www.tektutorialshub.com/angular/setvalue-patchvalue-in-angular/) method of the ngForm to set the initial value

Set the default or initial value

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | ngOnInit() {       this.contact = {        firstname: "Sachin",        lastname: "Tendulkar",        email: "sachin@gmail.com",        gender: "male",        isMarried: true,        country: "2",        address: {          city: "Mumbai",          street: "Perry Cross Rd",          pincode: "400050"        }      };        setTimeout(() => {        this.contactForm.setValue(this.contact);      });      } |

Note that we are using the setTimeout That is because the form controls are yet initialized when the OnInit is fired. We will get the following error message

There are no form controls registered with this group yet. If you’re using ngModel, you may want to check next tick (e.g. use setTimeout).

Set the value individually or dynamically

You can also set the value individually using the setValue method of the individual FormControl.

You will get the reference to the individual FormControl from the controls collection of the ngForm. Once you get the reference use the setValue on the FormControl instance to change the value.

For Example, this code will change the country to India

|  |  |
| --- | --- |
| 1  2  3  4  5 | changeCountry() {     this.contactForm.controls["country"].setValue("1");  } |

Call the changeCountry method from the Template.

|  |  |
| --- | --- |
| 1  2  3 | <button type="button" (click)="changeCountry()">Change Country</button> |

Reset values

You can reset the form to empty value using the reset or resetForm method of the ngForm. These also resets the form status like dirty, valid, pristine & touched, etc

|  |  |
| --- | --- |
| 1  2  3  4  5 | reset() {    this.contactForm.reset();  } |

|  |  |
| --- | --- |
| 1  2  3  4  5 | resetForm() {     this.contactForm.resetForm();  } |

Set Default Value

You can invoke the setValue anytime to set the form back to the default value. This will set the entire form to the value held by the contact form.

|  |  |
| --- | --- |
| 1  2  3  4  5 | setDefaults() {      this.contactForm.setValue(this.contact);    } |

patch value

You can make use of the [patchValue](https://www.tektutorialshub.com/angular/setvalue-patchvalue-in-angular/) to change the only few fields anytime. The control property of the ngForm returns the reference to the top level FormGroup. Then, you can make use of the patchValue method to change only firstname, lastname & email fields

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | patchValue() {      let obj = {        firstname: "Rahul",        lastname: "Dravid",        email: "rahul@gmail.com",      };        this.contactForm.control.patchValue(obj);      } |

Set value of nested FormGroup

You can update nested [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) by getting a reference to the nested FormGroup from the controls collection of ngForm.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | changeAddress() {      let obj = {        city: "Bangalore",        street: "Brigade Road",        pincode: "600100"      };      let address= this.contactForm.controls["address"] as FormGroup      address.patchValue(obj);      } |

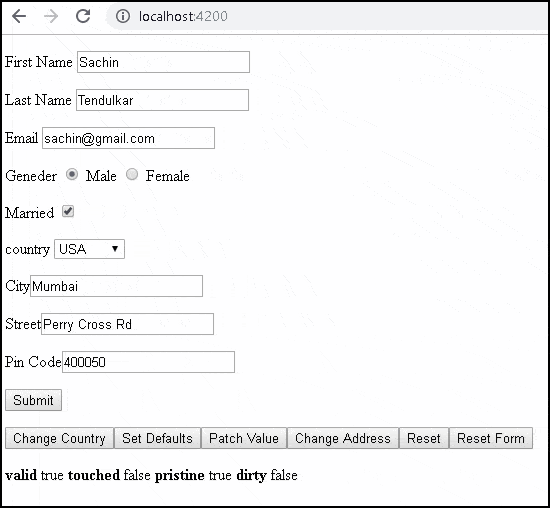
**The complete code.**

app.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114 | import { Component, ViewChild, ElementRef, OnInit } from '@angular/core';  import { NgForm, FormGroup } from '@angular/forms';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {    title = 'Template driven forms';      @ViewChild('contactForm', null) contactForm: NgForm;      countryList: country[] = [      new country("1", "India"),      new country('2', 'USA'),      new country('3', 'England')    ];      contact: contact;      ngOnInit() {        this.contact = {        firstname: "Sachin",        lastname: "Tendulkar",        email: "sachin@gmail.com",        gender: "male",        isMarried: true,        country: "2",        address: {          city: "Mumbai",          street: "Perry Cross Rd",          pincode: "400050"        }      };        setTimeout(() => {        this.contactForm.setValue(this.contact);      });      }      onSubmit() {      console.log(this.contactForm.value);    }      setDefaults() {      this.contactForm.setValue(this.contact);    }      changeCountry() {      this.contactForm.controls["country"].setValue("1");    }      patchValue() {      let obj = {        firstname: "Rahul",        lastname: "Dravid",        email: "rahul@gmail.com",      };        this.contactForm.control.patchValue(obj);      }      changeAddress() {      let obj = {        city: "Bangalore",        street: "Brigade Road",        pincode: "600100"      };      let address= this.contactForm.controls["address"] as FormGroup      address.patchValue(obj);      }      reset() {      this.contactForm.reset();    }      resetForm() {      this.contactForm.resetForm();    }  }      export class contact {    firstname: string;    lastname: string;    email: string;    gender: string;    isMarried: boolean;    country: string;    address: {      city: string;      street: string;      pincode: string;    }  }      export class country {    id: string;    name: string;      constructor(id: string, name: string) {      this.id = id;      this.name = name;    }  } |

app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80 | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)">      <p>      <label for="firstname">First Name </label>      <input type="text" id="firstname" name="firstname" ngModel>    </p>      <p>      <label for="lastname">Last Name </label>      <input type="text" id="lastname" name="lastname" ngModel>    </p>      <p>      <label for="email">Email </label>      <input type="text" id="email" name="email"  ngModel>    </p>      <p>      <label for="gender">Geneder </label>      <input type="radio" value="male" id="gender" name="gender" ngModel> Male      <input type="radio" value="female" id="gender" name="gender" ngModel> Female    </p>      <p>      <label for="isMarried">Married </label>      <input type="checkbox" id="isMarried" name="isMarried" ngModel>    </p>      <p>      <label for="country">country </label>        <select id="country" name="country" ngModel>        <option [ngValue]="c.id" \*ngFor="let c of countryList">          {{c.name}}        </option>      </select>      </p>      <div ngModelGroup="address">        <p>        <label for="city">City</label>        <input type="text" id="city" name="city" ngModel>      </p>        <p>        <label for="street">Street</label>        <input type="text" id="street" name="street" ngModel>      </p>      <p>        <label for="pincode">Pin Code</label>        <input type="text" id="pincode" name="pincode"  ngModel>      </p>      </div>      <p>      <button type="submit">Submit</button>    </p>      <p>      <button type="button" (click)="changeCountry()">Change Country</button>      <button type="button" (click)="setDefaults()">Set Defaults</button>      <button type="button" (click)="patchValue()">Patch Value</button>      <button type="button" (click)="changeAddress()">Change Address</button>      <button type="button" (click)="reset()">Reset</button>      <button type="button" (click)="resetForm()">Reset Form</button>    </p>      <b>valid</b> {{contactForm.valid}}    <b>touched</b> {{contactForm.touched}}    <b>pristine</b> {{contactForm.pristine}}    <b>dirty</b> {{contactForm.dirty}}    </form> |



Summary

In this tutorial, we learned how to set the form values in the template-driven forms. We can either use the setValue of the ngForm directive or use the [two-way data binding](https://www.tektutorialshub.com/angular/angular-data-binding/#two-way-binding).

What are Reactive Forms?

Reactive forms are forms where we define the structure of the form in the component class. I,e we create the form model with Form Groups, Form Controls, and Form Arrays. We also define the validation rules in the component class. Then, we bind it to the HTML form in the template. This is different from the template-driven forms, where we define the logic and controls in the HTML template.

How to use Reactive Forms

Import ReactiveFormsModule

Create Form Model in component class using Form Group, Form Control & Form Arrays

Create the HTML Form resembling the Form Model.

Bind the HTML Form to the Form Model

Reactive Forms Example Application

Use ng new to create a new application

|  |  |
| --- | --- |
| 1  2  3 | ng new mdf  --routing=true --style=css |

Run ng serve and verify if everything is installed correctly.

Import ReactiveFormsModule

To work with Reactive forms, we must import the ReactiveFormsModule. We usually import it in root module or in a shared module. The ReactiveFormsModule contains all the form directives and constructs for working with angular reactive forms.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { ReactiveFormsModule } from '@angular/forms';    import { AppRoutingModule } from './app-routing.module';  import { AppComponent } from './app.component';    @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule,      AppRoutingModule,      ReactiveFormsModule    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

Model

In the template-driven approach, we used ngModel & ngModelGroup directive on the HTML elements. The FormsModule created the [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) & [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/)instances from the template. This happens behind the scene.

[In Reactive](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/) Forms approach, It is our responsibility to build the Model using FormGroup, FormControl and FormArray.

The FormGroup, FormControl & [FormArray](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/) are the three building blocks of the Angular Forms. We learned about them in [Angular Forms Tutorial](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/).

FormControl encapsulates the state of a **single form element** in our form. It stores the value and state of the form element and helps us to interact with them using properties & methods.

FormGroup represents a collection of form Controls. It can also contain form groups and form arrays. In fact, an angular form is a FormGroup.

Let’s create the model for our Form.

First, we need to import FormGroup, FormControl & Validator from the @angular/forms. Open the app.component.ts and the add following import statement.

|  |  |
| --- | --- |
| 1  2  3 | import { FormGroup, FormControl, Validators } from '@angular/forms' |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

FormGroup

The FormGroup is created with the following syntax

|  |  |
| --- | --- |
| 1  2  3 | contactForm = new FormGroup({}) |

The FormGroup takes 3 arguments. a collection of a child FormControl, a validator, and an asynchronous validator. The validators are optional.

FormControl

The first argument to FormGroup is the collection of FormControl. They are added using the FormControl method as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | contactForm = new FormGroup({    firstname: new FormControl(),    lastname: new FormControl(),    email: new FormControl(),    gender: new FormControl(),    isMarried: new FormControl(),    country: new FormControl()  }) |

In the above, we have created an instance of a FormGroup and named it as contactForm. contactForm is our top-level FormGroup. Under the contactForm, we have five FormControl instances each representing the properties firstname. lastname.email, gender, ismarried & country.

The Other two arguments to FormGroup are Sync Validator & Async Validator. They are optional.

HTML Form

The next task is to build an HTML form. The following is a regular HTML form. We enclose it in a <form> tag. We have included two text input (FirstName & LastName), an email field, a radio button (gender), a checkbox (isMarried), and a select list (country). These are Form elements.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | <form">      <p>      <label for="firstname">First Name </label>      <input type="text" id="firstname" name="firstname">    </p>      <p>      <label for="lastname">Last Name </label>      <input type="text" id="lastname" name="lastname">    </p>      <p>      <label for="email">Email </label>      <input type="text" id="email" name="email">    </p>      <p>      <label for="gender">Geneder </label>      <input type="radio" value="male" id="gender" name="gender"> Male      <input type="radio" value="female" id="gender" name="gender"> Female    </p>      <p>      <label for="isMarried">Married </label>      <input type="checkbox" id="isMarried" name="isMarried">    </p>      <p>      <label for="country">country </label>      <select id="country" name="country">        <option [ngValue]="c.id" \*ngFor="let c of countryList">          {{c.name}}        </option>      </select>    </p>      <p>      <button type="submit">Submit</button>    </p>    </form> |

Binding the template to the model

Now we need to associate our model to the Template. We need to tell angular that we have a model for the form.

This is done using the formGroup directive as shown below.

|  |  |
| --- | --- |
| 1  2  3 | <form [formGroup]="contactForm"> |

We have used the square bracket ([one-way binding](https://www.tektutorialshub.com/angular/angular-data-binding/)) around FormGroup directive and set that equal the model.

Next, we need to bind form fields to the FormControl models. We use the FormControlName directive for this. We add this directive to every form field element in our form. The value is set to the name of the corresponding FormControl instance in the component class.

|  |  |
| --- | --- |
| 1  2  3  4 | <input type="text" id="firstname" name="firstname" formControlName="firstname">  <input type="text" id="lastname" name="lastname" formControlName="lastname"> |

Submit form

We submit the form data to the component using the [Angular directive](https://www.tektutorialshub.com/angular/angular-directives/) named ngSubmit.  Note that we already have a submit button in our form. The ngSubmit directive binds itself to the click event of the submit button. We are using [event binding](https://www.tektutorialshub.com/angular/angular-data-binding/) (parentheses) to bind ngSubmit to OnSubmit method. When the user clicks on the submit button ngSubmit invokes the OnSubmit method on the Component class

|  |  |
| --- | --- |
| 1  2  3 | <form [formGroup]="contactForm" (ngSubmit)="onSubmit()"> |

Final Template

Our Final Template is as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | <form [formGroup]="contactForm" (ngSubmit)="onSubmit()">      <p>      <label for="firstname">First Name </label>      <input type="text" id="firstname" name="firstname" formControlName="firstname">    </p>      <p>      <label for="lastname">Last Name </label>      <input type="text" id="lastname" name="lastname" formControlName="lastname">    </p>      <p>      <label for="email">Email </label>      <input type="text" id="email" name="email" formControlName="email">    </p>      <p>      <label for="gender">Geneder </label>      <input type="radio" value="male" id="gender" name="gender" formControlName="gender"> Male      <input type="radio" value="female" id="gender" name="gender" formControlName="gender"> Female    </p>      <p>      <label for="isMarried">Married </label>      <input type="checkbox" id="isMarried" name="isMarried" formControlName="isMarried">    </p>      <p>      <label for="country">country </label>      <select id="country" name="country"  formControlName="country">        <option [ngValue]="c.id" \*ngFor="let c of countryList">          {{c.name}}        </option>      </select>    </p>      <p>      <button type="submit">Submit</button>    </p>    </form> |

Receive the data in the Component class

The last step is to receive the form data in the component class. All we need to do is to create the onSubmit method in our component class.

|  |  |
| --- | --- |
| 1  2  3  4  5 | onSubmit() {    console.log(this.contactForm.value);  } |

We are using the console.log(this.contactForm.value) to send the value of our form data to the console window.

Test the form

Now you can run the app and see the result. Open the developer console and see the value returned by the contactForm.value. The values of the form are returned as JSON object as shown below, which you can pass it your backend API to persist the information to the database.

FormControl

A FormControl takes 3 arguments. a default value, a validator and an asynchronous validator. All of them are optional.

Default Value

You can pass a default value as either as a string or as an object of key-value pair. When you pass object you can set both the value and the whether or not the control is disabled.

|  |  |
| --- | --- |
| 1  2  3  4 | //Setting Default value as string  firstname= new FormControl(‘Sachin’); |

|  |  |
| --- | --- |
| 1  2  3  4 | //Setting Default value & disabled state as object  firstname: new FormControl({value: ‘Rahul’, disabled: true}), |

Sync Validator

The second parameter is an array of sync Validators. Angular has some built-in Validators such as required and minLength etc.

You can pass with Validator function as shown below.

|  |  |
| --- | --- |
| 1  2  3 | firstname: new FormControl('', [Validators.required,Validators.minLength(10)]), |

Asynchronous validator

The third argument is the Async Validator. The syntax of Async Validators is similar to Sync Validators.

More on validation in our next tutorial Validations in Reactive forms.

Grouping the controls using FormGroup

We can group various form controls together. For Example fields such as street, city, and Pincode each will have their own form control, but can be grouped together as an address form group

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | contactForm = new FormGroup({    firstname: new FormControl(),    lastname: new FormControl(),    email: new FormControl(),    gender: new FormControl(),    isMarried: new FormControl(),    country: new FormControl(),    address:new FormGroup({      city: new FormControl(),      street: new FormControl(),      pincode:new FormControl()    })  }) |

In the code above, we have created new FormGroup Address and added three form controls i.e city, street & Pincode

In the template use the formGroupName directive to enclose the control using a div element as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | <div formGroupName="address">        <div class="form-group">          <label for="city">City</label>          <input type="text" class="form-control" name="city" formControlName="city" >      </div>        <div class="form-group">          <label for="street">Street</label>          <input type="text" class="form-control" name="street" formControlName="street" >      </div>        <div class="form-group">          <label for="pincode">Pin Code</label>          <input type="text" class="form-control" name="pincode" formControlName="pincode">      </div>      </div> |

Summary

We learned how to build Angular Reactive Forms in this tutorial. In the next tutorial, we will add validation rules to our application.

What is FormBuilder

The FormBuilder is the helper API to build forms in Angular.  It provides shortcuts to create the instance of the FormControl, [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) or FormArray. It reduces the code required to write the complex forms.

How to use FormBuilder

Import & inject FormBuilder API

To use the FormBuilder, first, we need to import it in our component

|  |  |
| --- | --- |
| 1  2  3 | import { FormBuilder } from '@angular/forms' |

Next, we need to inject it into our component class

|  |  |
| --- | --- |
| 1  2  3  4 | constructor(private formBuilder: FormBuilder) {  } |

Finally, use the group, array & control methods to build the FormModel

FormGroup

We use the group method to build the Form Group. We pass the list of [Form Controls](https://www.tektutorialshub.com/angular/formcontrol-in-angular/), [Form Array](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/), or another [Form Group](https://www.tektutorialshub.com/angular/formgroup-in-angular/) to the group method as key-value pair. Where the key is the name of the FormControl, FormGroup or FormArray. The value is the configuration of the control.

In the following example, we have added six form controls. The First argument to the FormControl is the initial value, which we set to empty string.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | this.contactForm = this.formBuilder.group({    firstname: [''],    lastname: [''],    email: [''],    gender: [''],    isMarried: [''],    country: [''],  }); |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Nested FormGroup

Creating a Nested FormGroup is just as easy. use the formbuilder.group method, as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | this.contactForm = this.formBuilder.group({    firstname: [''],    lastname: [''],    email: [''],    gender: [''],    isMarried: [''],    country: [''],    address: this.formBuilder.group({      city: [''],      street: [''],      pincode: ['']    })  }) |

Validations

The second argument to the FormControl is the list of sync validators. The following example shows how to add validators.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | this.contactForm = this.formBuilder.group({    firstname: ['', [Validators.required, Validators.minLength(10)]],    lastname: ['', [Validators.required, Validators.maxLength(15), Validators.pattern("^[a-zA-Z]+$")]],    email: ['', [Validators.required, Validators.email]],    gender: ['', [Validators.required]],    isMarried: ['', [Validators.required]],    country: ['', [Validators.required]],    address: this.formBuilder.group({      city: ['', [Validators.required]],      street: ['', [Validators.required]],      pincode: ['', [Validators.required]],    })  }); |

FormBuilder Example

We learned how to build reactive forms the [Angular Reactive forms tutorial](https://www.tektutorialshub.com/angular/angular-reactive-forms/).

**app.component.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123 | import { Component, ViewChild, ElementRef } from '@angular/core';  import { FormGroup, FormControl, Validators } from '@angular/forms'  import { FormBuilder } from '@angular/forms'  import { groupBy } from 'rxjs/internal/operators/groupBy';      @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {      title = 'Angular Reactive forms';      contactForm;      constructor(private formBuilder: FormBuilder) {            // this.contactForm = this.formBuilder.group({      //   firstname: [''],      //   lastname: [''],      //   email: [''],      //   gender: [''],      //   isMarried: [''],      //   country: [''],      // });        // this.contactForm = this.formBuilder.group({      //   firstname: [''],      //   lastname: [''],      //   email: [''],      //   gender: [''],      //   isMarried: [''],      //   country: [''],      //   address: this.formBuilder.group({      //     city: [''],      //     street: [''],      //     pincode: ['']      //   })      // });        this.contactForm = this.formBuilder.group({        firstname: ['', [Validators.required, Validators.minLength(10)]],        lastname: ['', [Validators.required, Validators.maxLength(15), Validators.pattern("^[a-zA-Z]+$")]],        email: ['', [Validators.required, Validators.email]],        gender: ['', [Validators.required]],        isMarried: ['', [Validators.required]],        country: ['', [Validators.required]],        address: this.formBuilder.group({          city: ['', [Validators.required]],          street: ['', [Validators.required]],          pincode: ['', [Validators.required]],        })      });    }        get firstname() {      return this.contactForm.get('firstname');    }      get lastname() {      return this.contactForm.get('lastname');    }      get email() {      return this.contactForm.get('email');    }      get gender() {      return this.contactForm.get('gender');    }      get isMarried() {      return this.contactForm.get('isMarried');    }      get country() {      return this.contactForm.get('country');    }      get city() {      return this.contactForm.get("address").get('city');    }      get street() {      return this.contactForm.get("address").get('street');    }      get pincode() {      return this.contactForm.get("address").get('pincode');    }        countryList: country[] = [      new country("1", "India"),      new country('2', 'USA'),      new country('3', 'England')    ];      onSubmit() {      console.log(this.contactForm.value);    }    }      export class country {    id: string;    name: string;      constructor(id: string, name: string) {      this.id = id;      this.name = name;    }  } |

**app.component.html**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147 | <div style="float: left; width:50%;">    <form [formGroup]="contactForm" (ngSubmit)="onSubmit()" novalidate>      <p>      <label for="firstname">First Name </label>      <input type="text" id="firstname" name="firstname" formControlName="firstname">    </p>      <div      \*ngIf="!firstname?.valid && (firstname?.dirty ||firstname?.touched)">      <div [hidden]="!firstname.errors.required">        First Name is required      </div>      <div [hidden]="!firstname.errors.minlength">        Min Length is 10      </div>    </div>      <p>      <label for="lastname">Last Name </label>      <input type="text" id="lastname" name="lastname" formControlName="lastname">    </p>      <div \*ngIf="!lastname.valid && (lastname.dirty ||lastname.touched)">      <div [hidden]="!lastname.errors.pattern">        Only characters are allowed      </div>      <div [hidden]="!lastname.errors.maxLength">        Max length allowed is {{lastname.errors.maxlength?.requiredLength}}      </div>      <div [hidden]="!lastname.errors.required">        Last Name is required      </div>    </div>      <p>      <label for="email">Email </label>      <input type="text" id="email" name="email" formControlName="email">    </p>    <div \*ngIf="!email.valid && (email.dirty ||email.touched)">      <div [hidden]="!email.errors.required">        email is required      </div>      <div [hidden]="!email.errors.email">        invalid email id      </div>    </div>        <p>      <label for="gender">Geneder </label>      <input type="radio" value="male" id="gender" name="gender" formControlName="gender"> Male      <input type="radio" value="female" id="gender" name="gender" formControlName="gender"> Female    </p>    <div \*ngIf="!gender.valid && (gender.dirty ||gender.touched)">      <div [hidden]="!gender.errors.required">        gender is required      </div>    </div>      <p>      <label for="isMarried">Married </label>      <input type="checkbox" id="isMarried" name="isMarried" formControlName="isMarried">    </p>    <div \*ngIf="!isMarried.valid && (isMarried.dirty ||isMarried.touched)">      <div [hidden]="!isMarried.errors.required">        isMarried is required      </div>    </div>        <p>      <label for="country">country </label>      <select id="country" name="country" formControlName="country">        <option [ngValue]="c.id" \*ngFor="let c of countryList">          {{c.name}}        </option>      </select>    </p>    <div \*ngIf="!country.valid && (country.dirty ||country.touched)">      <div [hidden]="!country.errors.required">        country is required      </div>    </div>      <div formGroupName="address">        <div class="form-group">        <label for="city">City</label>        <input type="text" class="form-control" name="city" formControlName="city">      </div>      <div \*ngIf="!city.valid && (city.dirty ||city.touched)">        <div [hidden]="!city.errors.required">          city is required        </div>      </div>          <div class="form-group">        <label for="street">Street</label>        <input type="text" class="form-control" name="street" formControlName="street">      </div>      <div \*ngIf="!street.valid && (street.dirty ||street.touched)">        <div [hidden]="!street.errors.required">          street is required        </div>      </div>        <div class="form-group">        <label for="pincode">Pin Code</label>        <input type="text" class="form-control" name="pincode" formControlName="pincode">      </div>      <div \*ngIf="!pincode.valid && (pincode.dirty ||pincode.touched)">        <div [hidden]="!pincode.errors.required">          pincode is required        </div>      </div>      </div>      <p>      <button type="submit" [disabled]="!contactForm.valid">Submit</button>    </p>    </form>    </div>    <div style="float: right; width:50%;">      <h3>Form Status</h3>    <b>valid : </b>{{contactForm.valid}}    <b>invalid : </b>{{contactForm.invalid}}    <b>touched : </b>{{contactForm.touched}}    <b>untouched : </b>{{contactForm.untouched}}    <b>pristine : </b>{{contactForm.pristine}}    <b>dirty : </b>{{contactForm.dirty}}    <b>disabled : </b>{{contactForm.disabled}}    <b>enabled : </b>{{contactForm.enabled}}      <h3>Form Value</h3>    {{contactForm.value |json}}    </div> |

References

[FormBuilder API](https://angular.io/api/forms/FormBuilder)

Summary

FormBuilder API makes it easier to work with the reactive forms in Angular. We can make use of the group, array & control methods to build ourFormModel. FormBuilder reduces the code required to write the complex forms.

Angular Forms

i.e FormControl, FormGroup & FormArray. All these components have methods setValue & patchValue and behave differently

SetValue

setValue(value: { [key: string]: any; }, options: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

We use the SetValue to update the FormControl , FormGroup or FormArray. When we use it to update the FormGroup or [FormArray](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/) the SetValue requires that the object must match the structure of the FormGroup or FormArray exactly. Otherwise, it will result in an error.

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

PatchValue

patchValue(value: { [key: string]: any; }, options: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

The PatchValue is used to update only a subset of the elements of the FormGroup or FormArray. It will only update the matching objects and ignores the rest.

onlySelf

The Angular checks the validation status of the form, whenever there is a change in value. [The validation](https://www.tektutorialshub.com/angular/formgroup-in-angular/) starts from the control whose value was changed and propagates to the top level FormGroup. This is the default behavior

There may be circumstances, where you do not want angular to check the validity of the entire form, whenever you change the value using the setValue or patchValue. We do that by setting the onlySelf=true as the argument. In such cases, the angular only checks the validity of the current control, but does not check any other control and does not propagate the validity checking to the parent form group.

emitEvent

The Angular forms emit two events. One is [ValueChanges](https://www.tektutorialshub.com/angular/valuechanges-in-angular-forms/) & the other one is StatusChanges. The ValueChanges event is emitted whenever the value of the form is changed. The StatusChanges event is emitted whenever angular calculates the validation status of the Form. This is the default behavior

We can stop that from happening, by setting the emitEvent=false

SetValue Vs PatchValue

The difference is that with setValue we must include all the controls, while with the patchValue you can exclude some controls.

Example form setup

Create a new angular application. Import both FormsModule, ReactiveFormsModule from @angular/forms. Also add it into the imports metadata

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule, ReactiveFormsModule } from '@angular/forms';    import { AppRoutingModule } from './app-routing.module';  import { AppComponent } from './app.component';  import { TemplateComponent } from './template-component';  import { ReactiveComponent } from './reactive.component';    @NgModule({    declarations: [      AppComponent,TemplateComponent,ReactiveComponent    ],    imports: [      BrowserModule,      AppRoutingModule,      FormsModule,      ReactiveFormsModule    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

Create two new components reactive.component.ts & template-component.ts with their respective templates. Also, update the app.component.ts & its template as shown below

app.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import { Component} from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {  } |

app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | <h1>Angular Forms SetValue & PatchValue Example</h1>    <ul>    <li>      <a [routerLink]="['/template']" routerLinkActive="router-link-active" >Template</a>    </li>    <li>      <a [routerLink]="['/reactive']" routerLinkActive="router-link-active" >Reactive</a>    </li>  </ul>    <router-outlet></router-outlet> |

SetValue & PatchValue in Reactive Forms

Here is our template-component.ts & template-component.html.

reactive-component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150 | import { Component, OnInit } from '@angular/core';  import { FormGroup, FormControl, Validators } from '@angular/forms'      @Component({    templateUrl: './reactive.component.html',  })  export class ReactiveComponent implements OnInit {    title = 'Reactive Forms';        countryList: country[] = [      new country("1", "India"),      new country('2', 'USA'),      new country('3', 'England')    ];      // reactiveForm = new FormGroup({    //   firstname: new FormControl('Sachin'),    //   lastname: new FormControl('Tendulkar'),    //   email: new FormControl('sachin@gmail.com'),    //   gender: new FormControl('male'),    //   isMarried: new FormControl(true),    //   country: new FormControl('2'),    //   address:new FormGroup({    //     city: new FormControl("Mumbai"),    //     street: new FormControl("Perry Cross Rd"),    //     pincode:new FormControl("400050")    //   })    // })      reactiveForm = new FormGroup({      firstname: new FormControl(),      lastname: new FormControl(),      email: new FormControl(),      gender: new FormControl(),      isMarried: new FormControl(),      country: new FormControl(),      address:new FormGroup({        city: new FormControl(),        street: new FormControl(),        pincode:new FormControl()      })    })      onSubmit() {      console.log(this.reactiveForm.value);    }      ngOnInit() {      this.setDefault();    }      setDefault() {        let contact = {        firstname: "Sachin",        lastname: "Tendulkar",        email: "sachin@gmail.com",        gender: "male",        isMarried: true,        country: "2",        address: {          city: "Mumbai",          street: "Perry Cross Rd",          pincode: "400050"        }      };        this.reactiveForm.setValue(contact);    }      setValue() {        let contact = {        firstname: "Rahul",        lastname: "Dravid",        email: "rahul@gmail.com",        gender: "male",        isMarried: true,        country: "1",        address: {          city: "Bangalore",          street: "Brigade Road",          pincode: "600070"        }      };        this.reactiveForm.setValue(contact);    }      setAddress() {        let address= {        city: "Bangalore",        street: "Brigade Road",        pincode: "600070",      };        this.reactiveForm.get("address").setValue(address);      };      setCountry() {        this.reactiveForm.get("country").setValue("1");      };        patchAddress() {        let address= {        city: "Bangalore",        street: "Brigade Road",        //pincode: "600070",        //firstname:'saurv'      };        this.reactiveForm.get("address").patchValue(address);      }      patchName() {      let contact = {        firstname: "Rahul",        lastname: "Dravid",      }        this.reactiveForm.patchValue(contact);      }      reset() {      this.reactiveForm.reset();    }    }    export class country {    id: string;    name: string;      constructor(id: string, name: string) {      this.id = id;      this.name = name;    }  } |

reactive-component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100 | <h3>{{title}}</h3>    <div style="float: left; width:50%;">      <form [formGroup]="reactiveForm" (ngSubmit)="onSubmit()" novalidate>        <p>        <label for="firstname">First Name </label>        <input type="text" id="firstname" name="firstname" formControlName="firstname">      </p>        <p>        <label for="lastname">Last Name </label>        <input type="text" id="lastname" name="lastname" formControlName="lastname">      </p>        <p>        <label for="email">Email </label>        <input type="text" id="email" name="email" formControlName="email">      </p>        <p>        <label for="gender">Geneder </label>        <input type="radio" value="male" id="gender" name="gender" formControlName="gender"> Male        <input type="radio" value="female" id="gender" name="gender" formControlName="gender"> Female      </p>        <p>        <label for="isMarried">Married </label>        <input type="checkbox" id="isMarried" name="isMarried" formControlName="isMarried">      </p>        <p>        <label for="country">country </label>        <select id="country" name="country" formControlName="country">          <option [ngValue]="c.id" \*ngFor="let c of countryList">            {{c.name}}          </option>        </select>      </p>          <div formGroupName="address">          <p>          <label for="city">City</label>          <input type="text" class="form-control" name="city" formControlName="city">        </p>          <p>          <label for="street">Street</label>          <input type="text" class="form-control" name="street" formControlName="street">        </p>          <p>          <label for="pincode">Pin Code</label>          <input type="text" class="form-control" name="pincode" formControlName="pincode">        </p>        </div>          <button>Submit</button>      <div>        <button type="button" (click)="setDefault()">Default</button>      </div>      <div>        <button type="button" (click)="setValue()">SetValue</button>        <button type="button" (click)="setAddress()">Address</button>        <button type="button" (click)="setCountry()">Country</button>      </div>      <div>        <button type="button" (click)="patchName()">Name</button>        <button type="button" (click)="patchAddress()">Address</button>        <button type="button" (click)="reset()">Reset</button>      </div>        </form>  </div>    <div style="float: right; width:50%;">      <h3>Form Status</h3>    <b>valid : </b>{{reactiveForm.valid}}    <b>invalid : </b>{{reactiveForm.invalid}}    <b>touched : </b>{{reactiveForm.touched}}    <b>untouched : </b>{{reactiveForm.untouched}}    <b>pristine : </b>{{reactiveForm.pristine}}    <b>dirty : </b>{{reactiveForm.dirty}}    <b>disabled : </b>{{reactiveForm.disabled}}    <b>enabled : </b>{{reactiveForm.enabled}}        <h3>Form Value</h3>    {{reactiveForm.value |json}}    </div> |

Setting Initial /Default Value

There are two ways, in which set the initial value. One at the time of defining the Form Model as the first argument to the FormControl as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | reactiveForm = new FormGroup({      firstname: new FormControl('Sachin'),      lastname: new FormControl('Tendulkar'),      email: new FormControl('sachin@gmail.com'),      gender: new FormControl('male'),      isMarried: new FormControl(true),      country: new FormControl('2'),      address:new FormGroup({        city: new FormControl("Mumbai"),        street: new FormControl("Perry Cross Rd"),        pincode:new FormControl("400050")      })    }) |

Another option is to use the setValue in ngOnInit method. To do that, first, create a contact object with the properties exactly matching the Form Model and then invoke the setValue as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | ngOnInit() {    this.setDefault();  }    setDefault() {        let contact = {        firstname: "Sachin",        lastname: "Tendulkar",        email: "sachin@gmail.com",        gender: "male",        isMarried: true,        country: "2",        address: {          city: "Mumbai",          street: "Perry Cross Rd",          pincode: "400050"        }      };        this.reactiveForm.setValue(contact);    } |

The advantageous of the second option is that you can call the setDefault any time and set the default values again.

As said earlier, the setValue only works, when the properties match exactly. If you remove any of the properties or add a new property, then it will result in an error.

Ex: if you comment out isMarried field, then you will see the following error in the console window.

|  |  |
| --- | --- |
| 1  2  3 | Must supply a value for form control with name: 'isMarried'. |

Or if you add a new property surname, you will see the following error.

|  |  |
| --- | --- |
| 1  2  3 | Cannot find form control with name: surname. |

Nested FormGroup

As mentioned earlier, the setValue updates the entire FormGroup. Hence we can update the nested form group separately.

In the following example, we get the reference to the address form group and then invoke the setValue to update only the address.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | setAddress() {        let address= {        city: "Bangalore",        street: "Brigade Road",        pincode: "600070",      };        this.reactiveForm.get("address").setValue(address);      }; |

Here again, the properties of the address must match completely. Otherwise, it will result in an error.

FormControl

The value of individual control can be easily set

|  |  |
| --- | --- |
| 1  2  3  4  5 | setCountry() {      this.reactiveForm.get("country").setValue("1");  }; |

PatchValue

We use patchValue when we want to update only the subset of properties.

For Example, the following shows how to update only city & street properties using the patchValue method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | patchAddress() {        let address= {        city: "Bangalore",        street: "Brigade Road",      };        this.reactiveForm.get("address").patchValue(address);    } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | patchName() {      let contact = {        firstname: "Rahul",        lastname: "Dravid",      }        this.reactiveForm.patchValue(contact);      } |

Reset Form

|  |  |
| --- | --- |
| 1  2  3  4  5 | reset() {    this.reactiveForm.reset();  } |

OnlySelf Example

The angular forms calculate the validity status of the form, whenever the values of any of the controls on the form change. The validation check starts from the control and is run for the parent control until it reaches the top-level FormGroup.

We can use the onlySelf:true argument to tell angular not to run validation on the parent control.

For Example, we have added a required validator to the firstname FormControl. Now enter some text in the firstname field to make the form Valid and then set the value to blank as shown below. **The Form becomes invalid**.

|  |  |
| --- | --- |
| 1  2  3  4  5 | withOutOnlySelf(){      this.reactiveForm.get("firstname").setValue("");    } |

Make the form valid again by entering some text in the firstname field. Now, try the same with onlySelf:true added. **The Form stays Valid**.

|  |  |
| --- | --- |
| 1  2  3  4  5 | withOnlySelf(){      this.reactiveForm.get("firstname").setValue("",{onlySelf:true});    } |

emitEvent example

The Angular forms emit two events. One is ValueChanges & the Other one is statusChanges. You can stop them from happening using the emitEvent:false argument as shown below.

First, subscribe to statusChanges & valueChanges event at Form Level and also at the control level.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | ngOnInit() {      this.setDefault();        this.reactiveForm.get("firstname").statusChanges.subscribe(x => {        console.log('firstname status changes')      })        this.reactiveForm.get("firstname").valueChanges.subscribe(x => {        console.log('firstname value changed')      })        this.reactiveForm.statusChanges.subscribe(x => {        console.log('form status changes')      })        this.reactiveForm.valueChanges.subscribe(x => {        console.log('form value changed')      })    } |

And then change the value of the firstname and you will see all the four events are fired.

|  |  |
| --- | --- |
| 1  2  3  4  5 | withouEmitEvent(){    this.reactiveForm.get("firstname").setValue("Sachin");  } |

And when you use the emitEvent:false the events are suppressed.

|  |  |
| --- | --- |
| 1  2  3  4  5 | withEmitEvent(){    this.reactiveForm.get("firstname").setValue("",{emitEvent:false});  } |

SetValue & PatchValue in Template-driven Forms

You can make use of the setValue & patchValue in [template-driven forms](https://www.tektutorialshub.com/angular/angular-template-driven-forms/) also. We learned how to do it in [set Value in template-driven forms in the angular](https://www.tektutorialshub.com/angular/how-to-set-value-in-template-driven-forms-in-angular/) tutorial.

To do that, we first need the reference to the Form model in the template, using the viewchild

|  |  |
| --- | --- |
| 1  2  3 | @ViewChild('templateForm',null) templateForm: NgForm; |

Once, we have the reference, you can make use of SetValue & PatchValue as shown in the following examples. For a more detailed explanation refer to the tutorial [Set Value in template-driven forms in the angular](https://www.tektutorialshub.com/angular/how-to-set-value-in-template-driven-forms-in-angular/)

template-component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153 | import { Component, ViewChild, ElementRef, OnInit } from '@angular/core';  import { NgForm } from '@angular/forms';      @Component({    templateUrl: './template.component.html',  })  export class TemplateComponent implements OnInit {      title = 'Template driven forms';      @ViewChild('templateForm',null) templateForm: NgForm;      countryList: country[] = [      new country("1", "India"),      new country('2', 'USA'),      new country('3', 'England')    ];      contact: contact;      onSubmit() {      console.log(this.templateForm.value);    }      ngOnInit() {        setTimeout(() => {        this.setDefault();      });      }      setDefault() {        let contact = {        firstname: "Sachin",        lastname: "Tendulkar",        email: "sachin@gmail.com",        gender: "male",        isMarried: true,        country: "2",        address: {          city: "Mumbai",          street: "Perry Cross Rd",          pincode: "400050"        }      };        this.templateForm.control.setValue(contact);    }        setValue() {        let contact = {        firstname: "Rahul",        lastname: "Dravid",        email: "rahul@gmail.com",        gender: "male",        isMarried: true,        country: "1",        address: {          city: "Bangalore",          street: "Brigade Road",          pincode: "600070"        }      };        this.templateForm.setValue(contact);    }      setAddress() {        let address= {        city: "Bangalore",        street: "Brigade Road",        pincode: "600070"      };        this.templateForm.control.get("address").setValue(address);      };      setCountry() {        let address= {        city: "Bangalore",        street: "Brigade Road",        pincode: "600070"      };        this.templateForm.control.get("country").setValue("1");      };        patchAddress() {        let address= {        city: "Bangalore",        street: "Brigade Road",        //pincode: "600070",        //firstname:'saurv'      };        this.templateForm.control.get("address").patchValue(address);      }      patchName() {      let contact = {        firstname: "Rahul",        lastname: "Dravid",      }        this.templateForm.control.patchValue(contact);      }      reset() {      this.templateForm.reset();    }    }    export class contact {    firstname:string;    lastname:string;    gender:string;    email:string;    isMarried:boolean;    country:string;    address: {      city:string;      street:string;      pincode:string;    }  }      export class country {    id: string;    name: string;      constructor(id: string, name: string) {      this.id = id;      this.name = name;    }  } |

template-component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 | <h3>{{title}}</h3>    <div style="float: left; width:50%;">    <form #templateForm="ngForm" (ngSubmit)="onSubmit(templateForm)">        <p>        <label for="firstname">First Name </label>        <input type="text" id="firstname" name="firstname" #fname="ngModel" ngModel>        </p>      <p>        <label for="lastname">Last Name </label>        <input type="text" id="lastname" name="lastname" ngModel>      </p>        <p>        <label for="email">Email </label>        <input type="text" id="email" name="email" ngModel>      </p>          <p>        <label for="gender">Geneder </label>        <input type="radio" value="male" id="gender" name="gender" ngModel> Male        <input type="radio" value="female" id="gender" name="gender" ngModel> Female      </p>        <p>        <label for="isMarried">Married </label>        <input type="checkbox" id="isMarried" name="isMarried" ngModel>      </p>        <p>        <label for="country">country </label>        <select id="country" name="country" ngModel>          <option [ngValue]="c.id" \*ngFor="let c of countryList">            {{c.name}}          </option>        </select>      </p>        <div ngModelGroup="address">          <p>          <label for="city">City</label>          <input type="text" id="city" name="city" ngModel>        </p>          <p>          <label for="street">Street</label>          <input type="text" id="street" name="street" ngModel>        </p>        <p>          <label for="pincode">Pin Code</label>          <input type="text" id="pincode" name="pincode" ngModel>        </p>        </div>        <p>        <button type="submit">Submit</button>      </p>        <div>        <button type="button" (click)="setDefault()">Default</button>      </div>      <div>        <button type="button" (click)="setValue()">SetValue</button>        <button type="button" (click)="setAddress()">Address</button>        <button type="button" (click)="setCountry()">Country</button>      </div>      <div>        <button type="button" (click)="patchName()">Name</button>        <button type="button" (click)="patchAddress()">Address</button>        <button type="button" (click)="reset()">Reset</button>      </div>        </form>  </div>    <div style="float: right; width:50%;">    <h3>Form Status</h3>    <b>valid : </b>{{templateForm.valid}}    <b>invalid : </b>{{templateForm.invalid}}    <b>touched : </b>{{templateForm.touched}}    <b>untouched : </b>{{templateForm.untouched}}    <b>pristine : </b>{{templateForm.pristine}}    <b>dirty : </b>{{templateForm.dirty}}    <b>disabled : </b>{{templateForm.disabled}}    <b>enabled : </b>{{templateForm.enabled}}        <h3>Form Value</h3>    {{templateForm.value | json }}    </div>> |

Summary

In this tutorial, we learned how to use setValue & pathcValue to set the values of the [Reactive forms in Angular](https://www.tektutorialshub.com/angular/angular-reactive-forms/).

How to use StatusChanges

The [Angular Forms](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/) has three building blocks. FormControl, [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) & FormArray. All of these controls extend the AbstractControl base class. The AbstractControl base class implements StatusChanges event

We can subscribe to StatusChanges by getting the reference of the control and subscribing it as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | this.reactiveForm.get("firstname").statusChanges.subscribe(newStaus => {     console.log('firstname status changed')     console.log(newStaus)  }) |

You can also subscribe to the top-level form as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | this.reactiveForm.statusChanges.subscribe(newStaus => {      console.log('form Status changed event')      console.log(newStaus)  }) |

StatusChanges Example

Create a [reactive form](https://www.tektutorialshub.com/angular/angular-reactive-forms/) as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | reactiveForm = new FormGroup({     firstname: new FormControl('', [Validators.required]),     lastname: new FormControl(),     address: new FormGroup({       city: new FormControl(),       street: new FormControl(),       pincode: new FormControl()     })  }) |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

StatusChanges of FormControl

You can subscribe to StatusChanges of a single FormControl as shown below. Here in the newStatus variable, we will get the latest status of the firstname. You can also retreive the latest status of the firstname using this.reactiveForm.get("firstname").status

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | this.reactiveForm.get("firstname").statusChanges.subscribe(newStatus => {     console.log('firstname status changed')     console.log(newStatus)                                   //latest status     console.log(this.reactiveForm.get("firstname").status)  //latest status  }) |

But, the top-level form is not yet updated at this point, hence this.reactiveForm.status still shows the old status of the firstname and also the form.

The statusChanges event for the firstname fires immediately **after** the new status is updated but **before** the change is bubbled up to its parent. Hence the this.reactiveForm.status still shows the old status.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | this.reactiveForm.get("firstname").statusChanges.subscribe(newStatus=> {     console.log('firstname status changed')     console.log(newStatus)                                   //latest status     console.log(this.reactiveForm.get("firstname").status)  //latest status     console.log(this.reactiveForm.status)                   //Previous status  }) |

You can work around this by waiting for the next tick using setTimeout as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | this.reactiveForm.get("firstname").statusChanges.subscribe(newStatus=> {     console.log('firstname status changed')     console.log(newStatus)                                    //latest status     console.log(this.reactiveForm.get("firstname").status)   //latest status     console.log(this.reactiveForm.status)                    //Previous status       setTimeout(() => {       console.log(this.reactiveForm.status)                  //latest status     })    }) |

StatusChanges of FormGroup

The StatusChanges event of FormGroup or [FormArray](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/) is fired, whenever the status of any of its child controls are calculated. For Example, the following StatusChanges will fire even whenever the status of the city, state & pincode are calculated.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | this.reactiveForm.get("address").statusChanges.subscribe(newStaus => {    console.log('address status changed')    console.log(newStaus)  }) |

StatusChanges of Form

The following example show we can subscribe to the changes made to the entire form.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | this.reactiveForm.statusChanges.subscribe(newStaus => {    console.log('form status changed')    console.log(newStaus)  }) |

emitEvent & StatusChanges

The statusChanges event is fired even when the angular calculates the status of the control either via UI or programmatically. In some circumstances, you might not want to raise the statusChanges event. To do that we can use the emitEvent: false

In the following example, the statusChanges event is **not fired** at all, even though the value of the firstname is changed making it and the form INVALID.

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.get("firstname").setValue("", { emitEvent: false }); |

You can use emitEvent: false with the setValue, patchValue, markAsPending, disable, enable, updateValueAndValidity & setErrors methods.

onlySelf & StatusChanges

When onlySelf: true the changes will only affect only this FormControl and change is **not** bubbled up to its parent. Hence the StatusChanges event of the parent FormGroup does not fire.

For Example, the following code will result in the StatusChanges of the firstname. but not of its parent (i.e. top-level form)

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.get("firstname").setValue("", { onlySelf: true }); |

You can use the onlySelf: true with the setValue, patchValue, markAsUntouched, markAsDirty, markAsPristine, markAsPending, disable, enable, and updateValueAndValidity methods

Complete Source Code

reactive.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101 | import { Component, OnInit } from '@angular/core';  import { FormGroup, FormControl, Validators } from '@angular/forms'  import { timeout } from 'q';      @Component({    templateUrl: './reactive.component.html',  })  export class ReactiveComponent implements OnInit {    title = 'Reactive Forms';      reactiveForm = new FormGroup({      firstname: new FormControl('', [Validators.required]),      lastname: new FormControl(),      address: new FormGroup({        city: new FormControl(),        street: new FormControl(),        pincode: new FormControl()      })    })      onSubmit() {      //console.log(this.reactiveForm.value);    }      ngOnInit() {        this.reactiveForm.get("firstname").statusChanges.subscribe(newStatus=> {        console.log('firstname status changed')        console.log(newStatus)        console.log(this.reactiveForm.get("firstname").status)        console.log(this.reactiveForm.status)          setTimeout(() => {          console.log(this.reactiveForm.status)        })        })        this.reactiveForm.get("address").statusChanges.subscribe(newStatus=> {        console.log('address status changed')        console.log(newStatus)      })        this.reactiveForm.statusChanges.subscribe(newStatus=> {        console.log('form status changed')        console.log(newStatus)      })    }      setValue() {        let contact = {        firstname: "Rahul",        lastname: "Dravid",        address: {          city: "Bangalore",          street: "Brigade Road",          pincode: "600070"        }      };        this.reactiveForm.setValue(contact);    }      setAddress() {        this.reactiveForm.get("address").setValue(        {          city: "Bangalore",          street: "Brigade Road",          pincode: "600070"        }      );    }      setFirstname() {      this.reactiveForm.get("firstname").setValue("Saurav")    }      withoutOnlySelf() {      this.reactiveForm.get("firstname").setValue("");    }    withOnlySelf() {      this.reactiveForm.get("firstname").setValue("", { onlySelf: true });    }      withEmitEvent() {      this.reactiveForm.get("firstname").setValue("Sachin");    }    withoutEmitEvent() {      this.reactiveForm.get("firstname").setValue("", { emitEvent: false });    }      reset() {      this.reactiveForm.reset();    }    } |

reactive.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67 | <h3>{{title}}</h3>    <div style="float: left; width:50%;">      <form [formGroup]="reactiveForm" (ngSubmit)="onSubmit()" novalidate>        <p>        <label for="firstname">First Name </label>        <input type="text" id="firstname" name="firstname" formControlName="firstname">        <label for="lastname">Last Name </label>        <input type="text" id="lastname" name="lastname" formControlName="lastname">      </p>        <div formGroupName="address">          <p>          <label for="city">City</label>          <input type="text" class="form-control" name="city" formControlName="city">          <label for="street">Street</label>          <input type="text" class="form-control" name="street" formControlName="street">          <label for="pincode">Pin Code</label>          <input type="text" class="form-control" name="pincode" formControlName="pincode">        </p>        </div>        <button>Submit</button>      </form>      <div>      <button type="button" (click)="setValue()">SetValue</button>      <button type="button" (click)="setAddress()">Address</button>      <button type="button" (click)="setFirstname()">First Name</button>    </div>    <div>      <button type="button" (click)="withoutOnlySelf()">Without Only Self</button>      <button type="button" (click)="withOnlySelf()">With Only Self</button>    </div>    <div>      <button type="button" (click)="withouEmitEvent()">Without EmitEvent</button>      <button type="button" (click)="withEmitEvent()">With EmitEvent</button>    </div>      </div>    <div style="float: right; width:50%;">      <h3>Form Status</h3>    <b>status : </b>{{reactiveForm.status}}    <b>valid : </b>{{reactiveForm.valid}}    <b>invalid : </b>{{reactiveForm.invalid}}    <b>touched : </b>{{reactiveForm.touched}}    <b>untouched : </b>{{reactiveForm.untouched}}    <b>pristine : </b>{{reactiveForm.pristine}}    <b>dirty : </b>{{reactiveForm.dirty}}    <b>disabled : </b>{{reactiveForm.disabled}}    <b>enabled : </b>{{reactiveForm.enabled}}        <h3>Form Value</h3>    {{reactiveForm.value |json}}    </div> |

app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | <h3>Angular StatusChanges Example</h3>    <ul>    <li>      <a [routerLink]="['/template']" routerLinkActive="router-link-active" >Template</a>    </li>    <li>      <a [routerLink]="['/reactive']" routerLinkActive="router-link-active" >Reactive</a>    </li>  </ul>    <router-outlet></router-outlet> |

app.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import { Component} from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {  } |

app.module.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule, ReactiveFormsModule } from '@angular/forms';    import { AppRoutingModule } from './app-routing.module';  import { AppComponent } from './app.component';  import { TemplateComponent } from './template-component';  import { ReactiveComponent } from './reactive.component';    @NgModule({    declarations: [      AppComponent,TemplateComponent,ReactiveComponent    ],    imports: [      BrowserModule,      AppRoutingModule,      FormsModule,      ReactiveFormsModule    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

StatusChanges in Template Driven Forms

StatusChanges event can also be used in the [template-driven forms](https://www.tektutorialshub.com/angular/angular-template-driven-forms/). All you need to do is to get the reference to the Form Model in the component as shown below

|  |  |
| --- | --- |
| 1  2  3 | @ViewChild('templateForm',null) templateForm: NgForm; |

You can refer to the example code below

template-component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119 | import { Component, ViewChild, ElementRef, OnInit, OnDestroy } from '@angular/core';  import { NgForm } from '@angular/forms';      @Component({    templateUrl: './template.component.html',  })  export class TemplateComponent implements OnInit {      title = 'Template driven forms';      @ViewChild('templateForm',null) templateForm: NgForm;      contact: contact;      onSubmit() {      console.log(this.templateForm.value);    }      ngOnInit() {        setTimeout(() => {          this.templateForm.control.get("firstname").statusChanges.subscribe(newStatus=> {          console.log('firstname status changed')          console.log(newStatus)          console.log(this.templateForm.control.get("firstname").status)          console.log(this.templateForm.control.status)            setTimeout(() => {            console.log(this.templateForm.control.status)          })          })          this.templateForm.control.get("address").statusChanges.subscribe(newStatus => {          console.log('address status changed')          console.log(newStatus)        })          this.templateForm.control.statusChanges.subscribe(newStatus=> {          console.log('form status changed')          console.log(newStatus)        })        });            }          setValue() {      let contact = {        firstname: "Rahul",        lastname: "Dravid",        address: {          city: "Bangalore",          street: "Brigade Road",          pincode: "600070"        }      };        this.templateForm.setValue(contact);    }      setAddress() {      let address= {        city: "Bangalore",        street: "Brigade Road",        pincode: "600070"      };        this.templateForm.control.get("address").setValue(address);      };      setFirstname() {      this.templateForm.control.get("firstname").setValue("Saurav")    }        withoutOnlySelf() {      this.templateForm.control.get("firstname").setValue("");    }    withOnlySelf() {      this.templateForm.control.get("firstname").setValue("", { onlySelf: true });    }      withouEmitEvent() {      this.templateForm.control.get("firstname").setValue("Sachin");    }    withEmitEvent() {      this.templateForm.control.get("firstname").setValue("", { emitEvent: false });    }      reset() {      this.templateForm.reset();    }    }    export class contact {    firstname:string;    lastname:string;    gender:string;    email:string;    isMarried:boolean;    country:string;    address: {      city:string;      street:string;      pincode:string;    }  } |

template-component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70 | <h3>{{title}}</h3>    <div style="float: left; width:50%;">    <form #templateForm="ngForm" (ngSubmit)="onSubmit(templateForm)">        <p>        <label for="firstname">First Name </label>        <input type="text" id="firstname" name="firstname" #fname="ngModel" ngModel>        </p>      <p>        <label for="lastname">Last Name </label>        <input type="text" id="lastname" name="lastname" ngModel>      </p>        <div ngModelGroup="address">          <p>          <label for="city">City</label>          <input type="text" id="city" name="city" ngModel>          <label for="street">Street</label>          <input type="text" id="street" name="street" ngModel>          <label for="pincode">Pin Code</label>          <input type="text" id="pincode" name="pincode" ngModel>        </p>        </div>        <button>Submit</button>      </form>      <div>      <button type="button" (click)="setValue()">SetValue</button>      <button type="button" (click)="setAddress()">Address</button>      <button type="button" (click)="setFirstname()">First Name</button>    </div>    <div>      <button type="button" (click)="withoutOnlySelf()">Without Only Self</button>      <button type="button" (click)="withOnlySelf()">With Only Self</button>    </div>    <div>      <button type="button" (click)="withouEmitEvent()">Without EmitEvent</button>      <button type="button" (click)="withEmitEvent()">With EmitEvent</button>    </div>          </div>    <div style="float: right; width:50%;">    <h3>Form Status</h3>    <b>status : </b>{{templateForm.status}}    <b>valid : </b>{{templateForm.valid}}    <b>invalid : </b>{{templateForm.invalid}}    <b>touched : </b>{{templateForm.touched}}    <b>untouched : </b>{{templateForm.untouched}}    <b>pristine : </b>{{templateForm.pristine}}    <b>dirty : </b>{{templateForm.dirty}}    <b>disabled : </b>{{templateForm.disabled}}    <b>enabled : </b>{{templateForm.enabled}}        <h3>Form Value</h3>    {{templateForm.value | json }}    </div> |

Summary

In this tutorial, we learned how to make use of StatusChanges in Angular Forms. The StatusChanges event is fired whenever the angular calculates the validity status of the FormControl, FormGroup or FormArray. It is an observable and we can subscribe to it. The StatusChanges event does not fire depending on how we set emitEvent or onlySelf, when updating the value and validity of the form controls.

How to use ValueChanges

The [Angular Forms](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/) has three building blocks. FormControl, [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) & FormArray. All of these controls extend the AbstractControl base class. The AbstractControl base class implements ValueChanges event

We can subscribe to ValueChanges by getting the reference of the control and subscribing it as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | this.reactiveForm.get("firstname").valueChanges.subscribe(x => {     console.log('firstname value changed')     console.log(x)  }) |

You can also subscribe to the top-level form as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | this.reactiveForm.valueChanges.subscribe(x => {      console.log('form value changed')      console.log(x)  }) |

ValueChanges Example

Create a [reactive form](https://www.tektutorialshub.com/angular/angular-reactive-forms/) as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | reactiveForm = new FormGroup({     firstname: new FormControl('', [Validators.required]),     lastname: new FormControl(),     address: new FormGroup({       city: new FormControl(),       street: new FormControl(),       pincode: new FormControl()     })  }) |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

ValueChanges of FormControl

You can subscribe to ValueChanges of a single FormControl as shown below. Here in selectedValue variable, we will get the latest value of the firstname. You can also retrieve the latest value of the firstname using this.reactiveForm.get("firstname").value

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | this.reactiveForm.get("firstname").valueChanges.subscribe(selectedValue => {    console.log('firstname value changed')    console.log(selectedValue)                              //latest value of firstname    console.log(this.reactiveForm.get("firstname").value)   //latest value of firstname  }) |

ValueChanges shows previous value

But, the top-level form is not yet updated at this point, hence this.reactiveForm.value still shows the previous value of the firstname.

The valueChanges event for the firstname fires immediately **after** the new value is updated but **before** the change is bubbled up to its parent. Hence the this.reactiveForm.value still shows the previous value.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | this.reactiveForm.get("firstname").valueChanges.subscribe(selectedValue => {    console.log('firstname value changed')    console.log(selectedValue)    console.log(this.reactiveForm.get("firstname").value)    console.log(this.reactiveForm.value)   //still shows the old first name  }) |

You can work around this by waiting for the next tick using setTimeout as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | this.reactiveForm.get("firstname").valueChanges.subscribe(selectedValue => {    console.log('firstname value changed')    console.log(selectedValue)    console.log(this.reactiveForm.get("firstname").value)    console.log(this.reactiveForm.value)    //shows the old first name      setTimeout(() => {      console.log(this.reactiveForm.value)   //shows the latest first name    })    }) |

ValueChanges of FormGroup

The ValueChanges event of FormGroup or FormArray is fired, whenever the value of any of its child controls value changes. For Example, the following ValueChanges will fire even whenever the value of the city, state & pincode changes.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | this.reactiveForm.get("address").valueChanges.subscribe(selectedValue  => {    console.log('address changed')    console.log(selectedValue)  }) |

ValueChanges of Form

The following example show we can subscribe to the changes made to the entire form.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | this.reactiveForm.valueChanges.subscribe(selectedValue  => {    console.log('form value changed')    console.log(selectedValue)  }) |

EmitEvent & ValueChanges

The ValueChanges event is fired even when the values of the control are changed programmatically. In some circumstances, you might not want to raise the ValueChanges event. To do that we can use the emitEvent: false

In the following example, the ValueChanges event is **not fired** at all, even though the value of the firstname is changed.

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.get("firstname").setValue("", { emitEvent: false }); |

You can use emitEvent: false with the [setValue,](https://www.tektutorialshub.com/angular/setvalue-patchvalue-in-angular/) patchValue, markAsPending, disable, enable, updateValueAndValidity & setErrors methods.

OnlySelf & ValueChanges

WhenonlySelf: true the changes will only affect only this FormControl and change is **not** bubbled up to its parent. Hence the ValueChanges event of the parent FormGroup does not fire.

For Example, the following code will result in the ValueChanges of the firstname. but not of its parent (i.e. top-level form)

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.get("firstname").setValue("", { onlySelf: true }); |

You can use the onlySelf: true with the setValue, patchValue, markAsUntouched, markAsDirty, markAsPristine, markAsPending, disable, enable, and updateValueAndValidity methods

Complete Source Code

reactive.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103 | import { Component, OnInit } from '@angular/core';  import { FormGroup, FormControl, Validators } from '@angular/forms'  import { timeout } from 'q';      @Component({    templateUrl: './reactive.component.html',  })  export class ReactiveComponent implements OnInit {    title = 'Reactive Forms';      reactiveForm = new FormGroup({      firstname: new FormControl('', [Validators.required]),      lastname: new FormControl(),      address: new FormGroup({        city: new FormControl(),        street: new FormControl(),        pincode: new FormControl()      })    })      onSubmit() {      console.log(this.reactiveForm.value);    }      ngOnInit() {        this.reactiveForm.get("firstname").valueChanges.subscribe(selectedValue => {        console.log('firstname value changed')        console.log(selectedValue)        console.log(this.reactiveForm.get("firstname").value)        console.log(this.reactiveForm.value)          setTimeout(() => {          console.log(this.reactiveForm.value)        })        })        this.reactiveForm.get("address").valueChanges.subscribe(selectedValue => {        console.log('address changed')        console.log(selectedValue)      })        this.reactiveForm.valueChanges.subscribe(selectedValue => {        console.log('form value changed')        console.log(selectedValue)      })    }          setValue() {        let contact = {        firstname: "Rahul",        lastname: "Dravid",        address: {          city: "Bangalore",          street: "Brigade Road",          pincode: "600070"        }      };        this.reactiveForm.setValue(contact);    }      setAddress() {        this.reactiveForm.get("address").setValue(        {          city: "Bangalore",          street: "Brigade Road",          pincode: "600070"        }      );    }      setFirstname() {      this.reactiveForm.get("firstname").setValue("Saurav")    }      withoutOnlySelf() {      this.reactiveForm.get("firstname").setValue("");    }    withOnlySelf() {      this.reactiveForm.get("firstname").setValue("", { onlySelf: true });    }      withEmitEvent() {      this.reactiveForm.get("firstname").setValue("Sachin");    }    withoutEmitEvent() {      this.reactiveForm.get("firstname").setValue("", { emitEvent: false });    }      reset() {      this.reactiveForm.reset();    }    } |

reactive.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | <h3>{{title}}</h3>    <div style="float: left; width:50%;">      <form [formGroup]="reactiveForm" (ngSubmit)="onSubmit()" novalidate>        <p>        <label for="firstname">First Name </label>        <input type="text" id="firstname" name="firstname" formControlName="firstname">        <label for="lastname">Last Name </label>        <input type="text" id="lastname" name="lastname" formControlName="lastname">      </p>        <div formGroupName="address">          <p>          <label for="city">City</label>          <input type="text" class="form-control" name="city" formControlName="city">          <label for="street">Street</label>          <input type="text" class="form-control" name="street" formControlName="street">          <label for="pincode">Pin Code</label>          <input type="text" class="form-control" name="pincode" formControlName="pincode">        </p>        </div>          <button>Submit</button>      <div>        <button type="button" (click)="setValue()">SetValue</button>        <button type="button" (click)="setAddress()">Address</button>        <button type="button" (click)="setFirstname()">First Name</button>      </div>      <div>        <button type="button" (click)="withoutOnlySelf()">Without Only Self</button>        <button type="button" (click)="withOnlySelf()">With Only Self</button>      </div>      <div>        <button type="button" (click)="withouEmitEvent()">Without EmitEvent</button>        <button type="button" (click)="withEmitEvent()">With EmitEvent</button>      </div>      </form>  </div>    <div style="float: right; width:50%;">      <h3>Form Status</h3>    <b>status : </b>{{reactiveForm.status}}    <b>valid : </b>{{reactiveForm.valid}}    <b>invalid : </b>{{reactiveForm.invalid}}    <b>touched : </b>{{reactiveForm.touched}}    <b>untouched : </b>{{reactiveForm.untouched}}    <b>pristine : </b>{{reactiveForm.pristine}}    <b>dirty : </b>{{reactiveForm.dirty}}    <b>disabled : </b>{{reactiveForm.disabled}}    <b>enabled : </b>{{reactiveForm.enabled}}        <h3>Form Value</h3>    {{reactiveForm.value |json}}    </div> |

app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | <h3>Angular ValueChanges Example</h3>    <ul>    <li>      <a [routerLink]="['/template']" routerLinkActive="router-link-active" >Template</a>    </li>    <li>      <a [routerLink]="['/reactive']" routerLinkActive="router-link-active" >Reactive</a>    </li>  </ul>    <router-outlet></router-outlet> |

app.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import { Component} from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {  } |

app.module.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule, ReactiveFormsModule } from '@angular/forms';    import { AppRoutingModule } from './app-routing.module';  import { AppComponent } from './app.component';  import { TemplateComponent } from './template-component';  import { ReactiveComponent } from './reactive.component';    @NgModule({    declarations: [      AppComponent,TemplateComponent,ReactiveComponent    ],    imports: [      BrowserModule,      AppRoutingModule,      FormsModule,      ReactiveFormsModule    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

ValueChanges in Template Driven Forms

ValueChanges event can also be used in the [template-driven forms](https://www.tektutorialshub.com/angular/angular-template-driven-forms/). All you need to do is to get the reference to the Form Model in the component as shown below

|  |  |
| --- | --- |
| 1  2  3 | @ViewChild('templateForm',null) templateForm: NgForm; |

You can refer to the example code below

template-component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115 | import { Component, ViewChild, ElementRef, OnInit, OnDestroy } from '@angular/core';  import { NgForm } from '@angular/forms';      @Component({    templateUrl: './template.component.html',  })  export class TemplateComponent implements OnInit {      title = 'Template driven forms';      @ViewChild('templateForm',null) templateForm: NgForm;      contact: contact;      onSubmit() {      console.log(this.templateForm.value);    }      ngOnInit() {        setTimeout(() => {          this.templateForm.control.get("firstname").valueChanges.subscribe(selectedValue => {          console.log('firstname value changed')          console.log(selectedValue)          console.log(this.templateForm.control.get("firstname").value)          console.log(this.templateForm.control.value)            setTimeout(() => {            console.log(this.templateForm.control.value)          })        })          this.templateForm.control.get("address").valueChanges.subscribe(selectedValue => {          console.log('address changed')          console.log(selectedValue)        })          this.templateForm.valueChanges.subscribe(selectedValue => {          console.log('form value changed')          console.log(selectedValue)        })        });      }      setValue() {      let contact = {        firstname: "Rahul",        lastname: "Dravid",        address: {          city: "Bangalore",          street: "Brigade Road",          pincode: "600070"        }      };        this.templateForm.setValue(contact);    }      setAddress() {      let address= {        city: "Bangalore",        street: "Brigade Road",        pincode: "600070"      };        this.templateForm.control.get("address").setValue(address);      };      setFirstname() {      this.templateForm.control.get("firstname").setValue("Saurav")    }        withoutOnlySelf() {      this.templateForm.control.get("firstname").setValue("");    }    withOnlySelf() {      this.templateForm.control.get("firstname").setValue("", { onlySelf: true });    }      withouEmitEvent() {      this.templateForm.control.get("firstname").setValue("Sachin");    }    withEmitEvent() {      this.templateForm.control.get("firstname").setValue("", { emitEvent: false });    }      reset() {      this.templateForm.reset();    }    }      export class contact {    firstname:string;    lastname:string;    gender:string;    email:string;    isMarried:boolean;    country:string;    address: {      city:string;      street:string;      pincode:string;    }  } |

template-component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63 | <h3>{{title}}</h3>    <div style="float: left; width:50%;">    <form #templateForm="ngForm" (ngSubmit)="onSubmit(templateForm)">        <p>        <label for="firstname">First Name </label>        <input type="text" id="firstname" name="firstname" #fname="ngModel" ngModel>        </p>      <p>        <label for="lastname">Last Name </label>        <input type="text" id="lastname" name="lastname" ngModel>      </p>        <div ngModelGroup="address">          <p>          <label for="city">City</label>          <input type="text" id="city" name="city" ngModel>          <label for="street">Street</label>          <input type="text" id="street" name="street" ngModel>          <label for="pincode">Pin Code</label>          <input type="text" id="pincode" name="pincode" ngModel>        </p>        </div>        <button>Submit</button>      <div>        <button type="button" (click)="setValue()">SetValue</button>        <button type="button" (click)="setAddress()">Address</button>        <button type="button" (click)="setFirstname()">First Name</button>      </div>      <div>        <button type="button" (click)="withoutOnlySelf()">Without Only Self</button>        <button type="button" (click)="withOnlySelf()">With Only Self</button>      </div>      <div>        <button type="button" (click)="withouEmitEvent()">Without EmitEvent</button>        <button type="button" (click)="withEmitEvent()">With EmitEvent</button>      </div>      </form>  </div>    <div style="float: right; width:50%;">    <h3>Form Status</h3>    <b>status : </b>{{templateForm.status}}    <b>valid : </b>{{templateForm.valid}}    <b>invalid : </b>{{templateForm.invalid}}    <b>touched : </b>{{templateForm.touched}}    <b>untouched : </b>{{templateForm.untouched}}    <b>pristine : </b>{{templateForm.pristine}}    <b>dirty : </b>{{templateForm.dirty}}    <b>disabled : </b>{{templateForm.disabled}}    <b>enabled : </b>{{templateForm.enabled}}      <h3>Form Value</h3>    {{templateForm.value | json }}  </div>> |

Summary

In this tutorial, we learned how to make use of ValueChanges in Angular Forms. The ValueChanges event is fired whenever the value of the FormControl, FormGroup or FormArray changes. It is an observable and we can subscribe to it. We can then use it to validate the forms. update the computed fields, etc. The ValueChanges event does not fire depending on how we set emitEvent or onlySelf, when updating the value and validity of the form controls.

What is FormControl

Consider a simple text input box

|  |  |
| --- | --- |
| 1  2  3 | First Name : <input type="text" name="firstname" /> |

As a developer, you would like to know the current value in the text box. You would also be like to know if the value is valid or not.. If the user has changed the value(dirty) or is it unchanged. You would like to be notified when the user changes its value.

The FormControl is an object that encapsulates all the information related to the single input element. It Tracks the value and validation status of each of these control.

The FormControl is just a class. A FormControl is created for each form field. We can refer to them in our component class and inspect its properties and methods

We can use FormControl to set the value of the Form field. Find the status of form field like (valid/invalid, pristine/dirty, touched/untouched ), etc. You can add validation rules to it.

Using FormControl

The Angular has two approaches to building the [Angular Forms](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/). One is [Template-driven](https://www.tektutorialshub.com/angular/angular-template-driven-forms/) and the other one is [Reactive Forms](https://www.tektutorialshub.com/angular/angular-reactive-forms/).

To use the Angular forms, First, we need to import the FormsModule (for template-driven forms) & ReactiveFormsModule ( for Reactive Forms) from the @angular/forms in your route module.

|  |  |
| --- | --- |
| 1  2  3 | import { FormsModule, ReactiveFormsModule } from '@angular/forms'; |

Also, add it to the imports metadata.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | imports: [     BrowserModule,     AppRoutingModule,     FormsModule,     ReactiveFormsModule  ], |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Reactive Forms

In [Reactive Forms](https://www.tektutorialshub.com/angular/angular-reactive-forms/) approach, It is our responsibility to build the Model using FormGroup, FormControl and FormArray.

To use FormControl, first, we need to import the FormControl from the @angular/forms

|  |  |
| --- | --- |
| 1  2  3 | import { FormGroup, FormControl, Validators } from '@angular/forms' |

Then create the top-level FormGroup. The first argument to FormGroup is the collection of FormControl. They are added using the FormControl method as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | reactiveForm = new FormGroup({    firstname: new FormControl('',[Validators.required]),    lastname: new FormControl(),    email: new FormControl(),  }) |

Or you can make use of the [FormBuilder](https://www.tektutorialshub.com/angular/angular-formbuilder-in-reactive-forms/) API

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | this.reactiveForm = this.formBuilder.group({    firstname: ['',[Validators.required]],    lastname: [''],    email: [''],  }); |

Bind the form element with the template using the **formControlName**directive as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | <form [formGroup]="reactiveForm" (ngSubmit)="onSubmit()" novalidate>      <p>      <label for="firstname">First Name </label>      <input type="text" id="firstname" name="firstname" formControlName="firstname">    </p>      <p>      <label for="lastname">Last Name </label>      <input type="text" id="lastname" name="lastname" formControlName="lastname">    </p>      <p>      <label for="email">Email </label>      <input type="text" id="email" name="email" formControlName="email">    </p>      <p>      <button type="submit">Submit</button>    </p>    </form> |

Template-driven forms

In[template-driven forms](https://www.tektutorialshub.com/angular/angular-template-driven-forms/), the FormControl is defined in the Template. The <Form> directive creates the top-level FormGroup. We use the ngModel directive on each Form element, which automatically creates the FormControl instance.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | <form #templateForm="ngForm" (ngSubmit)="onSubmit(templateForm)" novalidate>      <p>      <label for="firstname">First Name</label>      <input type="text" name="firstname" ngModel>    </p>      <p>      <label for="lastname">Last Name</label>      <input type="text" name="lastname" ngModel>    </p>      <p>      <label for="email">Email </label>      <input type="text" id="email" name="email" ngModel>    </p>      <p>      <button type="submit">Submit</button>    </p>    </form> |

Use the viewChild to get the reference to the FormModel in the Component class. The control property of the NgForm returns the top-level formgroup

|  |  |
| --- | --- |
| 1  2  3 | @ViewChild('templateForm',null) templateForm: NgForm; |

Setting the value

setValue()

abstract setValue(value: any, options?: Object): void

We use setValue or patchValue method of the FormControl to set a new value for the form control. There is no difference between setValue and patchValue at the FormControl level.

|  |  |
| --- | --- |
| 1  2  3  4  5 | setEmail() {    this.reactiveForm.get("email").setValue("sachin.tendulakar@gmail.com");  }; |

|  |  |
| --- | --- |
| 1  2  3  4  5 | setEmail() {    this.templateForm.control.get("email").setValue("sachin.tendulkar@gmail.com");  }; |

patchValue()

abstract patchValue(value: any, options?: Object): void

|  |  |
| --- | --- |
| 1  2  3  4  5 | setEmail() {    this.reactiveForm.get("email").setValue("sachin.tendulakar@gmail.com");  }; |

|  |  |
| --- | --- |
| 1  2  3  4  5 | setEmail() {    this.templateForm.control.get("email").setValue("sachin.tendulkar@gmail.com");  }; |

**Must Read**: [setValue & patchValue in Angular forms](https://www.tektutorialshub.com/angular/setvalue-patchvalue-in-angular/)

Two-way binding

The two-way data binding is the preferred way to to keep the component model in sync with the FormModel in Template-driven forms.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <p>      <label for="firstname">First Name </label>      <input type="text" id="firstname" name="firstname" [(ngModel)]="contact.firstname">    </p> |

[Using two-way data binding in Reactive forms is deprecated since the Angular 7](https://angular.io/api/forms/FormControlName#use-with-ngmodel)

Finding the Value

value

value: any

The value returns the current value of FormControl It is Readonly. To set the value of the control either use the setValue or patchValue method

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //reactive forms  this.reactiveForm.get("firstname").value    //template driven forms  this.templateForm.control.get("firstname").value |

valueChanges

valueChanges: Observable<any>

The angular emits the valueChanges event whenever the value of the control changes. The value may change when the user updates the element in the UI or programmatically through the setValue/patchValue method. We can subscribe to it as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //reactive Forms    this.fNameChange = this.reactiveForm.get("firstname").valueChanges.subscribe(x => {     console.log(x);  }) |

Similarly in template-driven forms.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | setTimeout(() => {     this.fNameChange = this.templateForm.control.get("firstname").valueChanges.subscribe(x => {       console.log(x);     })  }); |

**Must Read**: [ValueChanges in Angular](https://www.tektutorialshub.com/angular/valuechanges-in-angular-forms/)

Control Status

The FormControl tracks the validation status of the HTML Element to which it is bound. The following is the list of status-related properties

status

status: string

The Angular runs validation checks, whenever the value of a form control changes. Based on the result of the validation, the control can have four possible states.

**VALID:** The FormControl has passed all validation checks.  
**INVALID:** This control has failed at least one validation check.  
**PENDING:** This control is in the midst of conducting a validation check.  
**DISABLED:** This control is exempt from validation checks

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //reactive forms  this.reactiveForm.get("firstname").status    //template driven forms  this.templateForm.control.get("firstname").status |

valid

valid: boolean

A control is valid when it has passed all the validation checks and is not disabled.

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.get("firstname").valid |

invalid

invalid: boolean

A control is invalid when it has failed one of the validation checks and is not disabled

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.get("firstname").invalid |

pending

pending: boolean

A control is pending when it is in the midst of conducting a validation check.

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.get("firstname").pending |

disabled

disabled: boolean

Control is disabled when its status is DISABLED.

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.get("firstname").disabled |

enabled

enabled: boolean

Control is enabled as long as the status is not DISABLED.

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.get("firstname").disabled |

pristine

pristine: boolean

Control is pristine if the user has not yet changed the value in the UI.

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.get("firstname").pristine |

dirty

dirty: boolean

Control is dirty if the user has changed the value in the UI.

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.get("firstname").dirty |

touched

touched: boolean

True if the control is marked as touched. A control is marked touched once the user has triggered a blur event on it.

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.get("firstname").touched |

untouched

untouched: boolean

True if the control has not been marked as touched. A control is untouched if the user has not yet triggered a blur event on it.

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.get("firstname").untouched |

Changing the Status

We can also change the status of the control programmatically by using the following methods.

When we change the status of a control programmatically or via UI, the validity & value of the parent control is also calculated and updated. There may arise circumstances when you do not want that to happen. In such circumstances, you can make use of the onlySelf:true to ensure that the parent control is not checked.

markAsTouched

This method will mark the control as touched.

markAsTouched(opts: { onlySelf?: boolean; } = {}): void

onlySelf if true then only this control is marked. If false it will also mark all its direct ancestors also as touched. The default is false.

|  |  |
| --- | --- |
| 1  2  3  4 | this.reactiveForm.get("firstname").markAsTouched()  this.reactiveForm.get("firstname").markAsTouched({ onlySelf:true; }) |

markAllAsTouched

markAllAsTouched(): void

Marks the control and all its descendant controls as touched.

markAsUntouched

markAsUntouched(opts: { onlySelf?: boolean; } = {}): void

Marks the control as untouched.

onlySelf if true only this control is marked as untouched. When false or not supplied, mark all direct ancestors as untouched. The default is false.

markAsDirty

markAsDirty(opts: { onlySelf?: boolean; } = {}): void

Marks the control as dirty. A control becomes dirty when the control’s value is changed through the UI.

onlySelf if true, only this control is marked as dirty else all the direct ancestors are marked as dirty. The default is false.

markAsPristine

markAsPristine(opts: { onlySelf?: boolean; } = {}): void

Marks the control as pristine.

onlySelf if true, only this control is marked as pristine else all the direct ancestors are marked as pristine. The default is false.

markAsPending

markAsPending(opts: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

Marks the control as pending. We mark it as pending when the control is in the midst of conducting a validation check.

onlySelf: When true, mark only this control. When false or not supplied, mark all direct ancestors. The default is false.

emitEvent: When true or not supplied (the default), the statusChanges observable emits an event with the latest status the control is marked pending. When false, no events are emitted.

disable

disable(opts: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

Disables the control. This means the control is exempt from validation checks and excluded from the aggregate value of any parent. Its status is DISABLED.

onlySelf: When true, mark only this control. When false or not supplied, mark all direct ancestors. Default is false..

emitEvent: When true or not supplied (the default), both the statusChanges and valueChanges observables emit events with the latest status and value when the control is disabled. When false, no events are emitted.

enable

enable(opts: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

Enables control. This means the control is included in validation checks and the aggregate value of its parent. Its status recalculates based on its value and its validators.

onlySelf: When true, mark only this control. When false or not supplied, mark all direct ancestors. The default is false.

emitEvent: When true or not supplied (the default), both the statusChanges and valueChanges observables emit events with the latest status and value when the control is enabled. When false, no events are emitted.

Status Change Event

statusChanges

statusChanges: Observable<any>

We can subscribe to the status changes event by subscribing it to the statusChanges as shown below. The event is fired whenever the validation status of the control is calculated.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | //Reactive Forms  this.reactiveForm.get("firstname").statusChanges.subscribe(x => {    console.log(x);  })    //Template Driven Forms  this.templateForm.control.get("firstname").statusChanges.subscribe(x => {     console.log(x);  }) |

**Must Read**: [StatusChanges in Angular](https://www.tektutorialshub.com/angular/statuschanges-in-angular-forms/)

Validation

The way we add validators depends on whether we use the Template-driven forms or reactive forms.

In Reactive forms, the validators are added while declaring the controls

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | reactiveForm = new FormGroup({    firstname: new FormControl('',[Validators.required]),    lastname: new FormControl(),    email: new FormControl(),  }) |

While in the template-driven forms in the template

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <p>    <label for="firstname">First Name </label>    <input type="text" id="firstname" name="firstname" ngModel required >  </p> |

updateValueAndValidity()

updateValueAndValidity(opts: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

The updateValueAndValidity forces the form to perform validation. This is useful when you add/remove validators dynamically using setValidators, RemoveValidators etc

onlySelf: When true, only update this control. When false or not supplied, update all direct ancestors. Default is false..

emitEvent: When true or not supplied (the default), both the statusChanges and valueChanges observables emit events with the latest status and value when the control is updated. When false, no events are emitted.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //reactive forms  this.reactiveForm.get("firstname").updateValueAndValidity();    //template driven forms  this.templateForm.control.get("firstname").updateValueAndValidity(); |

setValidators() / setAsyncValidators()

Programmatically adds the sync or async validators. This method will remove all the previously added sync or async validators.

setValidators(newValidator: ValidatorFn | ValidatorFn[]): void  
setAsyncValidators(newValidator: AsyncValidatorFn | AsyncValidatorFn[]): void

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //Reactive Form  setValidator() {    this.reactiveForm.get("firstname").setValidators([Validators.required, Validators.minLength(5)]);    this.reactiveForm.get("firstname").updateValueAndValidity();  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //Template driven forms  setValidator() {    this.templateForm.control.get("firstname").setValidators([Validators.required, Validators.minLength(5)]);    this.templateForm.control.get("firstname").updateValueAndValidity();  } |

clearValidators() / clearAsyncValidators()

clearValidators(): void  
clearAsyncValidators(): void

clearValidators & clearAsyncValidators clears all validators.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //reactive forms  clearValidation() {     this.reactiveForm.get("firstname").clearValidators();     this.reactiveForm.get("firstname").updateValueAndValidity();  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //template driven forms  clearValidation() {    this.templateForm.control.get("firstname").clearValidators();    this.templateForm.control.get("firstname").updateValueAndValidity();  } |

errors()

errors: ValidationErrors | null

An object containing any errors generated by failing validation, or null if there are no errors.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | getErrors() {      const controlErrors: ValidationErrors = this.reactiveForm.get("firstname").errors;    if (controlErrors) {      Object.keys(controlErrors).forEach(keyError => {        console.log("firtname "+ ' '+keyError);      });    }  } |

setErrors()

setErrors(errors: ValidationErrors, opts: { emitEvent?: boolean; } = {}): void

|  |  |
| --- | --- |
| 1  2  3  4  5 | setErrors() {      this.reactiveForm.get("firstname").setErrors( {customerror:'custom error'});  } |

getError()

getError(errorCode: string, path?: string | (string | number)[]): any

Reports error data for the control with the given path.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | this.reactiveForm.getError("firstname")    //  this.reactiveForm.getError("address.pincode");  this.reactiveForm.getError(["address","pincode"]); |

hasError

hasError(errorCode: string, path?: string | (string | number)[]): boolean

Reports whether the control with the given path has the error specified.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | this.reactiveForm.hasError("firstname")    //  this.reactiveForm.hasError("address.pincode");  this.reactiveForm.hasError(["address","pincode"]); |

Reset

abstract reset(value?: any, options?: Object): void

Resets the control. We can also pass the default value.

|  |  |
| --- | --- |
| 1  2  3  4 | this.reactiveForm.get("firstname").reset('');  this.reactiveForm.get("firstname").reset('test'); |

Summary

In this tutorial, we learned what is FormControl is and looked at the various methods & properties that are available.

What is FormGroup

Consider a simple HTML form

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <form>    First Name : <input type="text" name="firstname" />    Last Name  : <input type="text" name="lastname" />    Email      : <input type="text" name="email" />  </form> |

We create a FormControl for each of these input fields. It tracks the value & validity of these elements. All of the above input fields are represented as the separate FormControl. If we wanted to check the validity of our form, we have to check the validity of each and every FormControl for validity. Imagine a form having a large no of fields. It is cumbersome to loop over large no of FormControls and check for validity

The FormGroup solve’s this issue by providing a wrapper around a collection of FormControls It encapsulates all the information related to a group of form elements. It Tracks the value and validation status of each of these control. We can use it to check the validity of the elements. set its values & listen for change events, add and run validations on the group, etc

The FormGroup is just a class. We create a FormGroup to organize and manage the related elements. For Example form elements like address, city.state, pin code etc can be grouped together as a single FormGroup. It makes it easier to manage them. A FormGroup aggregates the values of each child FormControl into one object, with each control name as the key. It calculates its status by reducing the status values of its children. For example, if one of the controls in a group is invalid, the entire group becomes invalid.

Using FormGroup

Every Angular Form must have at least one top-level FormGroup. It tracks the value & validity of the entire form.

The Angular has two approaches to building the [Angular Forms](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/). One is [Template-driven](https://www.tektutorialshub.com/angular/angular-template-driven-forms/) and the other one is [Reactive Forms](https://www.tektutorialshub.com/angular/angular-reactive-forms/).

To use the Angular forms, First, we need to import the FormsModule (for template-driven forms) & ReactiveFormsModule ( for Reactive Forms) from the @angular/forms in your route module.

|  |  |
| --- | --- |
| 1  2  3 | import { FormsModule, ReactiveFormsModule } from '@angular/forms'; |

Also, add it to the imports metadata

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | imports: [      BrowserModule,      AppRoutingModule,      FormsModule,      ReactiveFormsModule    ], |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Reactive Forms

In Reactive forms, we create the form model in the component class. First, we need to import the FormGroup, FormControl, Validators

|  |  |
| --- | --- |
| 1  2  3 | import { FormGroup, FormControl, Validators } from '@angular/forms' |

When instantiating a FormGroup, pass in a collection of child controls as the first argument. The key for each child registers the name for the control

The following form model has two Form Groups. One is the top-level Form group, which we have named as reactiveForm. The other one is nested Form Group, which we have named it as address.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | reactiveForm = new FormGroup({    firstname: new FormControl('', [Validators.required]),    lastname: new FormControl(''),    email: new FormControl(''),    address: new FormGroup({      address: new FormControl(''),      city: new FormControl(''),      state: new FormControl(''),    })  }) |

And in the Template, we use formGroup, formControlName and formGroupName directive to bind the Form to the template.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | <form [formGroup]="reactiveForm" (ngSubmit)="onSubmit()" novalidate>        <p>        <label for="firstname">First Name </label>        <input type="text" id="firstname" name="firstname" formControlName="firstname">      </p>        <p>        <label for="lastname">Last Name </label>        <input type="text" id="lastname" name="lastname" formControlName="lastname">      </p>        <p>        <label for="email">Email </label>        <input type="text" id="email" name="email" formControlName="email">      </p>        <div formGroupName="address">          <p>          <label for="address">Address</label>          <input type="text" class="form-control" name="address" formControlName="address">        </p>          <p>          <label for="city">City</label>          <input type="text" class="form-control" name="city" formControlName="city">        </p>          <p>          <label for="state">State</label>          <input type="text" class="form-control" name="state" formControlName="state">        </p>        </div>        <button>Submit</button>      </form> |

Template Driven forms

In Template-driven forms. the model is built in the template first. The top-level form is bound to ngForm directive, which we have named as templateForm. We add ngModel directive to each form element to create Form Controls. The name attribute will become the name of the Form Control. The ngModelGroup directive is used to create the nested Form Group.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | <form #templateForm="ngForm" (ngSubmit)="onSubmit(templateForm)">        <p>        <label for="firstname">First Name </label>        <input type="text" id="firstname" name="firstname" ngModel required>      </p>      <p>        <label for="lastname">Last Name </label>        <input type="text" id="lastname" name="lastname" ngModel>      </p>        <p>        <label for="email">Email </label>        <input type="text" id="email" name="email" ngModel>      </p>          <div ngModelGroup="address">          <p>          <label for="address">Address</label>          <input type="text" class="form-control" name="address" ngModel>        </p>          <p>          <label for="city">City</label>          <input type="text" class="form-control" name="city" ngModel>        </p>          <p>          <label for="state">State</label>          <input type="text" class="form-control" name="state" ngModel>        </p>      </div>        <p>        <button type="submit">Submit</button>      </p>      <div>      </div>      </form> |

We can get the reference to the top-level form group in component class using the ViewChild as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | import { NgForm, Validators } from '@angular/forms';    export class TemplateComponent implements OnInit {     @ViewChild('templateForm', null) templateForm: NgForm;       ....  } |

Setting Value

We use [setValue](https://www.tektutorialshub.com/angular/setvalue-patchvalue-in-angular/) or patchValue method of the FormGroup to set a new value for the entire FormGroup.

SetValue

Sets the value of the FormGroup. It accepts an object that matches the structure of the group, with control names as keys. The structure must match exactly, otherwise, it will result in an error.

setValue(value: { [key: string]: any; }, options: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | setValue() {      this.reactiveForm.setValue({      firstname: "Sachin",      lastname: "Tendulakr",      email: "sachin@gmail.com",      address: {        address: "19-A, Perry Cross Road, Bandra (West)",        city: "Mumbai",        state: "Maharatsra",      }    })  } |

You can also update the nested FormGroup separately,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | setAddress() {    this.reactiveForm.get("address").setValue({      address: "19-A, Perry Cross Road, Bandra (West)",      city: "Mumbai",      state: "Maharatsra",    })  } |

patchValue

Patches the value of the [FormGroup](https://angular.io/api/forms/FormGroup). It accepts an object with control names as keys and does its best to match the values to the correct controls in the group.

patchValue(value: { [key: string]: any; }, options: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | patchValue() {      this.reactiveForm.patchValue({      email: "sachin@gmail.com",      address: {        state: "Maharatsra",      }    })  } |

We can Both setValue & patchValue

onlySelf: When true, each change only affects this control and not its parent. The default is true.

emitEvent: When true or not supplied (the default), both the statusChanges and valueChanges observables emit events with the latest status and value when the control value is updated. When false, no events are emitted. The configuration options are passed to the updateValueAndValidity method.

Finding the Value

value

The value returns the object with a key-value pair for each member of the Form Group. It is Readonly. To Set [Value either setValue or patchValue](https://www.tektutorialshub.com/angular/setvalue-patchvalue-in-angular/)

value: any

|  |  |
| --- | --- |
| 1  2  3  4  5 | onSubmit() {    console.log(this.reactiveForm.value);  } |

valueChanges

valueChanges: Observable<any>

The angular emits the [valueChanges](https://www.tektutorialshub.com/angular/valuechanges-in-angular-forms/) event whenever the value of any of the controls in the Form Group changes. The value may change when user updates the element in the UI or programmatically through the setValue/patchValue method. We can subscribe to it as shown below

In the example below, the first valuesChanges are fired, when any of the control is changed. While the second valuesChanges event is raised only when the controls under the address form group are changed

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | ngOnInit() {    this.reactiveForm.valueChanges.subscribe(x => {      console.log(x);    })    this.reactiveForm.get("address").valueChanges.subscribe(x => {      console.log(x);    })  } |

Adding Controls Dynamically to Form Group

We usually add controls, while initializing the FormGroup.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | reactiveForm = new FormGroup({    firstname: new FormControl('', [Validators.required]),    } |

The Forms API also allows add controls dynamically

addControl()

Adds a control to the FormGroup and also updates validity & validation status. If the control already exists, then ignores it

addControl(name: string, control: AbstractControl): void

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | addControl() {      this.middleName = new FormControl('', [Validators.required]);      this.reactiveForm.addControl("middleName",this.middleName);    } |

registerControl()

Adds control to this FormGroup but does not update the validity & validation status. If the control already exists, then ignores it

registerControl(name: string, control: AbstractControl): AbstractControl

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | registerControl() {    this.middleName = new FormControl('', [Validators.required]);    this.reactiveForm.addControl("middleName",this.middleName);  } |

removeControl()

This method will remove the control with the provided name from the FormGroup.

removeControl(name: string): void

|  |  |
| --- | --- |
| 1  2  3  4  5 | remodeControl() {    this.reactiveForm.removeControl("middleName");  } |

setControl()

Replaces the control with the provided name with the new control.

setControl(name: string, control: AbstractControl): void

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | setControl() {    this.middleName = new FormControl('test', [Validators.required]);    this.reactiveForm.setControl("middleName",this.middleName);  } |

contains()

Check whether the control with the provided name exists or not..

contains(controlName: string): boolean

|  |  |
| --- | --- |
| 1  2  3  4  5 | containsControl() {    console.log(this.reactiveForm.contains("middleName"));  } |

Control Status

The FormGroup tracks the validation status of all the FormControls, which is part of the FormGroup. That also includes the status of nested FormGroup or FormArray. If any of the control becomes invalid, then the entire FormGroup becomes invalid.

The following is the list of status-related properties

status

status: string

The Angular runs validation checks, whenever the value of a form control changes. Based on the result of the validation, the FormGroup can have four possible states.

**VALID:** All the controls of the FormGroup has passed all validation checks.  
**INVALID:** At least one of the control has failed at least one validation check.  
**PENDING:** This Group is in the midst of conducting a validation check.  
**DISABLED:** This FormGroup is exempt from validation checks

|  |  |
| --- | --- |
| 1  2  3  4 | //reactive forms  this.reactiveForm.status |

valid

valid: boolean

A FormGroup is valid when it has passed all the validation checks and the FormGroup is not disabled

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.valid |

invalid

invalid: boolean

A FormGroup is invalid when one of its controls has failed a validation check or the entire FormGroup is disabled.

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.invalid |

pending

pending: boolean

A FormGroup is pending when it is in the midst of conducting a validation check.

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.pending |

disabled

disabled: boolean

A FormGroup is disabled when all of its controls are disabled.

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.disabled |

enabled

enabled: boolean

A FormGroup is enabled as long one of its control is enabled.

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.disabled |

pristine

pristine: boolean

A FormGroup is pristine if the user has not yet changed the value in the UI in any of the controls

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.pristine |

dirty

dirty: boolean

A FormGroup is dirty if the user has changed the value in the UI in any one of the control.

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.dirty |

touched

touched: boolean

True if the FomGroup is marked as touched. A FormGroup is marked as touched once the user has triggered a blur event on any one of the controls

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.touched |

untouched

untouched: boolean

True if the FormGroup has not been marked as touched. A FormGroup is untouched if the user has not yet triggered a blur event on any of its child controls

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.untouched |

Changing the Status

We can also change the status of the FormGroup by using the following method.

markAsTouched

The FormGroup is marked as touched if anyone of its control is marked as touched. The control is marked as touched once the user has triggered a blur event on it.

markAsTouched(opts: { onlySelf?: boolean; } = {}): void

onlySelf if true then only this control is marked. If false it will also mark all its direct ancestors also as touched. The default is false.

In the following example, the City is marked as touched. It will also mark both the address & reactiveFormGroup as touched.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markCityAsTouched() {      this.reactiveForm.get("address").get("city").markAsTouched();    } |

By Passing the onlySelf:true argument, you can ensure that only the city is marked as touched, while address & reactiveForm are not affected.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markCityAsTouched() {      this.reactiveForm.get("address").get("city").markAsTouched({onlySelf:true});    } |

The following code marks the address FormGroup as touched. while the child controls are not marked as touched. The parent FormGroup is marked as touched.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markAddressAsTouched() {      this.reactiveForm.get("address").markAsTouched();    } |

While onlySelf:true marks only the address group as touched, leaving the top-level FormGroup

|  |  |
| --- | --- |
| 1  2  3  4  5 | markAddressAsTouched() {      this.reactiveForm.get("address").markAsTouched({onlySelf:true});    } |

markAllAsTouched

Marks the control and all its descendant controls as touched.

markAllAsTouched(): void

The following example marks the address and all its controls i.e city, state & address as touched. The parent FormGroup stays as it is.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markAllAddressTouched() {    this.reactiveForm.get("address").markAllAsTouched();  } |

markAsUntouched

Marks the control as untouched.

markAsUntouched(opts: { onlySelf?: boolean; } = {}): void

onlySelf if true only this control is marked as untouched. When false or not supplied, mark all direct ancestors as untouched. The default is false.

The following code will mark the city as untouched. It will recalculate the untouched & touched status of the parent Group. If all the other controls are untouched then the parent FormGroup address is also marked as untouched.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markCityAsUnTouched() {    this.reactiveForm.get("address").get("city").markAsUntouched();  } |

By using the onlySelf:true you can ensure that only the city is marked as untouched, leaving the parent FormGroup as it is.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markCityAsUnTouched() {    this.reactiveForm.get("address").get("city").markAsUntouched({onlySelf:true});  } |

Similarly, you can mark the entire FormGroup as untouched. While this does not have any effect on the child controls, but it does recalculate the untouched status of the parent FormGroup. You can use the onlySelf:true ensure that it does not happen.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markAddressAsUnTouched() {    this.reactiveForm.get("address").markAsUntouched();  } |

markAsDirty

The FormGroup becomes dirty when any one of its control is marked as dirty. A control becomes dirty when the control’s value is changed through the UI. We can use the markAsDirty method to manipulate the dirty status.

markAsDirty(opts: { onlySelf?: boolean; } = {}): void

onlySelf if true, only this control is marked as dirty else all the direct ancestors are marked as dirty. The default is false.

The Following code marks the entire form as dirty. It does not change the status of any of the child controls.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markFormAsDirty() {    this.reactiveForm.markAsDirty();  } |

The following code marks the City as dirty. It will also change the Dirty status of Parent FormGroup.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markCityAsDirty() {    this.reactiveForm.get("address").get("city").markAsDirty();  } |

You can use the onlySelf:false to ensure that the parent FormGroup is not affected by our change.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markCityAsDirty() {      this.reactiveForm.get("address").get("city").markAsDirty({onlySelf:false});    } |

You can also make the entire FormGroup as dirty. It does not affect the child controls, but parent FormGroup is also marked as dirty. Unless you pass the {onlySelf:true} argument

|  |  |
| --- | --- |
| 1  2  3  4  5 | markAddressAsDirty() {    this.reactiveForm.get("address").markAsDirty({onlySelf:false});  } |

markAsPristine

The FormGroup becomes pristine when none of its controls values are changed via UI. The pristine is the opposite of dirty. We can use the markAsPrisitine method to manipulate the pristine status.

markAsPristine(opts: { onlySelf?: boolean; } = {}): void

onlySelf if true, only this control is marked as pristine else all the direct ancestors are marked as pristine. The default is false.

The following code marks the Form as Pristine. It will also mark all the child controls as Pristine

|  |  |
| --- | --- |
| 1  2  3  4  5 | markFormAsPristine() {    this.reactiveForm.markAsPristine();  } |

The following code marks the city as Pristine. It will also calculate the Pristine status of the Parent FormGroup. If all the other controls are pristine then the parent FormGroup becomes pristine.

|  |  |
| --- | --- |
| 1  2  3  4  5 | markCityAsPristine() {      this.reactiveForm.get("address").get("city").markAsPristine({onlySelf:false});    } |

You can make use of the onlySelf:true to ensure that the pristine status of the parent group is not calculated.

markAsPending

Marks the control as pending. We usually use this when we running our validation checks. Pending means the status of the control cannot be determined at this time.

markAsPending(opts: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

onlySelf: When true, mark only this control. When false or not supplied, mark all direct ancestors. The default is false.

emitEvent: When true or not supplied (the default), the statusChanges observable emits an event with the latest status the control is marked pending. When false, no events are emitted.

The following code marks the entire form as Pending. It does not change the status of child Controls.

|  |  |
| --- | --- |
| 1  2  3 | this.reactiveForm.markAsPending(); |

The following will mark the address FormGroup as Pending. It will also mark the Parent FormGroup as Pending also, which you can control using the onlySelf:true argument

|  |  |
| --- | --- |
| 1  2  3  4  5 | markAddressAsPendng() {    this.reactiveForm.get("address").markAsPending();  } |

This method also triggers the [statusChange](https://www.tektutorialshub.com/angular/statuschanges-in-angular-forms/) Event. You can make use of emitEvent:false argument, which will stop the statusChange event being triggered.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | markAddressAsPendng() {    this.reactiveForm.get("address").markAsPending({emitEvent:false});  } |

disable

Disables the control. This means the control is exempt from validation checks and excluded from the aggregate value of any parent. Its status is DISABLED.

disable(opts: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

onlySelf: When true, mark only this control. When false or not supplied, mark all direct ancestors. Default is false..

emitEvent: When true or not supplied (the default), both the statusChanges and valueChanges observables emit events with the latest status and value when the control is disabled. When false, no events are emitted.

The following code disables all the controls in the FormGroup.

|  |  |
| --- | --- |
| 1  2  3  4  5 | disableAll() {    this.reactiveForm.disable();  } |

If you disable all the controls individually, then the FormGroup is automatically disabled.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | disableAll() {    this.reactiveForm.get("firstname").disable();    this.reactiveForm.get("lastname").disable();    this.reactiveForm.get("email").disable();    this.reactiveForm.get("address").disable();  } |

enable

Enables control. This means the control is included in validation checks and the aggregate value of its parent. Its status recalculates based on its value and its validators.

enable(opts: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

onlySelf: When true, mark only this control. When false or not supplied, mark all direct ancestors. The default is false.

emitEvent: When true or not supplied (the default), both the statusChanges and valueChanges observables emit events with the latest status and value when the control is enabled. When false, no events are emitted.

The following command enables all the controls in the Group. Even the controls previously disabled are also enabled.

|  |  |
| --- | --- |
| 1  2  3  4  5 | enableAll() {     this.reactiveForm.enable();  } |

Enables only address FormGroup.

|  |  |
| --- | --- |
| 1  2  3  4  5 | enableAddress() {    this.reactiveForm.get("address").enable();  } |

Enable a Single Control

|  |  |
| --- | --- |
| 1  2  3  4  5 | enableFirstName() {    this.reactiveForm.get("firstname").enable();  } |

Status Change Event

statusChanges

statusChanges: Observable<any>

The statusChanges event is fired whenever the status of the form is calculated. We can subscribe to this event as shown below. We can subscribe it at the FormControl level or at the FormGroup level.

**Note that this event is fired whenever the status is calculated.**

In the example below, the first statusChanges is emitted, when the status of the top-level FormGroup is calculated. The second statusChange event is emitted, when the address FormGroup status is calculated.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | this.reactiveForm.statusChanges.subscribe(x => {    console.log(x);  })    this.reactiveForm.get("address").statusChanges.subscribe(x => {     console.log(x);  }) |

Validation

The validators can be added to FormControl, FormGroup or to the FormArray.

updateValueAndValidity()

The updateValueAndValidity forces the form to perform validation. When applied to the FormGroup, it will calculate the validity of all the child controls, including nested form groups & form arrays This is useful when you add/remove validators dynamically using setValidators, RemoveValidators etc

updateValueAndValidity(opts: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

onlySelf: When true, only update this control. When false or not supplied, update all direct ancestors. Default is false..

emitEvent: When true or not supplied (the default), both the statusChanges and valueChanges observables emit events with the latest status and value when the control is updated. When false, no events are emitted.

|  |  |
| --- | --- |
| 1  2  3  4 | this.reactiveForm.updateValueAndValidity();  this.reactiveForm.get("address").updateValueAndValidity(); |

setValidators() / setAsyncValidators()

Programmatically adds the sync or async validators. This method will remove all the previously added sync or async validators.

setValidators(newValidator: ValidatorFn | ValidatorFn[]): void  
setAsyncValidators(newValidator: AsyncValidatorFn | AsyncValidatorFn[]): void

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | setValidator() {    this.reactiveForm.get("address").setValidators([addressValidator]);    this.reactiveForm.get("address").updateValueAndValidity();  }      export const addressValidator = (control: AbstractControl): {[key: string]: boolean} => {    const city = control.get('city').value;    const state = control.get('state').value;    console.log(control.value);    if (city=="" && state=="") {      return { address:false };    }    return null;  }; |

clearValidators() / clearAsyncValidators()

clearValidators(): void  
clearAsyncValidators(): void

clearValidators & clearAsyncValidators clears all validators.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //reactive forms  clearValidation() {     this.reactiveForm.get("address").clearValidators();     this.reactiveForm.get("address").updateValueAndValidity();  } |

errors()

errors: ValidationErrors | null

An object containing any errors generated by failing validation, or null if there are no errors.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | getErrors() {      const controlErrors: ValidationErrors = this.reactiveForm.errors;    if (controlErrors) {      Object.keys(controlErrors).forEach(keyError => {        console.log("firtname "+ ' '+keyError);      });    }  } |

setErrors()

setErrors(errors: ValidationErrors, opts: { emitEvent?: boolean; } = {}): void

|  |  |
| --- | --- |
| 1  2  3  4  5 | setErrors() {    this.reactiveForm.setErrors( {customerror:'custom error'});  } |

getError()

getError(errorCode: string, path?: string | (string | number)[]): any

Reports error data for the control with the given path.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | this.reactiveForm.getError("firstname")    this.reactiveForm.getError("address.pincode");  this.reactiveForm.getError(["address","pincode"]); |

hasError

hasError(errorCode: string, path?: string | (string | number)[]): boolean

Reports whether the control with the given path has the error specified.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | this.reactiveForm.hasError("firstname")    //  this.reactiveForm.hasError("address.pincode");  this.reactiveForm.hasError(["address","pincode"]); |

Reset

abstract reset(value?: any, options?: Object): void

Resets the control. We can also pass the default value.

|  |  |
| --- | --- |
| 1  2  3  4  5 | this.reactiveForm.get("firstname").reset('');  this.reactiveForm.get("firstname").reset('test'); |

Summary

In this tutorial, we learned what is FormControl is and looked at the various methods & properties that are available.

What is FormArray

The FormArray is a way to manage the collection of Form Controls in Angular. The controls can be a [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/), [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/), or another FormArray.

We can group Form Controls in Angular forms in two ways. One is using the [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) and the other one is FormArray. The difference is how they implement it. In [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) controls becomes a property of the [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/). Each control is represented as key-value pair. While in FormArray, the controls become part of an array

Because it is implemented as an Array, it makes it easier dynamically add controls.

FormArray Example

Let us build a simple app, which allows us to add the new skill of a person dynamically.

Import FormArray

To use FormArray, First, you need to import the FormArray from the [Angular Forms](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/) Module.

|  |  |
| --- | --- |
| 1  2  3 | import { FormGroup, FormControl,FormArray, FormBuilder } from '@angular/forms' |

Build a Form Model

Build a form model skillsForm using the [FormBuilder](https://www.tektutorialshub.com/angular/angular-formbuilder-in-reactive-forms/). Our Form has two fields. name of the person and his skills. Since the person can have more than one skill, we define skills as FormArray.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | skillsForm: FormGroup;      constructor(private fb:FormBuilder) {        this.skillsForm = this.fb.group({        name: '',        skills: this.fb.array([]) ,      });      } |

Next, a getter method skills, which returns the skills FormArray from the skillsForm

|  |  |
| --- | --- |
| 1  2  3  4  5 | get skills() : FormArray {    return this.skillsForm.get("skills") as FormArray  } |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

The skill FormGroup

We need to capture two fields under each skill. Name of the skill & years of exp. Hence we create a [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/)with two fields. The method newSkill creates a new [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/)and returns it. Note that we won’t be able to assign a name to Form Group.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | newSkill(): FormGroup {     return this.fb.group({       skill: '',       exp: '',     })  } |

Dynamically adding skill

Now, we need to add a new skill to the skills FormArray. Since it is an array we can use the push method to add the new skill using the the newSkill method. Note that newSkill() method returns a FormGroup. The name of the FormGroup is its Index in the FormArray.

|  |  |
| --- | --- |
| 1  2  3  4  5 | addSkills() {     this.skills.push(this.newSkill());  } |

Dynamically Removing Skill

Use the removeAt method to remove the element from the skills FromArray.

|  |  |
| --- | --- |
| 1  2  3  4  5 | removeSkill(i:number) {    this.skills.removeAt(i);  } |

Submit

|  |  |
| --- | --- |
| 1  2  3  4  5 | onSubmit() {     console.log(this.skillsForm.value);  } |

Template

Now, it is time to build the Template. Use the [formGroup]="skillsForm" to bind the form to the skillsForm model. The formControlName="name" directive binds the name input element to name property of the skillsForm

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | <form [formGroup]="skillsForm" (ngSubmit)="onSubmit()">      <p>      <label for="name">Name </label>      <input type="text" id="name" name="name" formControlName="name">    </p>    <p>      <button type="submit">Submit</button>    </p>    </form> |

Binding FormArray to Template

We use the formArrayName directive to bind the skills form array to the div element. Now the div and anything inside the div element is bound to the skills form array.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div formArrayName="skills">    </div> |

Inside the div use ngFor to loop through each element of skills FormArray. let i=index will store the index value of the array in template local variable i. We will make use of it to remove the element from the skills array.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <div formArrayName="skills">    <div \*ngFor="let skill of skills().controls; let i=index">      </div>  </div> |

Each element under the skills is a FormGroup. We do not have a name to the FormGroup. The Index of the element is automatically assigned as the name for the element.

Hence we use the [formGroupName]="i" where i is the index of the FormArray to bind the FormGroup to the div element.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <div formArrayName="skills">    <div \*ngFor="let skill of skills().controls; let i=index">        <div [formGroupName]="i">          </div>    </div>  </div> |

Finally, we add the controls using the formControlName directive.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | Skills:    <div formArrayName="skills">      <div \*ngFor="let skill of skills().controls; let i=index">        <div [formGroupName]="i">          {{i}}          skill name :          <input type="text" formControlName="skill">          exp:          <input type="text" formControlName="exp">            <button (click)="removeSkill(i)">Remove</button>        </div>      </div>    </div> |

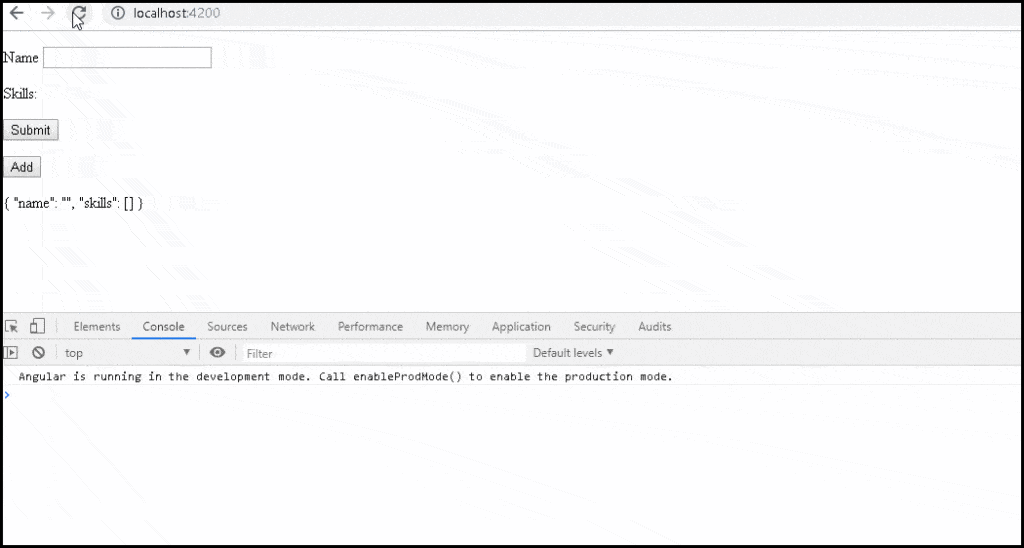
Also, pass the index i to removeSkill

|  |  |
| --- | --- |
| 1  2  3 | <button (click)="removeSkill(i)">Remove</button> |

Finally, call the addSkills method to add new skills.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <p>    <button type="button" (click)="addSkills()">Add</button>  </p> |

That’s it

Angular FormArray Example App Running

Source Code

app.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61 | import { Component, ViewChild, ElementRef } from '@angular/core';  import { FormGroup, FormControl,FormArray, FormBuilder } from '@angular/forms'      @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent  {      title = 'FormArray Example in Angular Reactive forms';      skillsForm: FormGroup;      constructor(private fb:FormBuilder) {        this.skillsForm = this.fb.group({        name: '',        skills: this.fb.array([]) ,      });      }      get skills() : FormArray {      return this.skillsForm.get("skills") as FormArray    }      newSkill(): FormGroup {      return this.fb.group({        skill: '',        exp: '',      })    }      addSkills() {      this.skills.push(this.newSkill());    }      removeSkill(i:number) {      this.skills.removeAt(i);    }      onSubmit() {      console.log(this.skillsForm.value);    }    }      export class country {    id: string;    name: string;      constructor(id: string, name: string) {      this.id = id;      this.name = name;    }  } |

app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | <form [formGroup]="skillsForm" (ngSubmit)="onSubmit()">      <p>      <label for="name">Name </label>      <input type="text" id="name" name="name" formControlName="name">    </p>        Skills:    <div formArrayName="skills">      <div \*ngFor="let skill of skills().controls; let i=index">        <div [formGroupName]="i">          {{i}}          skill name :          <input type="text" formControlName="skill">          exp:          <input type="text" formControlName="exp">            <button (click)="removeSkill(i)">Remove</button>        </div>      </div>    </div>      <p>      <button type="submit">Submit</button>    </p>    </form>      <p>    <button type="button" (click)="addSkills()">Add</button>  </p>    {{this.skillsForm.value | json}} |

Angular Nested Formarray

Create a new Angular Project. Open the app.module.ts and import the [ReactiveFormsModule](https://www.tektutorialshub.com/angular/angular-reactive-forms/)

|  |  |
| --- | --- |
| 1  2  3 | import { ReactiveFormsModule } from '@angular/forms'; |

Also, add it to the imports array of [NgModule](https://www.tektutorialshub.com/angular/angular-modules/) of the AppModule

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @NgModule({    imports: [BrowserModule, ReactiveFormsModule],    declarations: [AppComponent, HelloComponent],    bootstrap: [AppComponent]  })  export class AppModule {} |

You can refer to the completed Source Code from [StackBlitz](https://stackblitz.com/edit/angular-nested-formarray-dynamic-forms?file=src/app/app.component.ts)

Import FormArray

Go to the AppComponent. Import the FormArray from the [Angular Forms](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/) Module.

|  |  |
| --- | --- |
| 1  2  3 | import { FormGroup, FormArray, FormBuilder } from '@angular/forms' |

Build a Form Model

The First task is to build a Form Model empForm. It has only one property a FormArray of employees.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | empForm:FormGroup;    constructor(private fb:FormBuilder) {     this.empForm=this.fb.group({       employees: this.fb.array([]) ,     })  } |

Employee FormArray

Helper method, which returns the employees FormArray from the model empForm

|  |  |
| --- | --- |
| 1  2  3  4  5 | employees(): FormArray {    return this.empForm.get("employees") as FormArray  } |

The newEmployee method creates a new employee [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) and returns it. It has three properties. firstName, lastName and skills FormArray.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | newEmployee(): FormGroup {    return this.fb.group({      firstName: '',      lastName: '',      skills:this.fb.array([])    })  } |

Next, the method to add an employee. It uses the newEmployee method which returns the Employee FormGroup and ads it to employees array.

|  |  |
| --- | --- |
| 1  2  3  4  5 | addEmployee() {    this.employees().push(this.newEmployee());  } |

Method to remove the employee form the array. It needs the index position to remove it.

|  |  |
| --- | --- |
| 1  2  3  4  5 | removeEmployee(empIndex:number) {    this.employees().removeAt(empIndex);  } |

Skills FormArray

Under each employee, we have skills array. Hence create helper method which returns a skills array from the employee array. We need to pass the index position of the employee array as argument.

|  |  |
| --- | --- |
| 1  2  3  4  5 | employeeSkills(empIndex:number) : FormArray {    return this.employees().at(empIndex).get("skills") as FormArray  } |

newSkill method returns a skill [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/). It has two fields. Name of the skill and years of exp

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | newSkill(): FormGroup {    return this.fb.group({      skill: '',      exp: '',    })  } |

addEmployeeSkill method the skill to employee.

|  |  |
| --- | --- |
| 1  2  3  4  5 | addEmployeeSkill(empIndex:number) {    this.employeeSkills(empIndex).push(this.newSkill());  } |

Finally, removeEmployeeSkill method, which removes the skill of an employee.

|  |  |
| --- | --- |
| 1  2  3  4  5 | removeEmployeeSkill(empIndex:number,skillIndex:number) {    this.employeeSkills(empIndex).removeAt(skillIndex);  } |

The onSubmit method accepts the Employee Forms. You can validate and update the database from here.

|  |  |
| --- | --- |
| 1  2  3  4  5 | onSubmit() {      console.log(this.empForm.value);    } |

Template

Create a <form> and bind it to empForm using [formgroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) directive

|  |  |
| --- | --- |
| 1  2  3 | <form [formGroup]="empForm" (ngSubmit)="onSubmit()"> |

Under empForm we have employees array. Bind it to the div element using formArrayName directive

|  |  |
| --- | --- |
| 1  2  3 | <div formArrayName="employees"> |

Next, loop through the controls under the employees using [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/). let empIndex=index will save the index position in the empIndex local variable.

|  |  |
| --- | --- |
| 1  2  3  4 | <div \*ngFor="let employee of employees().controls; let empIndex=index"> |

The index is used as the name of the control in a Form Array. Hence use the [formGroupName]="empIndex" to bind it to the FormGroup. The style exists to show a nice border around employee

|  |  |
| --- | --- |
| 1  2  3 | <div [formGroupName]="empIndex" style="border: 1px solid blue; padding: 10px; width: 600px; margin: 5px;"> |

Input element for employee’s firstName, lastName. Also, place a button removeEmployee(empIndex) to remove this employee from the array.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | {{empIndex}}          First Name :          <input type="text" formControlName="firstName">          Last Name:          <input type="text" formControlName="lastName">            <button (click)="removeEmployee(empIndex)">Remove</button> |

Bind the skills of the empoyee to a div using formArrayName directive

|  |  |
| --- | --- |
| 1  2  3 | <div formArrayName="skills"> |

[ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/), now loops through the skills array of the employee.

|  |  |
| --- | --- |
| 1  2  3 | <div \*ngFor="let skill of employeeSkills(empIndex).controls; let skillIndex=index"> |

input fields for skill and exp, Also button to remove the skill, which calls the removeEmployeeSkill(empIndex,skillIndex)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <div [formGroupName]="skillIndex">                {{skillIndex}}                Skill :                <input type="text" formControlName="skill">                Exp:                <input type="text" formControlName="exp">                  <button (click)="removeEmployeeSkill(empIndex,skillIndex)">Remove</button>                </div> |

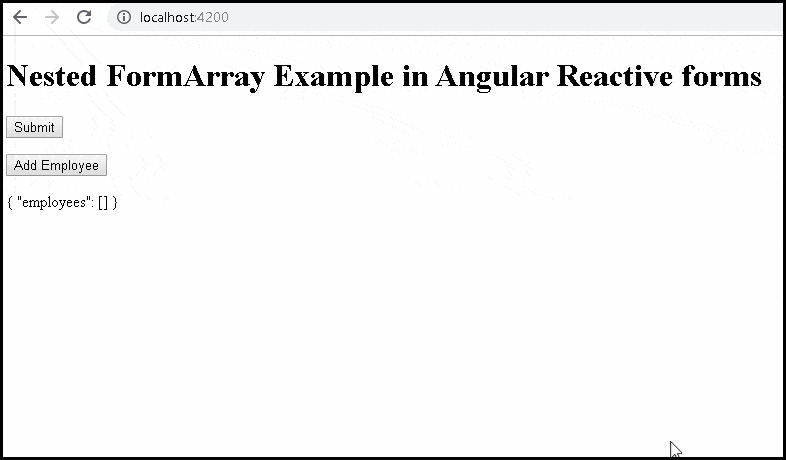
Finally a button the add the skill to the employee

|  |  |
| --- | --- |
| 1  2  3 | <button type="button" (click)="addEmployeeSkill(empIndex)">Add Skill</button> |

Finally a button the add the employee

|  |  |
| --- | --- |
| 1  2  3 | <button type="button" (click)="addEmployee()">Add Employee</button> |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Nested FormArray Example. Add Form Fields Dynamically

Updating the Nested forms With Initial Data

Setting the initial values in the nested Forms is a little tricky. You need to build the form dynamically and then use the PatchValue & SetValue methods to update the form.

[SetValue & PatchValue in FormArray Angular](https://www.tektutorialshub.com/angular/setvalue-patchvalue-in-formarray-angular/)

Source Code

You can view the source code from the [StackBlitz](https://stackblitz.com/edit/angular-nested-formarray-dynamic-forms?file=src/app/app.component.ts).

***app.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67 | import { Component, VERSION } from '@angular/core';  import { FormGroup, FormArray, FormBuilder } from '@angular/forms';    @Component({    selector: 'my-app',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    empForm: FormGroup;      constructor(private fb: FormBuilder) {}      ngOnInit() {      this.empForm = this.fb.group({        employees: this.fb.array([])      });    }      employees(): FormArray {      return this.empForm.get('employees') as FormArray;    }      newEmployee(): FormGroup {      return this.fb.group({        firstName: '',        lastName: '',        skills: this.fb.array([])      });    }      addEmployee() {      this.employees().push(this.newEmployee());    }      removeEmployee(empIndex: number) {      this.employees().removeAt(empIndex);    }      employeeSkills(empIndex: number): FormArray {      return this.employees()        .at(empIndex)        .get('skills') as FormArray;    }      newSkill(): FormGroup {      return this.fb.group({        skill: '',        exp: ''      });    }      addEmployeeSkill(empIndex: number) {      this.employeeSkills(empIndex).push(this.newSkill());    }      removeEmployeeSkill(empIndex: number, skillIndex: number) {      this.employeeSkills(empIndex).removeAt(skillIndex);    }      onSubmit() {      console.log(this.empForm.value);    }  } |

***app.component.html***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | <h1>Angular Nested FormArray / Dynamic FormArray</h1>    <form [formGroup]="empForm" (ngSubmit)="onSubmit()">    <div formArrayName="employees">      <div \*ngFor="let employee of employees().controls; let empIndex=index">        <div          [formGroupName]="empIndex"          style="border: 1px solid blue; padding: 10px; width: 600px; margin: 5px;"        >          {{empIndex}} First Name :          <input type="text" formControlName="firstName" />          Last Name:          <input type="text" formControlName="lastName" />            <button (click)="removeEmployee(empIndex)">Remove</button>            <div formArrayName="skills">            <div              \*ngFor="let skill of employeeSkills(empIndex).controls; let skillIndex=index"            >              <div [formGroupName]="skillIndex">                {{skillIndex}} Skill :                <input type="text" formControlName="skill" />                Exp:                <input type="text" formControlName="exp" />                  <button (click)="removeEmployeeSkill(empIndex,skillIndex)">                  Remove                </button>              </div>            </div>          </div>          <button type="button" (click)="addEmployeeSkill(empIndex)">            Add Skill          </button>        </div>      </div>      <button type="button" (click)="addEmployee()">Add Employee</button>    </div>  </form>    {{this.empForm.value | json}}    <br /><br />  <a    href="https://www.tektutorialshub.com/angular/nested-formarray-example-add-form-fields-dynamically/"    >Nested FormArray / Dynamic FormArray</a  > |

Nested FormArray Example

We built the nested Form array in the last tutorial. The code is as shown below. You can refer to the tutorial [nested FormArray Example](https://www.tektutorialshub.com/angular/nested-formarray-example-add-form-fields-dynamically/) for the explanation of the code

The following is the structure of our form model. There can be many teachers and each teacher can manage many batches. Each batch can contain several students.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | |Form  |-- Teacher  |-----Batch  |-------Students |

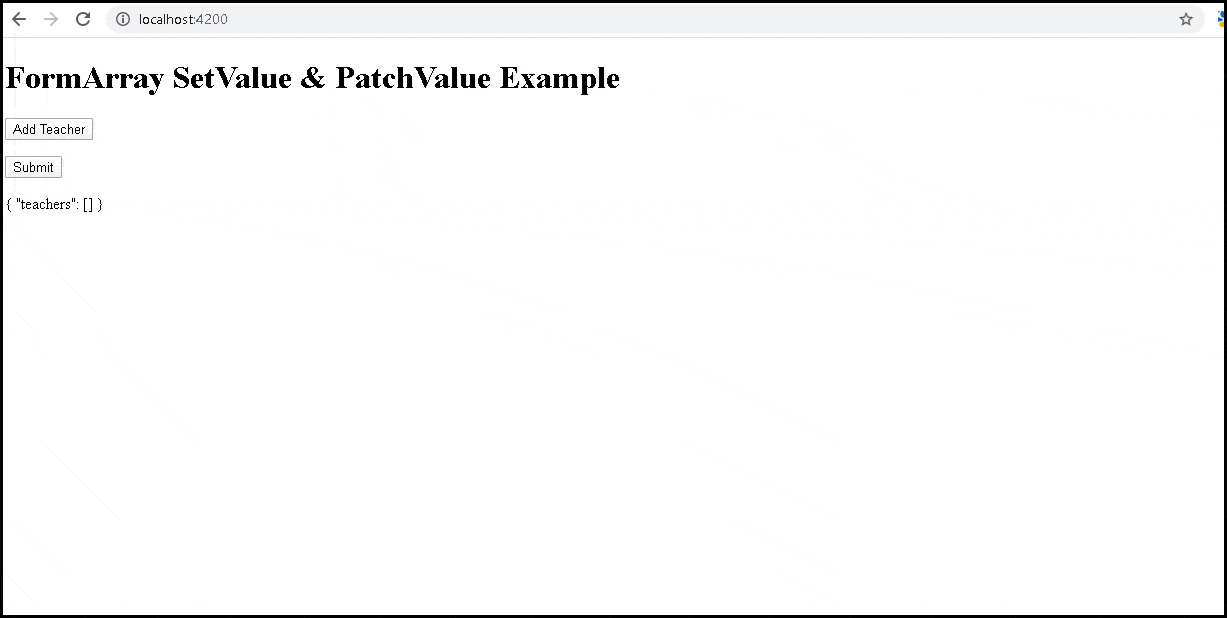
app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | <h1>{{title}}</h1>    <form [formGroup]="teachersForm" (ngSubmit)="onSubmit()">      <div formArrayName="teachers">        <div \*ngFor="let teacher of teachers().controls; let ti=index">          <div [formGroupName]="ti"          style="border: 1px solid blue; padding: 10px; width: 100%; display: inline-block; margin: 5px;">          Teachers Name :          <input type="text" formControlName="name">          <button (click)="removeTeacher(ti)">Remove</button>          <button type="button" (click)="addBatch(ti)">Add Batch</button>            <div formArrayName="batches">              <div \*ngFor="let batch of batches(ti).controls; let bi=index">                <div [formGroupName]="bi" style="border: 1px solid red; padding: 10px; margin: 5px; float:left">                  Batch Name :                <input type="text" formControlName="name">                <button (click)="removeBatch(ti,bi)">Remove Batch</button>                <button (click)="addStudent(ti,bi)">Add Student</button>                  <div formArrayName="students">                    <div \*ngFor="let batch of students(ti,bi).controls; let si=index">                    <div [formGroupName]="si" style="border: 1px solid blue; padding: 2px; ">                      Student Name :                      <input type="text" formControlName="name">                      <button (click)="removeStudent(ti,bi,si)">Remove student</button>                    </div>                  </div>                  </div>                </div>              </div>            </div>          </div>        </div>      </div>      <p>      <button type="button" (click)="addTeacher()">Add Teacher</button>    </p>    <p>      <button type="submit">Submit</button>    </p>    </form>    {{this.teachersForm.value | json}} |

app.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95 | import { Component } from '@angular/core';  import { FormGroup, FormArray, FormBuilder } from '@angular/forms'      @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {      title = 'FormArray SetValue & PatchValue Example';      teachersForm: FormGroup;      constructor(private fb: FormBuilder) {      this.teachersForm = this.fb.group({        teachers: this.fb.array([]),      })    }        /\*\* Teachers \*/    teachers(): FormArray {      return this.teachersForm.get("teachers") as FormArray    }      newTeacher(): FormGroup {      return this.fb.group({        name: '',        batches: this.fb.array([])      })    }        addTeacher() {      this.teachers().push(this.newTeacher());    }        removeTeacher(ti) {      this.teachers().removeAt(ti);    }        /\*\* batches \*/      batches(ti): FormArray {      return this.teachers().at(ti).get("batches") as FormArray    }        newBatch(): FormGroup {      return this.fb.group({        name: '',        students: this.fb.array([])      })    }      addBatch(ti: number) {      this.batches(ti).push(this.newBatch());    }      removeBatch(ti: number, bi: number) {      this.batches(ti).removeAt(ti);    }      /\*\* students \*/      students(ti, bi): FormArray {      return this.batches(ti).at(bi).get("students") as FormArray    }      newStudent(): FormGroup {      return this.fb.group({        name: '',      })    }      addStudent(ti: number, bi: number) {      this.students(ti, bi).push(this.newStudent());    }      removeStudent(ti: number, bi: number, si: number) {      this.students(ti, bi).removeAt(si);    }      onSubmit() {      console.log(this.teachersForm.value);    }    } |

Form in Action



How to load initial data in FormArray

When the form is loads for the first time, it will not have any controls in the FormArray. Calling PatchValue or SetValue will have no effect.

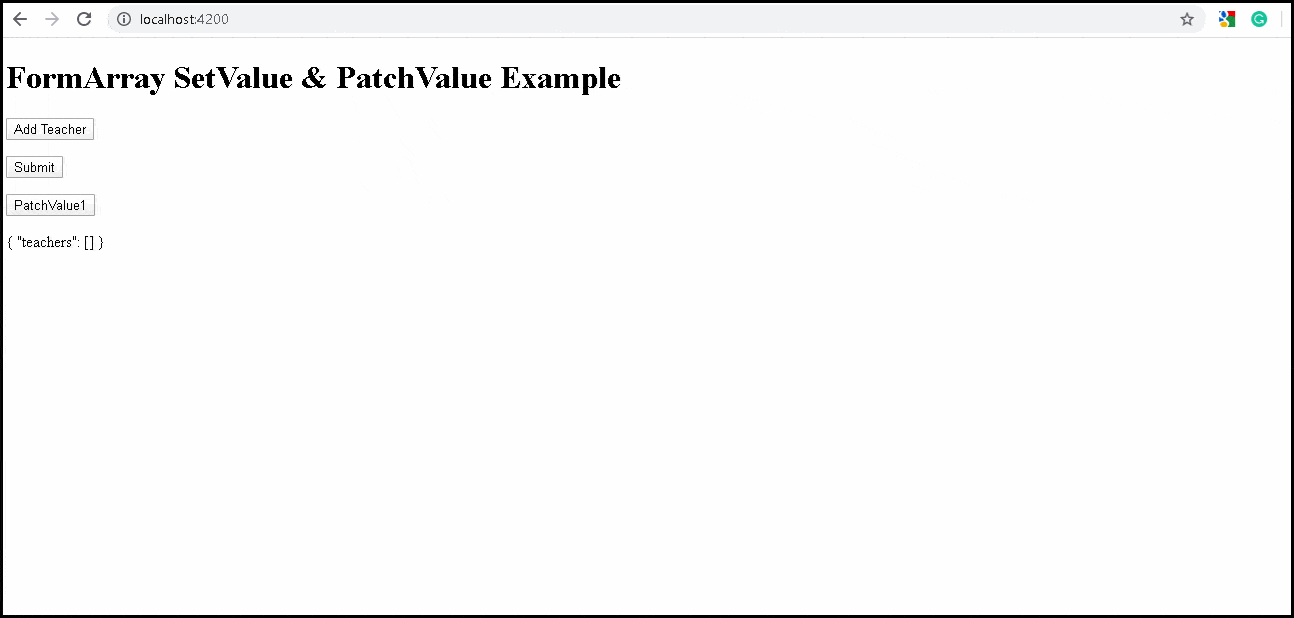
The patchValue1 method in the following example tries to load the data. The data contains one teacher managing two batches with three students in each batch.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | patchValue1() {      console.log('patchValue1')      var data = {        teachers: [          {            name: 'Teacher 1', batches: [              { name: 'Batch No 1', students: [{ name: 'Ramesh' }, { name: 'Suresh' }, { name: 'Naresh' }] },              { name: 'Batch No 2', students: [{ name: 'Vikas' }, { name: 'Harish' }, { name: 'Lokesh' }] },            ]          }        ]      }        this.teachersForm.patchValue(data);  } |

Invoke the method from the template.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <p>    <button (click)="patchValue1()">PatchValue1</button>  </p> |

As you can see from below patchValue won’t work unless we have all the controls loaded. Hence we need to build the form manually before calling patchValue



PatchValue Example

To load the data, we need to update our form programmatically to match the data. The patchValue2 method in the following example does that.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | patchValue2() {      var data = {      teachers: [        {          name: 'Teacher 1', batches: [            { name: 'Batch No 1', students: [{ name: 'Ramesh' }, { name: 'Suresh' }, { name: 'Naresh' }] },            { name: 'Batch No 2', students: [{ name: 'Vikas' }, { name: 'Harish' }, { name: 'Lokesh' }] },          ]        }      ]    }    this.clearFormArray();        data.teachers.forEach(t => {        var teacher: FormGroup = this.newTeacher();      this.teachers().push(teacher);        t.batches.forEach(b => {        var batch = this.newBatch();          (teacher.get("batches") as FormArray).push(batch)          b.students.forEach(s => {          (batch.get("students") as FormArray).push(this.newStudent())        })        });    });      this.teachersForm.patchValue(data);  }      clearFormArray() {      //Angular 8 +    this.teachers().clear();      //older Versions of angualar    //while (this.teachers().length) {    //  this.teachers().removeAt(0);    //}  } |

We start by clearing the FormArray. The clearing is useful if the user wants to discard and reload the original data again.

To clear the form, all You need to do is get hold of the top-level formArray and call clear method Note that clear is available from **Angular 8+**. If you are using the prior versions, then use the removeAt(index) method

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | clearFormArray() {      //Angular 8 +    this.teachers().clear();      //older Versions of angualar    //while (this.teachers().length) {    //  this.teachers().removeAt(0);    //}  } |

Now loop through the data

|  |  |
| --- | --- |
| 1  2  3 | data.teachers.forEach(t => { |

For every teacher create a nested form array and push it to the top-level formArray

|  |  |
| --- | --- |
| 1  2  3  4 | var teacher: FormGroup = this.newTeacher();      this.teachers().push(teacher); |

Now, loop through the batches and add it to the batches form array

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | t.batches.forEach(b => {          var batch = this.newBatch();        (teacher.get("batches") as FormArray).push(batch) |

Next, do the same for students

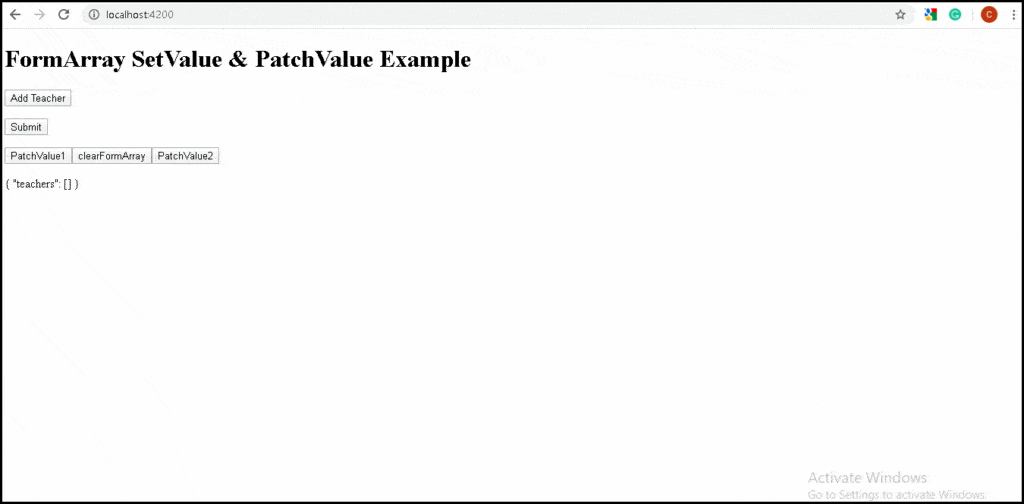
|  |  |
| --- | --- |
| 1  2  3  4  5 | b.students.forEach(s => {          (batch.get("students") as FormArray).push(this.newStudent())        }) |

Now, our form model matches the data structure. You can now call patchValue or setValue to update the Form. Remember the difference between PatchValue & SetValue.

The patchValue sets the data, even when data does not match the structure of the Form.

SetValue requires that the data must match the structure of the FormArray exactly

You can read about them from the tutorial [SetValue & PatchValue in Angular](https://www.tektutorialshub.com/angular/setvalue-patchvalue-in-angular/)



Summary

This guide demonstrated how to update the values of FormArray using the initial data. To do that first you need to update your form to match the structure of the data. Then make use of the either PatchValue or SetValue

Select Options in Reactive Forms

Create a contactForm using the [Reactive Form](https://www.tektutorialshub.com/angular/angular-reactive-forms/) as shown below. The Form has only one Formcontrol field i.e. country. The value for the country is going to come from the countries array. We will store the id in the country field

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | export class AppComponent implements OnInit  {      contactForm:FormGroup;      countries = [      { id: 1, name: "United States" },      { id: 2, name: "Australia" },      { id: 3, name: "Canada" },      { id: 4, name: "Brazil" },      { id: 5, name: "England" }    ];      constructor(private fb:FormBuilder) {    }      ngOnInit() {        this.contactForm = this.fb.group({        country: [null]      });    }      submit() {      console.log("Form Submitted")      console.log(this.contactForm.value)    }    } |

The select option for the country is as shown below.

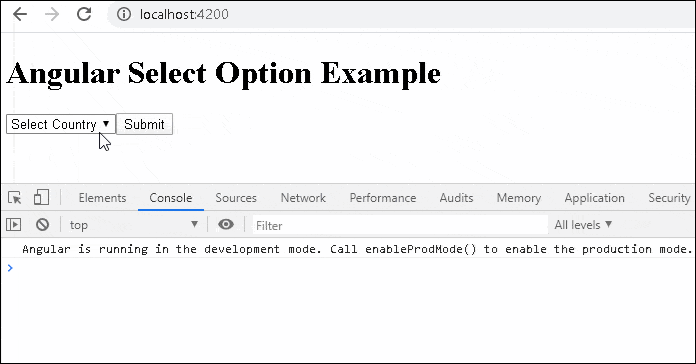
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | <form [formGroup]="contactForm" (ngSubmit)="submit()">    <p>      <select formControlName="country">      <option [ngValue]="null" disabled>Select Country</option>      <option \*ngFor="let country of countries" [ngValue]="country.id">{{country.name}}</option>    </select>      <button type="submit">Submit</button>  </p>    </form> |

The formControlName directive maps the select to the country field of the contactForm. ( <select formControlName="country">)

<option [ngValue]="null" disabled > displays the Select Country when no value for country is selected or country is null

The [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/) loops through the "countries" array and displays the country.name in the drop down. You can also display the country.name - country.id or any other property of the country

We want to return the id of the country. Hence we use the [ngValue]="country.id". You can also use [ngValue]="country", in which case it will return the country object.



[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Set the Default Value for the select

We can set the value of a [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) using either [setValue or patchValue](https://www.tektutorialshub.com/angular/setvalue-patchvalue-in-angular/).

Set the default values when defining the form to pass the default value. The following code sets the default country as Australia. This works only once when we load the component for the first time

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | ngOnInit() {      this.contactForm = this.fb.group({        country: [2]      });  ) |

The other method is to create a separate setDefaults method, which you call any time.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | ngOnInit() {      this.contactForm = this.fb.group({      country: [null]    });      this.setDefaults();  }    setDefaults() {    this.contactForm.get("country").patchValue(null);  } |

Listen to the Select option change event

Listen to the change event using the valueChanges event as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | this.contactForm.get("country").valueChanges      .subscribe(f=> {        this.onCountryChanged(f);    })      onCountryChanged(value) {      console.log('onCountryChanged')      console.log(value)    } |

Dynamically update the Drop down option

You can dynamically add or remove options from the countries array and it will reflect in the drop down immediately.

Ask for the new country in template.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | //Template    <p>    <input [(ngModel)]="country\_name">    <button (click)="addCountry()">Add</button>  </p> |

And add it to countries array after verifying it does not exists.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | //Component    country\_name="";    addCountry() {      const country = this.countries.find(el => el.name === this.country\_name);      if (!country) {        let id=Math.max.apply(Math, this.countries.map(function(o) { return o.id; }))        this.countries.push({id:id+1, name:this.country\_name})        this.country\_name="";      }    } |

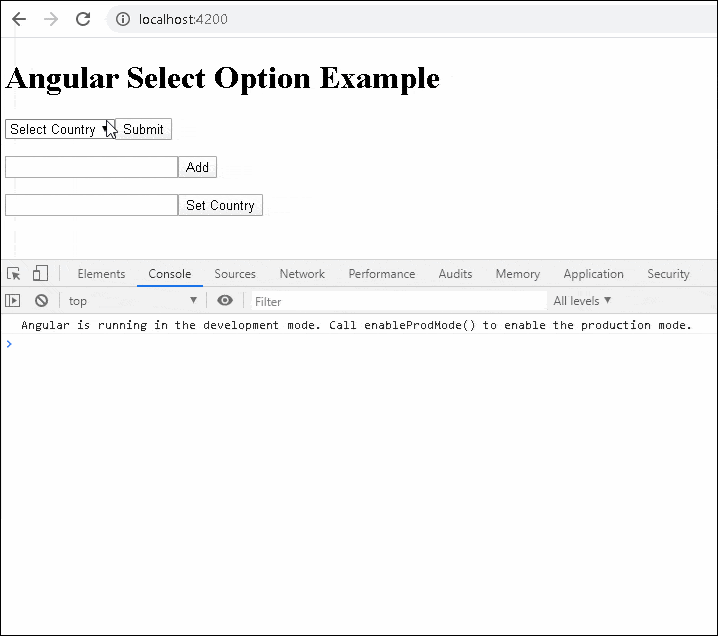
Dynamically Set Value

The following dynamically changes the value of the country via code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | //Template    <p>    <input [(ngModel)]="set\_country">    <button (click)="setCountry()">Set Country</button>  </p> |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | //Component    set\_country="";    setCountry() {    const country = this.countries.find(el => el.name === this.set\_country);    if (country) {      this.contactForm.get("country").patchValue(country.id);    }  } |

The final application

Select Option Drop down Example in Reactive Forms

Complete Code

app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | <h1> Angular Select Option Example</h1>    <form [formGroup]="contactForm" (ngSubmit)="submit()">    <p>      <select formControlName="country">      <option [ngValue]="null" disabled>Select Country</option>      <option \*ngFor="let country of countries" [ngValue]="country.id">{{country.name}}</option>    </select>      <button type="submit">Submit</button>  </p>    </form>    <p>    <input [(ngModel)]="country\_name">    <button (click)="addCountry()">Add</button>  </p>      <p>    <input [(ngModel)]="set\_country">    <button (click)="setCountry()">Set Country</button>  </p> |

app.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75 | import { Component, ViewChild, OnInit } from '@angular/core';  import { NgForm, FormGroup, FormControl, FormBuilder } from '@angular/forms';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {      title = 'Angular Select Options Example';    contactForm: FormGroup;      constructor(private fb: FormBuilder) {    }      country\_name="";    set\_country="";      countries = [      { id: 1, name: "United States" },      { id: 2, name: "Australia" },      { id: 3, name: "Canada" },      { id: 4, name: "Brazil" },      { id: 5, name: "England" }    ];        ngOnInit() {        this.contactForm = this.fb.group({        country: [null]      });        this.setDefaults();        this.contactForm.get("country").valueChanges        .subscribe(f => {          this.onCountryChanged(f);        })    }      submit() {      console.log("Form Submitted")      console.log(this.contactForm.value)    }      setDefaults() {      this.contactForm.get("country").patchValue(null);    }      onCountryChanged(value) {      console.log('onCountryChanged')      console.log(value)    }      addCountry() {      const country = this.countries.find(el => el.name === this.country\_name);      if (!country) {        let id=Math.max.apply(Math, this.countries.map(function(o) { return o.id; }))        this.countries.push({id:id+1, name:this.country\_name})        this.country\_name="";      }    }      setCountry() {      const country = this.countries.find(el => el.name === this.set\_country);      if (country) {        this.contactForm.get("country").patchValue(country.id);      }    }  } |

Select Options in Template Driven Forms

The following is the code for the select options in [Template Driven Forms](https://www.tektutorialshub.com/angular/angular-template-driven-forms/). Also refer to the tutorial on how to [set value in template driven forms](https://www.tektutorialshub.com/angular/how-to-set-value-in-template-driven-forms-in-angular/).

We get the reference to the contactForm using the @ViewChild.

Use the setTimeout() to wait for a change detection cycle so that the @ViewChild updates the reference to the contactForm. Without it the contactForm will return null.

To Listen to the change we use the [event binding](https://www.tektutorialshub.com/angular/event-binding-in-angular/) using the (ngModelChange)="onCountryChanged($event)". We can also use the valueChanges

Rest of the code is very similar to the Reactive forms.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73 | import { Component, ViewChild, OnInit } from '@angular/core';  import { NgForm, FormGroup, FormControl, FormBuilder } from '@angular/forms';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {      title = 'Angular Select Options Example';      @ViewChild("contactForm",null) contactForm:NgForm;        constructor(private fb: FormBuilder) {    }      country\_name="";    set\_country="";      countries = [      { id: 1, name: "United States" },      { id: 2, name: "Australia" },      { id: 3, name: "Canada" },      { id: 4, name: "Brazil" },      { id: 5, name: "England" }    ];        ngOnInit() {        setTimeout(() => {        this.setDefaults();        })      }      submit() {      console.log("Form Submitted")      console.log(this.contactForm.value)    }      setDefaults() {      this.contactForm.form.get("country").patchValue(null);    }      onCountryChanged(value) {      console.log('onCountryChanged')      console.log(value)    }        addCountry() {      const country = this.countries.find(el => el.name === this.country\_name);      if (!country) {        let id=Math.max.apply(Math, this.countries.map(function(o) { return o.id; }))        this.countries.push({id:id+1, name:this.country\_name})        this.country\_name="";      }    }      setCountry() {      const country = this.countries.find(el => el.name === this.set\_country);      if (country) {        this.contactForm.form.get("country").patchValue(country.id);      }    }    } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | <h1> Angular Select Option Example</h1>    <form #contactForm="ngForm" (ngSubmit)="submit()">      <p>        <select name="country" ngModel (ngModelChange)="onCountryChanged($event)">        <option [ngValue]="null" disabled>Select Country</option>        <option \*ngFor="let country of countries" [ngValue]="country.id">{{country.name}}</option>      </select>        <button type="submit">Submit</button>      </p>      </form>    <p>    <input [(ngModel)]="country\_name">    <button (click)="addCountry()">Add</button>  </p>      <p>    <input [(ngModel)]="set\_country">    <button (click)="setCountry()">Set Country</button>  </p> |

# Services & Dependency Injection

What is an Angular Service

Service is a piece of reusable code with a focused purpose. A code that you will use in many components across your application

Our components need to access the data. You can write data access code in each component, but that is very inefficient and breaks the rule of single responsibility. The Component must focus on presenting data to the user. The task of getting data from the back-end server must be delegated to some other class. We call such a class a Service class. Because it provides the service of providing data to every component that needs it.

What Angular Services are used for

Features that are independent of components such a logging services

Share logic or data across components

Encapsulate external interactions like data access

Advantageous of Angular Service

Services are easier to test.

They are easier to Debug.

We can reuse the service at many places.

How to create a Service in Angular

An Angular service is simply a Javascript function. All we need to do is to create a class and add methods & properties. We can then create an instance of this class in our component and call its methods.

One of the best uses of services is to get the data from the data source. Let us create a simple service, which gets the product data and passes it to our component.

You can download the source code from [stackBlitz](https://stackblitz.com/edit/angular-services-example-1?file=src/app/app.component.html)

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Product Model

Create a new file under the folder src/app and call it product.ts

***product.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | export class Product {        constructor(productID:number,    name: string ,   price:number) {          this.productID=productID;          this.name=name;          this.price=price;      }        productID:number ;      name: string ;      price:number;    } |

The Product class above is our domain model.

Product Angular Service

Next, let us build an Angular Service, which returns the list of products.

Create a new file under the folder src/app and call it product.service.ts

***product.service.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import {Product} from './product'    export class ProductService{        public  getProducts() {            let products:Product[];            products=[              new Product(1,'Memory Card',500),              new Product(1,'Pen Drive',750),              new Product(1,'Power Bank',100)          ]            return products;      }  } |

First, we import the Product model from the product.ts

Next, create ProductService class and export it. We need to export so that the Components & Other service class import it and use it

The getProducts method returns the collection of the products. In this example, we have hardcoded the products. In real life, you would send an [HTTP GET](https://www.tektutorialshub.com/angular/angular-http-get-example-using-httpclient/) request to your back end API to get the data

Now our service is ready.

Note that the above class is a simple JavaScript function. There is nothing Angular about it.

Invoking the ProductService

The Next step is to invoke the ProductService from the component. Open the app.componet.ts and add the following code.

***app.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | import { Component } from '@angular/core';    import { ProductService } from './product.service';  import { Product } from './product';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',  })    export class AppComponent  {       products:Product[];     productService;       constructor(){       this.productService=new ProductService();     }       getProducts() {         this.products=this.productService.getProducts();     }    } |

We start with importing Product & ProductService

We create an instance of ProductSerivce in the constructor of the AppComponet. In real-life Angular Apps, we use the [Dependency Injection in Angular](https://www.tektutorialshub.com/angular/angular-dependency-injection/) to inject the ProductSerivce in the constructor. We will learn that in the next tutorial.

The getProducts method calls the getProducts method of the ProductService. It returns a list of Products, which we store in the local variable products

Template

The next step is to display the Products to user

Open the app.component.html file and add the following code

***app.componenet.html***

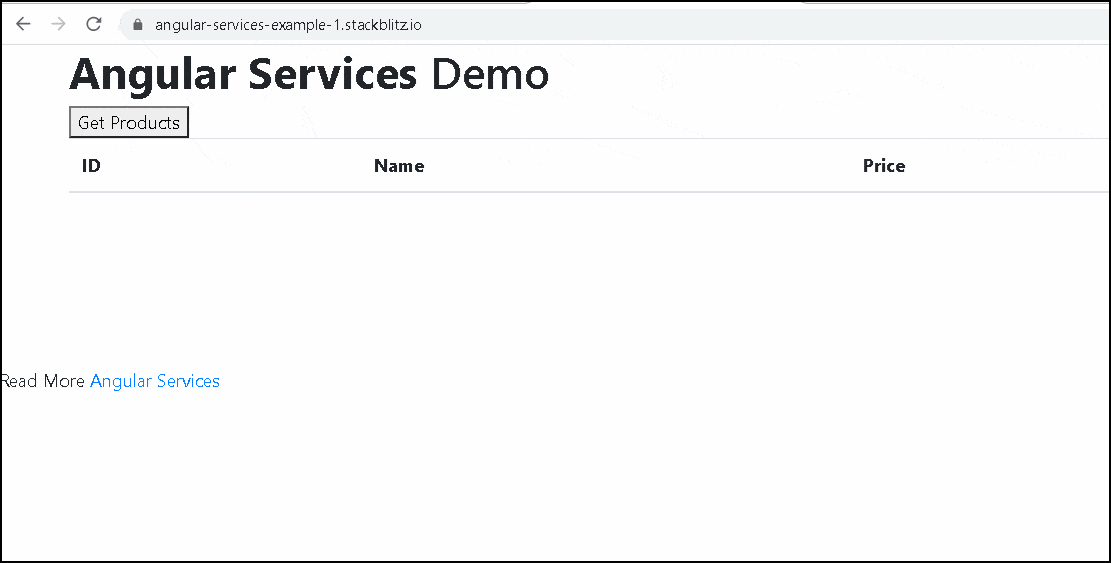
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | <div class="container">      <h1 class="heading"><strong>Services </strong>Demo</h1>        <button type="button" (click)="getProducts()">Get Products</button>         <div class='table-responsive'>              <table class='table'>                  <thead>                      <tr>                          <th>ID</th>                          <th>Name</th>                          <th>Price</th>                      </tr>                  </thead>                  <tbody>                      <tr \*ngFor="let product of products;">                          <td>{{product.productID}}</td>                          <td>{{product.name}}</td>                          <td>{{product.price}}</td>                      </tr>                  </tbody>              </table>          </div>    </div> |

We are using the bootstrap 4 to style our template here. You will find the link in the index.html.

The button Get Products, calls the getProducts method of the component class via [event binding](https://www.tektutorialshub.com/angular/angular-data-binding/)

We loop through the products via [ngFor directive](https://www.tektutorialshub.com/angular/angular-ngfor-directive/) and display it in a table.

Now, you can run the code and click on the Get Product button. You will see the List of Products



Injecting Services into Component

In the example, we instantiated the productService in the Component directly as shown below

|  |  |
| --- | --- |
| 1  2  3 | this.productService=new ProductService(); |

Directly instantiating the service, as shown above, has many disadvantageous

The ProductService is tightly coupled to the Component. If we change the ProductService class definition, then we need to update every code where service is used

If we want to change ProductService with BetterProductService, then we need to search wherever the ProductService is used and manually change it

Makes Testing difficult. We may need to provide mockProductService for testing and use the ProductService for Production.

What is Dependency

We built an ProductService in the [Angular Services](https://www.tektutorialshub.com/angular/angular-services/) tutorial. The AppComponent depends on the ProductService to provide the list of Products to display. You can view the code from [StackBlitz](https://stackblitz.com/edit/angular-services-example-1?file=src/app/app.component.ts)

In short, the AppComponent has a dependency on ProductService.

What is Angular Dependency Injection

**Dependency Injection (DI) is a technique in which a class receives its dependencies from external sources rather than creating them itself**.

Let us look at the ProductService, which we created in our [Angular Services](https://www.tektutorialshub.com/angular/angular-services/) tutorial. You can refer to the code from [StackBlitz](https://stackblitz.com/edit/angular-services-example-1?file=src/app/app.component.ts).

Our ProductService returns the hard-coded products when getProduct method invoked.

***product.service.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import {Product} from './Product'    export class ProductService{        public  getProducts() {            let products:Product[];            products=[              new Product(1,'Memory Card',500),              new Product(1,'Pen Drive',750),              new Product(1,'Power Bank',100)          ]            return products;      }  } |

[***StackBlitz***](https://stackblitz.com/edit/angular-services-example-1?file=src/app/product.ts)

If you used ng generate or VSCode “Generate Service”, it will add the following code. Please remove it. We will explain it at the end of the chapter.

@Injectable({ providedIn: 'root' })

We instantiated the productService directly in our [Component](https://www.tektutorialshub.com/angular/angular-component/) as shown below.

***app.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | import { Component } from '@angular/core';    import { ProductService } from './product.service';  import { Product } from './product';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',  })    export class AppComponent  {     products:Product[];     productService;       constructor(){       this.productService=new ProductService();     }       getProducts() {         this.products=this.productService.getProducts();     }    } |

[***StackBlitz***](https://stackblitz.com/edit/angular-services-example-1?file=src/app/product.ts)

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

The ProductService Instance is local to the Component. The AppComponent is now tightly coupled to the ProductService, This tight coupling brings a lot of Issues.

The ProductService hardcoded in our AppComponent. What if we want to use BetterProductService. We need to change wherever the ProductService is used and rename it to BetterProductService.

What if ProductService depends on another Service. And then we decide to change the service to some other service. Again we need to search and replace the code manually

It is hard to test this [Component](https://www.tektutorialshub.com/angular/angular-component/) as it is difficult to provide the Mock for the ProductService. For Instance, what if we wanted to substitute out the implementation of ProductService with MockProductService during testing.

Our Component Class has now tied one particular implementation of ProductService. It will make it difficult to reuse our components.

We would also like to make our ProductService singleton so that we can use it across our application.

How to solve all these problems. Move the creation of ProductService to the constructor the AppComponent class as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | export class AppComponent {     products:Product[];       constructor(private productService:ProductService) {       }       getProducts() {         this.products=this.productService.getProducts();     }  } |

Our AppComponent does not create the instance of the ProductService. It just asks for it in its Constructor. The responsibility of Creating the ProductService falls on the creator of the AppComponent.

The above pattern is the **Dependency Injection Pattern**.

Ad by Valueimpression

Benefits of Dependency Injection

loosely coupled

Our Component is now loosely coupled to the ProductService.

AppComponent does not know how to create the ProductService.  Actually, it does not know anything about the ProductService. It just works with the ProductService passed onto it. You can pass ProductService, BetterProductService or MockProductService. The AppComponent does not care.

Easier to Test

AppComponent is now easier to Test. Our AppComponent is not dependent on a particular implementation of ProductService anymore. It will work with any implementation of ProductService that is passed on to it. You can just create a mockProductService Class and pass it while testing.

Reusing the Component

Reusing of the component is becomes easier. Our Component will now work with any ProductService as long as the interface is honored.

Dependency injection pattern made our AppComponent testable, maintainable, etc.

But does it solve all our Problem ?. No, we just moved the Problem out of Component to the Creator of the Component.

How do we create an instance of ProductService and pass it to the AppComponent? That is what Angular Dependency Injection does.

Angular Dependency Injection Framework

Angular Dependency Injection framework implements the Dependency Injection in Angular. It creates & maintains the Dependencies and injects them into the Components, Directives, or Services.

There are five main players in the Angular Dependency injection Framework.

Consumer

The Consumer is the class (Component, Directive, or Service) that needs the Dependency. In the above example, the AppComponent is the Consumer.

Dependency

The [Service](https://www.tektutorialshub.com/angular/angular-services/) that we want to in our consumer. In the above example the ProductService is the Dependency

Injection Token (DI Token)

The [Injection Token](https://www.tektutorialshub.com/angular/injection-token-in-angular/) (DI Token) uniquely identifies a Dependency. We use [DI Token](https://www.tektutorialshub.com/angular/injection-token-in-angular/) when we register dependency

Provider

The [Providers](https://www.tektutorialshub.com/angular/angular-providers/) Maintain the list of Dependencies along with their [Injection Token](https://www.tektutorialshub.com/angular/injection-token-in-angular/). It uses the Injection Token is to identify the Dependency.

Injector

[Injector](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) holds the [*Providers*](https://www.tektutorialshub.com/angular/angular-providers/) and is responsible for resolving the dependencies and injecting the instance of the Dependency to the Consumer

The [Injector](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) uses Injection Token to search for Dependency in the [Providers](https://www.tektutorialshub.com/angular/angular-providers/). It then creates an instance of the dependency and injects it into the consumer

Using Dependency Injection

Registering the Dependency with the Injector

[Angular Provides](https://www.tektutorialshub.com/angular/angular-providers/) an instance of Injector & Provider to every component & directive in the application ( Consumers). It also creates an Injector instance at the module level and also at the root of the application. Basically, it creates a [Tree of Injectors with parent-child relationship](https://www.tektutorialshub.com/angular/how-dependency-injection-resolution-works-in-angular/)

The dependencies are registered with the [Provider](https://www.tektutorialshub.com/angular/angular-providers/). This is done in the [Providers](https://www.tektutorialshub.com/angular/angular-providers/) metadata of the Injector.

|  |  |
| --- | --- |
| 1  2  3 | providers: [ProductService] |

For Example, in the following code ProductService is registered with the Injector of the AppComponent

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Component({    selector: 'app-root',    templateUrl: './app.component.html',    providers: [ProductService]  })  export class AppComponent  { |

We can also add the Services to Providers array of the [**@NgModule**](https://www.tektutorialshub.com/angular/angular-modules/). Then they will be available for use in all the components & Services of the application. The ProductService in this case added to the Injector instance at the module level.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @NgModule({    declarations: [...],    imports: [...],    providers: [ProductService],    bootstrap: []  }) |

Asking for Dependency in the Constructor

The [Components](https://www.tektutorialshub.com/angular/angular-component/), [Directives](https://www.tektutorialshub.com/angular/angular-directives/) & [Services](https://www.tektutorialshub.com/angular/angular-services/) (Consumers) declare the dependencies that they need in their constructor.

|  |  |
| --- | --- |
| 1  2  3  4 | constructor(private productService:ProductService) {  } |

[Injector](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) reads the dependencies from the constructor of the Consumer. It then looks for that dependency in the provider. The Provider provides the instance and injector, then injects it into the consumer.

If the instance of the Dependency already exists, then it will reuse it. This will make the [dependency singleton](https://www.tektutorialshub.com/angular/angular-singleton-service/).

Angular Dependency Injection Example

We had created a simple ProductService in our last tutorial. Let us now update it to use **Dependency Injection.**

First, we need to register the dependencies with the provider. This is done in the providers metadata array of @Component decorator.

|  |  |
| --- | --- |
| 1  2  3 | providers: [ProductService] |

Next, we need to tell angular that our component needs dependency injection. This is done by using the [@Injectable()](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) decorator.

[@Injectable()](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) decorator is not needed if the class already has other [Angular decorators](https://www.tektutorialshub.com/angular/angular-decorators/) like @Component, @pipe or @directive etc. Because all these are a subtype of Injectible.

Since our AppComponent is already decorated with @Component, we do not need to decorate with the @Injectable

Next, our AppComponent needs to ask for the dependencies. We do that in the constructor of the Component.

|  |  |
| --- | --- |
| 1  2  3  4 | constructor(private productService:ProductService) {  } |

That’s it.

When AppComponent is instantiated it gets its own Injector instance. The Injector knows that AppComponent requires ProductService by looking at its constructor. It then looks at the Providers for a match and Provides an instance of ProductService to the AppComponent

The Complete AppComponent is as follows. You can refer to the [Stackblitz](https://stackblitz.com/edit/angular-depedency-injection-example-1?file=src/app/app.component.ts) for the source code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import { Component } from '@angular/core';    import { ProductService } from './product.service';  import { Product } from './product';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    providers: [ProductService]  })  export class AppComponent  {       products:Product[];       constructor(private productService:ProductService){     }       getProducts() {       this.products=this.productService.getProducts();     }    } |

Injecting Service into Another Service

We looked at how to inject ProductService to a component. Now let us look at how to inject service into another service.

Let us build loggerService, which logs every operation into a console window and inject it into our ProductService.

Logger Service

Create the logger.service.ts and add the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | import { Injectable } from '@angular/core';    @Injectable()  export class LoggerService {    log(message:any) {      console.log(message);    }  } |

The LoggerService has just one method log, which takes a message and writes it to the console.

We are also using @Injectible metadata to decorate our logger class. Technically, we do not have to do that as the logger service does not have any external dependencies.

We do not have to use the @Injectible if the class does not have any dependencies.

However, it is best practice is to decorate every service class with @Injectable(), even those that don’t have dependencies for the following reasons

**Future proofing:** No need to remember @Injectable() when we add a dependency later.

**Consistency:** All services follow the same rules, and we don’t have to wonder why a decorator is missing.

Product Service

Now we want to inject this into our ProductService class

The ProductService needs loggerService to be injected. Hence the class requires @Injectible metadata

|  |  |
| --- | --- |
| 1  2  3  4 | @Injectable()  export class ProductService{} |

Next, In the constructor of the ProductService ask for the loggerService.

|  |  |
| --- | --- |
| 1  2  3  4  5 | constructor(private loggerService: LoggerService) {      this.loggerService.log("Product Service Constructed");  } |

And update the GetProducts method to use the Logger Service.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public  getProducts() {            this.loggerService.log("getProducts called");          let products:Product[];            products=[              new Product(1,'Memory Card',500),              new Product(1,'Pen Drive',750),              new Product(1,'Power Bank',100)          ]            this.loggerService.log(products);          return products;      } |

Finally, we need to register LoggerService with the Providers metadata.

Angular does not have any options add providers in the Service Class. The Providers must be added to the Component, Directive, or in the Module.

Open the AppComponent update the providers array to include LoggerService

|  |  |
| --- | --- |
| 1  2  3 | providers: [ProductService,LoggerService] |

That’s it. As you click on the Get Products button, you will see the Console window updated with the Log messages

Providing Dependency from Angular Module

In the above example, we registered the dependencies in the Providers array of the component class. The dependencies are only available to the component where we register them and to its child components.

To Make the dependencies available to the entire application, we need to register it in the root module.

Remove the providers: [ProductService,LoggerService], from the AppComponent and move it to the  AppModule as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { HttpModule } from '@angular/http';  import { FormsModule } from '@angular/forms';    import { AppComponent } from './app.component';    import { ProductService } from './product.service';  import { LoggerService } from './logger.service';    @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule,      HttpModule,      FormsModule    ],    providers: [ProductService,LoggerService],    bootstrap: [AppComponent]  })  export class AppModule { } |

[***Stackblitz***](https://stackblitz.com/edit/angular-depedency-injection-example-2?file=src/app/app.module.ts)

Providing the service in the root module will create a single, shared instance of service and injects into any class that asks for it.

The above code works because Angular creates the [Tree of Injectors with parent-child relationship](https://www.tektutorialshub.com/angular/how-dependency-injection-resolution-works-in-angular/) similar to the Component Tree. We will cover it in the next tutorial.

ProvidedIn

Instead of adding ProductService to providers of the AppModule, you can also add it in the [providedIn](https://www.tektutorialshub.com/angular/providedin-root-any-platform-in-angular/) metadata with the value root.

In fact, using the [ProvidedIn](https://www.tektutorialshub.com/angular/providedin-root-any-platform-in-angular/) is the preferred way to provide a service in a module

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Injectable({    providedIn:'root'  })  export class ProductService { |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Injectable({    providedIn:'root'  })  export class LoggerService { |

Service Scope

The services that we provide at the root module are app-scoped**,** which means that we can access them from every component/service within the app.

Any service provided in the other Modules (Other than the [Lazy Loaded Module](https://www.tektutorialshub.com/angular/angular-lazy-loading/)) is also available for the entire application.

The services that are provided in a [Lazy Loaded Module](https://www.tektutorialshub.com/angular/angular-lazy-loading/) are module scoped and available only in the [Lazy loaded module](https://www.tektutorialshub.com/angular/angular-lazy-loading/).

The services provided at the Component level are available only to the Component & and to the child components.

What is Angular Injector

The [Angular](https://www.tektutorialshub.com/angular-tutorial/) Injector is responsible for instantiating the dependency and injecting it into the component or service.

The Injector looks for the dependency in the [Angular Providers](https://www.tektutorialshub.com/angular/angular-providers/) using the [Injection token](https://www.tektutorialshub.com/angular/angular-dependency-injection/#di-token). The Angular Providers array returns the Provider, which contains the information about how to create the instance of the dependency. The Injector creates the instance and injects it into the [Component](https://www.tektutorialshub.com/angular/angular-component/) or [service](https://www.tektutorialshub.com/angular/angular-services/).

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

When is Angular Injector is created

The Angular creates two Injector trees when the Application bootstraps. One is the ModuleInjector tree for the Modules and the other one is the ElementInjector tree which is for the Elements (Components & Directives etc).

The Angular loads the Root Module (named as AppModule) when the application [bootstraps](https://www.tektutorialshub.com/angular/angular-bootstrapping-application/). It creates RootModule Injector for the Root Module. This Injector has an application-wide scope. The RootModule Injector becomes part of the ModuleInjector Tree.

Angular Root Module loads the AppComponent, which is the root component of our app. The AppComponent gets its own Injector. We call this root Injector. This Injector becomes the root of the ElementInjector tree.

The Root Component contains all other components. [Angular App](https://www.tektutorialshub.com/angular/angular-create-first-application/) will create child components under the Root Component. All these child component can have their own child components creating a tree of components. The Angular also creates an Injector for all those components creating an Injector tree closely mimicking the component tree. These Injectors become part of the ElementInjector tree.

**To know more about the**[**Injector Tree**](https://www.tektutorialshub.com/angular/how-dependency-injection-resolution-works-in-angular/)**refer to the article**[**How Dependency Injection & Resolution Works in Angular**](https://www.tektutorialshub.com/angular/how-dependency-injection-resolution-works-in-angular/)**.**

The Every Injector gets its own copy of [Providers](https://www.tektutorialshub.com/angular/angular-providers/).

Registering the service with injector

We register all dependencies of the application with the Providers. Every injector has a Provider associated with it. The Providers metadata of @NgModule, @Component or @Directive is where we register our dependency

|  |  |
| --- | --- |
| 1  2  3 | providers: [ProductService, LoggerService] |

Where you register your dependency defines the scope of the dependency. The dependency registered with the Module using @NgModule decorator is attached the Root Provider ( Provider attached to the Root Injector). This Dependency is available to entire application.

The dependency registered with the component is available to that component and any child component of that component.

Another way to register the dependencies is to use the [ProvidedIn](https://www.tektutorialshub.com/angular/providedin-root-any-platform-in-angular/) property of the Injectable decorator

@Injectable

The Injectable is a decorator, which you need to add to the **consumer of the dependency**. This decorator tells angular that it must Inject the constructor arguments via the [Angular DI](https://www.tektutorialshub.com/angular/angular-dependency-injection/) system

Example of Injectable

We created an example application in the [Angular Dependency injection](https://www.tektutorialshub.com/angular/angular-dependency-injection/) tutorial. It had two services LoggerService & ProductService as shown below.

**LoggerService**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | import { Injectable } from '@angular/core';    @Injectable()  export class LoggerService {    log(message:any) {      console.log(message);    }  } |

**ProductService**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | import { Injectable } from '@angular/core';    import {Product} from './Product'  import {LoggerService} from './logger.service'    @Injectable()  export class ProductService{        constructor(private loggerService: LoggerService) {          this.loggerService.log("Product Service Constructed");      }        public  getProducts() {            this.loggerService.log("getProducts called");          let products:Product[];            products=[              new Product(1,'Memory Card',500),              new Product(1,'Pen Drive',750),              new Product(1,'Power Bank',100)          ]            this.loggerService.log(products);          return products;      }  } |

The ProductService has a dependency on the LoggerService. Hence it is decorated with the @Injectable decorator. Remove @Injectable() from ProductService and you will get the following error.

**Uncaught Error: Can’t resolve all parameters for ProductService: (?)**

That is because without [DI](https://www.tektutorialshub.com/angular/angular-dependency-injection/) Angular will not know how to inject LoggerService into ProductService.

Remove @Injectable() from LoggerService will not result in any error as the LoggerService do not have any dependency.

The [Components](https://www.tektutorialshub.com/angular/angular-component/) & [Directives](https://www.tektutorialshub.com/angular/angular-directives/) are already decorated with @Component & @Directive decorators. These decorators also tell Angular to use DI, hence you do not need to add the @Injectable().

The injectable decorator also has the [ProvidedIn](https://www.tektutorialshub.com/angular/providedin-root-any-platform-in-angular/) property using which you can specify how Angular should provide the dependency.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Injectable({     providedIn: 'root'  })  export class SomeService{  } |

@Inject

The @Inject() is a constructor parameter decorator, which tells angular to Inject the parameter with the dependency provided in the given token. It is a manual way of injecting the dependency

In the previous example, when we removed the @Injectable decorator from the ProductService we got an error.

We can manually inject the LoggerService by using the @Inject decorator applied to the parameter loggerService as shown below.

The @Inject takes the Injector token as the parameter. The token is used to locate the dependency in the Providers.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | export class ProductService{      constructor(@Inject(LoggerService) private loggerService) {          this.loggerService.log("Product Service Constructed");      }  } |

What are Angular Providers

The **Angular Provider** is an instruction (or recipe) that describes how an object for a certain token is created. The **Angular Providers** is an array of such instructions (Provider). Each provider is uniquely identified by a **token** (or DI  Token ) in the Providers Array.

We register the services participating in the [dependency injections](https://www.tektutorialshub.com/angular/angular-dependency-injection/) in the Providers metadata. There are two ways by which we can do it.

Register directly in the Providers array of the @NgModule or @Component or in @Directive.

Or use the providedIn property of the @[Injectable](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) decorator.

The Angular creates an [Injector](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) for each component/directive it creates. It also creates a root-level injector, which has the app-level scope. It also creates a Module level Injector for [Lazy Loaded Modules](https://www.tektutorialshub.com/angular/angular-lazy-loading/).

Each [Injector](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) gets its own copy of the Providers. We can the same dependency with multiple providers. Where & how you register the dependency defines the scope of the dependency

The [Angular Components](https://www.tektutorialshub.com/angular/angular-component/) or [Angular Services](https://www.tektutorialshub.com/angular/angular-services/) declare the dependencies they need in their constructor. The [Injector](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) reads the dependencies and looks for the provider in the providers array using the Token. It then instantiates the dependency using the instructions provided by the provider. The Injector then injects the instance of the dependency into the Components/Services.

Configuring the Angular Provider

To Provide an instance of the dependency, we need to register it in the Providers metadata

In our last tutorial on [Angular Dependency injection](https://www.tektutorialshub.com/angular/angular-dependency-injection/), we registered our ProductService using the Providers arrays as shown below in the @NgModule

|  |  |
| --- | --- |
| 1  2  3 | providers: [ProductService] |

The above is an actual shorthand notation for the following syntax

|  |  |
| --- | --- |
| 1  2  3 | providers :[{ provide: ProductService, useClass: ProductService }] |

The above syntax has two properties.

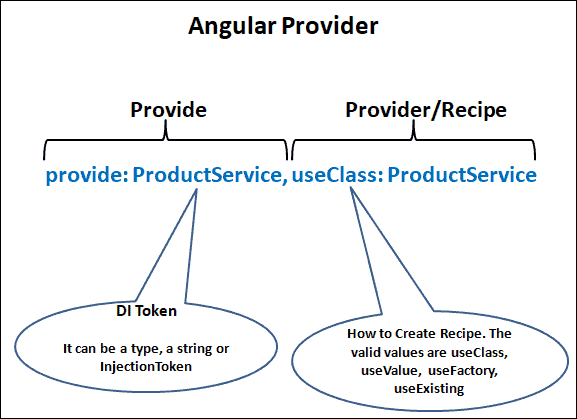
[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Provide

The first property is Provide holds the [**Token or DI Token**](https://www.tektutorialshub.com/angular/injection-token-in-angular/). The Injector uses the token to locate the provider in the Providers array. The Token can be either a type, a string or an instance of [InjectionToken](https://www.tektutorialshub.com/angular/injection-token-in-angular/).

Provider

The second property is the Provider definition object. It tells Angular how to create the instance of the dependency. The Angular can create the instance of the dependency in four different ways. It can create a dependency from the existing service class (useClass). It can inject a value, array, or object (useValue). It can use a factory function, which returns the instance of service class or value (useFactory). It can return the instance from an already existing token (useExisting).



DI Token

The Injector maintains an **internal collection of token-provider** in the Providers array. The token acts as a key to that collection & Injector use that Token (key) to locate the Provider.

The [DI Token](https://www.tektutorialshub.com/angular/injection-token-in-angular/) can be either type, a string or an instance of InjectionToken.

Type Token

Here the type being injected is used as the token.

For Example, we would like to inject the instance of the ProductService, we will use the ProducService as the token as shown below

|  |  |
| --- | --- |
| 1  2  3 | providers :[{ provide: ProductService, useClass: ProductService }] |

The ProductService is then injected to the component by using the following code.

|  |  |
| --- | --- |
| 1  2  3  4  5 | class ProductComponent {    constructor(private productService : ProductService ) {}  } |

String token

Instead of using a type, we can use a string literal to register the dependency. This is useful in scenarios where the dependency is a value or object etc, which is not represented by a class.

Example

|  |  |
| --- | --- |
| 1  2  3  4  5 | {provide:'PRODUCT\_SERVICE', useClass: ProductService },  {provide:'USE\_FAKE', useValue: true },  {provide:'APIURL', useValue: 'http://SomeEndPoint.com/api' }, |

You can then use the Inject the dependency using the @Inject method

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class ProductComponent {     constructor(@Inject('PRODUCTSERVICE') private prdService:ProductService,  @Inject('APIURL') private apiURL:string ) {  } |

Injection Token

The Problem with the string tokens is that two developers can use the same string token at a different part of the app. You also do not have any control over the third-party modules, which may use the same token. If the token is reused, the last to register overwrites all previously registered tokens.

The Angular provides [InjectionToken](https://www.tektutorialshub.com/angular/injection-token-in-angular/) class so as to ensure that the Unique tokens are created. The Injection Token is created by creating a new instance of the InjectionToken class.

|  |  |
| --- | --- |
| 1  2  3 | export const API\_URL= new InjectionToken<string>(''); |

Register the token in the providers array.

|  |  |
| --- | --- |
| 1  2  3  4  5 | providers: [      { provide: API\_URL, useValue: 'http://SomeEndPoint.com/api' }  ] |

It is then injected using the @Inject in the constructor of the service/component.

|  |  |
| --- | --- |
| 1  2  3  4 | constructor(@Inject(API\_URL) private apiURL: string) {  } |

The Types of Provider

The [Angular Dependency Injection](https://www.tektutorialshub.com/angular/angular-dependency-injection/) provides several types of providers.

Class Provider : useClass

Value Provider: useValue

Factory Provider: useFactory

Aliased Class Provider: useExisting

Class Provider: useClass

Use the Class Provider useClass, when you want to provide an instance of the provided class.

The useClass expects us to provide a type. The [Injector](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) creates a new instance from the type and injects it. It is similar to calling the new operator and returning instance. If the type requires any constructor parameters, the injector will resolve that also.

UseClass Example

|  |  |
| --- | --- |
| 1  2  3 | providers :[{ provide: ProductService, useClass: ProductService }] |

[***Stackblitz***](https://stackblitz.com/edit/angular-providers-useclass-example1?file=src/app/app.module.ts)

In the above, example ProductService is the Token (or key) and it maps to the ProductService Class. In this case both the Class name and token name match.

The Angular Provides a shortcut in cases where both token & class name matches as follows

|  |  |
| --- | --- |
| 1  2  3 | providers: [ProductService] |

Switching Dependencies

You can provide a mock/Fake class for Testing purposes as shown below.

|  |  |
| --- | --- |
| 1  2  3 | providers :[{ provide: ProductService, useClass: fakeProductService }] |

[***Stackblitz***](https://stackblitz.com/edit/angular-providers-useclass-example1?file=src/app/app.module.ts)

The above example shows us how easy to switch dependencies.

Value Provider: useValue

Use the Value Provider useValue, when you want to provide a simple value.

The Angular will injects whatever provided in the useValue as it is.

It is useful in scenarios like, where you want to provide API URL, application-wide configuration Option, etc

UseValue Example

In the example below, we pass a boolean value using token USE\_FAKE. You can check the [StackBlitz](https://stackblitz.com/edit/angular-providers-usevalue-example1?file=src/app/app.component.ts) for the source code

|  |  |
| --- | --- |
| 1  2  3 | providers :[ {provide:'USE\_FAKE', useValue: true}] |

We can inject it into the AppComponent using the @Inject

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | export class AppComponent {    constructor(      @Inject('USE\_FAKE') public useFake: string    ) {} |

You can pass an object. Use Object.freeze to freeze the value of the configuration, so that nobody can change it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | const APP\_CONFIG =  Object.freeze({    serviceURL: 'www.serviceUrl.comapi',    IsDevleomentMode: true  }); |

Register it.

|  |  |
| --- | --- |
| 1  2  3  4  5 | providers: [      { provide: 'APP\_CONFIG', useValue: APP\_CONFIG }    ] |

Inject it as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | export class AppComponent {  constructor(      @Inject('APP\_CONFIG') public appConfig: any    ) {}  } |

You can also provide a function

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | providers: [      {        provide: 'FUNC',        useValue: () => {          return 'hello';        }      }    ] |

The Injector will inject the function as it is. You need to invoke the function someFunc() to get a value from it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | export class AppComponent {    constructor(      @Inject('FUNC') public someFunc: any    ) {      console.log(someFunc());    }  } |

[***Stackblitz***](https://stackblitz.com/edit/angular-providers-usevalue-example1?file=src/app/app.component.ts)

Factory Provider: useFactory

The Factory Provider useFactory expects us to provide a function. It invokes the function and injects the returned value. We can also add optional arguments to the factory function using the deps array. The deps array specifies how to inject the arguments.

We usually use the useFactory when we want to return an object based on a certain condition.

UseFactory example

Consider the use case where we want to inject either ProductService or FakeProductService based on the value for USE\_FAKE. Also, one of the service (ProductService ) requires another service (LoggerService). Hence we need to inject USE\_FAKE & LoggerService into our factory function.

You can refer to [Stackblitz](https://stackblitz.com/edit/angular-providers-usefactory-example1?file=src/app/app.module.ts) for the example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | providers: [      { provide: LoggerService, useClass: LoggerService },        { provide: 'USE\_FAKE', useValue: true },        {        provide: ProductService,        useFactory: (USE\_FAKE, LoggerService) =>          USE\_FAKE ? new FakeProductService() : new ProductService(LoggerService),        deps: ['USE\_FAKE', LoggerService]      }    ] |

[***Stackblitz***](https://stackblitz.com/edit/angular-providers-usefactory-example1?file=src/app/app.module.ts)

We need to pass all the dependency of the as the argument to the factory function. The injector uses the deps array (third argument) to resolve the dependencies and inject them.

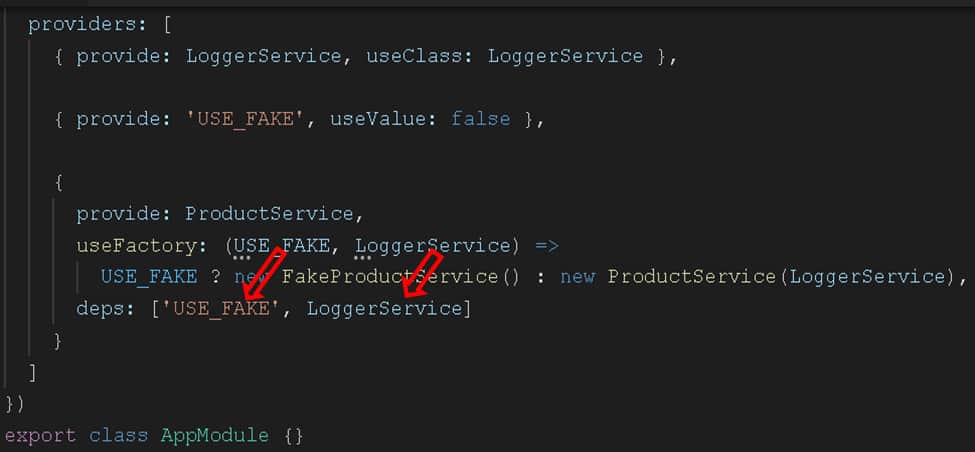
|  |  |
| --- | --- |
| 1  2  3 | useFactory: (USE\_FAKE, LoggerService) |

inside the factory function, we either return FakeProductService or ProductService depending on the value of USE\_FAKE

|  |  |
| --- | --- |
| 1  2  3  4 | =>     USE\_FAKE ? new FakeProductService() : new ProductService(LoggerService) |

In the last option, we need to tell the Injector how to inject the dependencies of the Factory function itself. Note that order must be the same as that of the factory function argument.

|  |  |
| --- | --- |
| 1  2  3 | deps: ['USE\_FAKE', LoggerService] |



The above example can also be written as follows.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | export function resolveProductService(USE\_FAKE, LoggerService) {    return USE\_FAKE      ? new FakeProductService()      : new ProductService(LoggerService);  }            providers: [      { provide: LoggerService, useClass: LoggerService },        { provide: 'USE\_FAKE', useValue: false },        {        provide: ProductService,        useFactory: resolveProductService,        deps: ['USE\_FAKE', LoggerService]      }    ] |

useFactory Vs useValue

In the useValue example, we used the following code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | providers: [      {        provide: 'FUNC',        useValue: () => {          return 'hello';        }      }    ] |

The useValue returns the function as it is. You need to call the function (someFunc()) to get the value.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | export class AppComponent {    constructor(      @Inject('FUNC') public someFunc: any    ) {      console.log(someFunc());    }  } |

You can achieve the same with the useFactory

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | providers: [      {        provide: 'FUNC',        useFactory: () => {          return 'hello';        }      }    ] |

The useFactory invokes the factory function and returns the result. Hence in the component, you will receive the value of the function and not the function itself.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | export class AppComponent {    constructor(      @Inject('FUNC') public someFunc: any    ) {      console.log(someFunc);    }  } |

Aliased Provider: useExisting

Use Aliased Provider useExisting when you want to use the new provider in place of the old Provider.

UseExisting Example

|  |  |
| --- | --- |
| 1  2  3  4  5 | providers: [      { provide: ProductService, useExisting: NewProductService },      { provide: NewProductService, useClass: NewProductService }, |

[***Stackblitz***](https://stackblitz.com/edit/angular-providers-useexisting-example1?file=src/app/app.module.ts)

For Example, in the above example, we map the ProductService to the NewProductService token using useExisting Provider. This will return the NewProductService whenever we use the ProductService.

Also, note that we need to pass the token to the useExisting and not type. The following example shows useExisting with string tokens.

|  |  |
| --- | --- |
| 1  2  3  4  5 | providers: [      { provide: ProductService, useExisting: 'PRODUCT\_SERVICE' },      { provide: 'PRODUCT\_SERVICE', useClass: NewProductService }, |

Multiple Providers with the same token

You can add as many dependencies to the Providers array.

The Injector does not complain, if you add more than one provider with the same token

For example, NgModule below adds both ProductService & FakeProductService using the same token ProductService.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @NgModule({      ...    providers: [      { provide: ProductService, useClass: ProductService },      { provide: ProductService, useClass: FakeProductService },      ]  })  export class AppModule {} |

In such a scenario, the last to register wins. The ProductService token always injects FakeProductService because we register it last.

Registering the Dependency at Multiple Providers

You can also register a Dependency with Multiple Providers.

For Example, here we register the ProductService in NgModule

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @NgModule({      ...    providers: [      { provide: ProductService, useClass: ProductService },    ]  })  export class AppModule {} |

We can also go and register it in AppComponent. In this case, AppComponent always gets the dependency registered in the component itself.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Component({    selector: 'my-app',    templateUrl: './app.component.html',    providers: [ProductService]  })  export class AppComponent {    products: Product[]; |

Provider Scope

Where you register the dependency, defines the lifetime of the dependency.

When we provide the service in the @ngModule of the ***root module*** or any ***eagerly loaded module***, the will be available everywhere in the application.

If we provide the services in the @Component, @pipe or @Directive then they are available only in that component and all of its child components

The Services provided in the @ngModule of the [lazy loaded module](https://www.tektutorialshub.com/angular/angular-lazy-loading/) are available **in that module only**.

Singleton services

Each Injector creates a singleton object of the dependency registered by the provider.

For Example, consider a service configured in @ngModule. Component A asks for the service it will get a new instance of the service. Now if Component B Asks for the same service, the injector does not create a new instance of the service, but it will reuse the already created service.

But if we register the service in @ngModule and also in Component A. Component A always gets a new instance of the service. While other components gets the instance of the service registered in @ngModule.

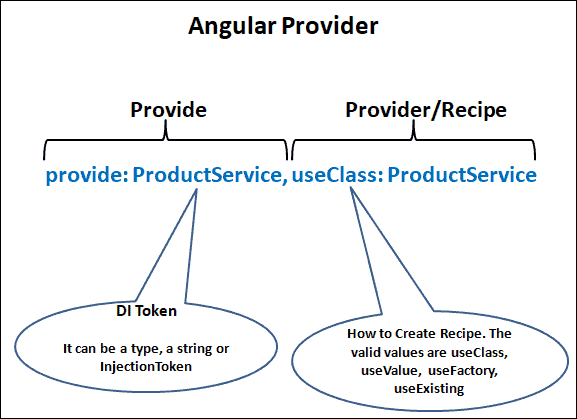
DI Tokens

We declare the Provider in with providers metadata. This is how it looks like.

|  |  |
| --- | --- |
| 1  2  3 | providers :[{ provide: ProductService, useClass: ProductService }] |

The syntax has two properties. **provide** (provide: ProductService) & **provider** (useClass: ProductService)

The first property is Provide holds the **Token or DI Token**. The Tokens act like a key. The DI systems need the key to locate the provider in the Providers array.



The Token can be either a type, a string or an instance of InjectionToken.

Type Token

Here the type being injected is used as the token.

For Example, we would like to inject the instance of the ProductService, we will use the ProducService as the token as shown below

|  |  |
| --- | --- |
| 1  2  3 | providers :[{ provide: ProductService, useClass: ProductService }] |

The ProductService is then injected into the component by using the following code.

|  |  |
| --- | --- |
| 1  2  3  4  5 | class ProductComponent {    constructor(private productService : ProductService ) {}  } |

You can keep the same token (ProductService) and change the class to another implementation of the Product service. For Example in the following code, we change it to BetterProductService.

|  |  |
| --- | --- |
| 1  2  3  4 | providers: [      { provide: ProductService, useClass: BetterProductService }, |

Angular does not complain if we use the token again. In the following example token ProductService used twice. In such a situation last to register wins (BetterProductService).

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | providers: [      { provide: ProductService, useClass: ProductService },      { provide: ProductService, useClass: BetterProductService }    ] |

[***Stackblitz***](https://stackblitz.com/edit/angular-injection-token-type-token-example?file=src/app/app.module.ts)

String token

You can use the Type token only if you have Type representation. But that is not the case always. Sometimes we need to inject simple string values or simple object literal, where there is no type.

We can use string tokens in such a scenario.

Example

|  |  |
| --- | --- |
| 1  2  3 | providers: [{ provide: 'PRODUCT\_SERVICE', useClass: ProductService }] |

You can then use the Inject the ProductService using the @Inject method

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | export class AppComponent {    products: Product[];      constructor(      @Inject('PRODUCT\_SERVICE') private productService: ProductService    ) {} |

Example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | providers: [      { provide: 'PRODUCT\_SERVICE', useClass: const CONFIG = {    apiUrl: 'http://my.api.com',    fake: true,    title: 'Injection Token Example'  };    @NgModule({    imports: [BrowserModule, FormsModule],    declarations: [AppComponent, HelloComponent],    bootstrap: [AppComponent],    providers: [      { provide: 'PRODUCT\_SERVICE', useClass: ProductService },      { provide: 'USE\_FAKE', useValue: true },      { provide: 'APIURL', useValue: 'http://SomeEndPoint.com/api' },      { provide: 'CONFIG', useValue: CONFIG }    ]  })  export class AppModule {} |

[***Stackblitz***](https://stackblitz.com/edit/angular-injection-token-string-token-example1?file=src/app/app.component.ts)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | export class AppComponent {    products: Product[];      constructor(      @Inject('PRODUCT\_SERVICE') private productService: ProductService,      @Inject('USE\_FAKE') public fake: String,      @Inject('APIURL') public ApiUrl: String,      @Inject('CONFIG') public Config: any    ) {} |

[***C***](https://stackblitz.com/edit/angular-injection-token-string-token-example1?file=src/app/app.component.ts)***ode***

Problems with the String Tokens

The String tokens are easy to use but prone to error. Two developers can use the same token at the different parts of the app. The third-party libraries can also use the same token.

If we re-use the token, then the last to register overwrites all previously registered tokens.

String tokens are easier to mistype and that makes it difficult to track & maintain in big applications.

This is where the InjectionToken comes into the picture.

What is an Injection Token

The Injection Token allows creating token that allows the injection of values that don’t have a runtime representation.

It is very similar to string tokens. But instead of using a hardcoded string, we create the Injection Token by creating a new instance of the InjectionToken class. They ensure that the tokens are always unique.

In Angular 4 and prior versions used OpaqueToken. It now deprecated and replaced by InjectionToken.

Creating an InjectionToken

To Create an Injection Token, first, we need to import InjectionToken from @angular/core

|  |  |
| --- | --- |
| 1  2  3 | import { InjectionToken } from '@angular/core'; |

Create a new Injection Token APIURL from InjectionToken

|  |  |
| --- | --- |
| 1  2  3 | export const APIURL = new InjectionToken<string>(''); |

Register it in the providers array.

|  |  |
| --- | --- |
| 1  2  3  4 | providers: [  { provide: APIURL, useValue: 'http://SomeEndPoint.com/api' }, |

Inject it into the Component

|  |  |
| --- | --- |
| 1  2  3  4  5 | export class AppComponent {    constructor(@Inject(APIURL) public ApiUrl: String,) { }  } |

InjectionToken Example

The following example shows how to use the Injection Token. You can refer to the [Stackblitz](https://stackblitz.com/edit/angular-injection-token-injection-token-example1?file=src/app/tokens.ts) for the code

***token.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | import { InjectionToken } from '@angular/core';    export const APIURL = new InjectionToken<string>('');  export const USE\_FAKE = new InjectionToken<string>('');  export const PRODUCT\_SERVICE = new InjectionToken<string>('');  export const APP\_CONFIG = new InjectionToken<string>(''); |

***app.module.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | import { NgModule } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { FormsModule } from '@angular/forms';    import { AppComponent } from './app.component';  import { HelloComponent } from './hello.component';  import { ProductService } from './product.service';  import { BetterProductService } from './better-product.service';  import { PRODUCT\_SERVICE, USE\_FAKE, APIURL, APP\_CONFIG } from './tokens';    const CONFIG = {    apiUrl: 'http://my.api.com',    fake: true,    title: 'Injection Token Example'  };    @NgModule({    imports: [BrowserModule, FormsModule],    declarations: [AppComponent, HelloComponent],    bootstrap: [AppComponent],    providers: [      { provide: PRODUCT\_SERVICE, useClass: ProductService },      { provide: USE\_FAKE, useValue: true },      { provide: APIURL, useValue: 'http://SomeEndPoint.com/api' },      { provide: APP\_CONFIG, useValue: CONFIG }    ]  })  export class AppModule {} |

[***Stackblitz***](https://stackblitz.com/edit/angular-injection-token-injection-token-example1?file=src/app/app.module.ts)

***app.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | import { Component, Inject } from '@angular/core';  import { ProductService } from './product.service';  import { Product } from './product';  import { PRODUCT\_SERVICE, USE\_FAKE, APIURL, APP\_CONFIG } from './tokens';    @Component({    selector: 'my-app',    templateUrl: './app.component.html',    providers: []  })  export class AppComponent {    products: Product[];      constructor(      @Inject(PRODUCT\_SERVICE) private productService: ProductService,      @Inject(USE\_FAKE) public fake: String,      @Inject(APIURL) public ApiUrl: String,      @Inject(APP\_CONFIG) public Config: any    ) {}      getProducts() {      this.products = this.productService.getProducts();    }  } |

Injector

The Angular creates an Injector instance for **every Component, Directive**, etc it loads. It also creates an injector instance for the **Root Module** and for **every lazy loaded module**. But eagerly loaded modules do not get their own injector but share the injector of the Root Module.

Injector Tree

Angular Creates not one but two injector trees. Module Injector tree & Element Injector tree.

Module Injector tree is for Modules (@NgModule). For Root Module & for every Lazy Loaded Module.

Element Injector tree is for DOM Elements like Components & Directives.

Module Injector Tree

Angular creates the ModuleInjector for the services to be provided at Module Levels.

We register the Module level services in two ways

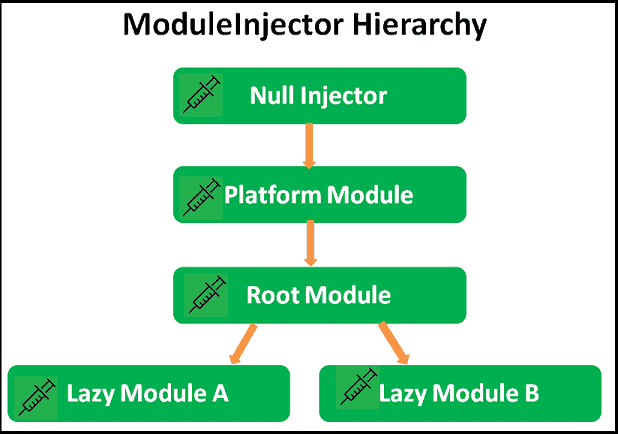
Using The Providers Metadata of the @[NgModule()](https://www.tektutorialshub.com/angular/angular-modules/)

Using the [@Injectable()](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) Decorator with providedIn : root in the service itself

Angular Creates the Module Injector tree when the Application starts.

At the top of the **Module Injector tree**, Angular creates an instance of **Null Injector**. The Null Injector always throws an error unless we decorate the dependency with the [Optional decorator](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/).

Under **Null Injector** Angular creates an instance of PlatformInjector. Platform Injector usually includes built-in providers like DomSanitize etc.



Under the Platform Injector, Angular creates the Injector for the Root Module. It is configured with the providers from the following locations.

Providers metadata of @NgModule of Root Module.

Providers metadata of @NgModule of all the imported Modules (i.e. all **eagerly** loaded modules).

All the services which have providedIn metadata with value root or any in their [@Injectable()](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) decorator. It includes services from both eagerly loaded & [Lazy Loaded Modules](https://www.tektutorialshub.com/angular/angular-lazy-loading/).

Under Root Module Injector, Angular creates an Injector instance for every Lazy loaded Module. Angular creates them only when it loads them. They are configured with the providers from the following locations.

Providers metadata of @NgModule of the Module being Lazy loaded

All the services which have providedIn metadata with value any in their [@Injectable()](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) decorator

Element Injector Tree

Angular creates the Element Injector tree for the services to be provided at the element level like Components & Directives.

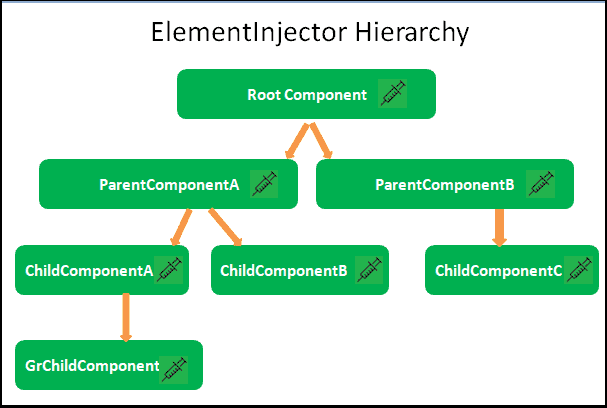
Angular creates the Element Injector tree when the application starts.

The Injector instance of the Root Component becomes the root Injector for the Element Injector tree. It gets the Providers from the provider’s property of the Root Component.

The Root Component acts as a parent to every element ( i.e. Component or Directives) we create. Each of those elements can contain child elements creating a tree of elements. The Angular creates an Injector for each of these elements creating a tree of Injectors.

Each Injector gets the list of Providers from the [@Directive()](https://www.tektutorialshub.com/angular/angular-directives/) or @[Component](https://www.tektutorialshub.com/angular/angular-component/)(). If the Providers array is empty, then Angular creates an empty Injector.

The Angular will destroy the Injector when Angular destroys the element.



Dependency Resolution

Angular resolves the dependency in two phases

First, resolve it using the Element Injector and its parents

If not found in the Element Injector, then resolve it against the Module Injector and its parents.

The components ask for the dependencies in the constructor using the token.

The search starts at the Injector associated with the component in the Element Injector tree. It uses the token to search for dependency in its Providers array. If it finds the provider, Injector checks to see if the instance of the service already exists. If exists then it injects it into the component else it creates a new instance of it. Then it injects it into the component.

Injector passes the request to the parent Injector in the Element Injector Hierarchy if it fails to find the provider. If the provider is found, the request returns the instance of the Provider. If not found then the request continues until the request reaches the topmost injector in the Element Injector tree.

The topmost Injector in the Element Injector tree belongs to the root component. If the dependency is not found, it does not throw the error but returns back.

Now, the search shifts to Module Injector Tree

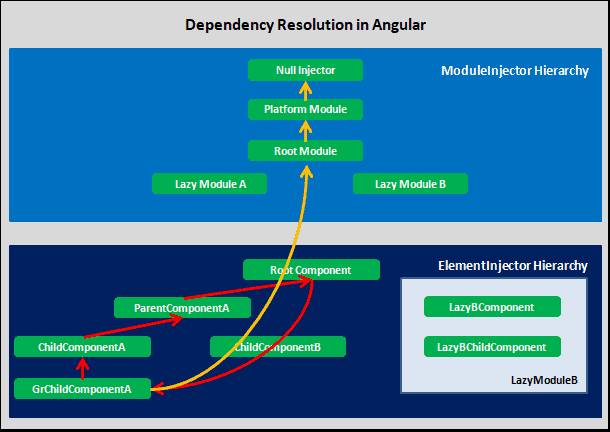
Search starts from the Injector associated with the module to which the element belongs. For the Root Modules & Eagerly loaded Module the search starts from the Root Module Injector. For components from the lazy loaded modules, the resolutions start from the Module Injector associated with Lazy Loaded Module.

The request continues until the request reaches the topmost injector in the Module Injector tree i.e. Null Injector. The Null Injector does not contain any providers. Its job is to throw No provider for Service error. But if we decorate the dependency with [@Optional decorator](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/), then it will return null instead of throwing an error.

Dependency Resolution in Picture

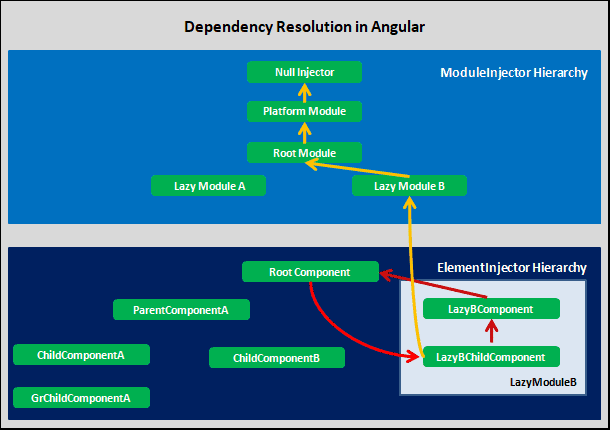
The following image shows how Angular Resolves the Dependency for Component in a Root Module.

The component (GrChildComponentA) asking for dependency is in the RootModule. The search starts in the Element Injector hierarchy starting with the Injector associated with the GrChildComponentA. The search moves up the Injector tree to reach the Root Injector, which belongs to the RootComponent. If the service not found, the search shifts to Module Injector Hierarchy starting from Root Module (because GrChildComponentA belongs to Root Module).



The following image shows how Angular Resolves the Dependency for Component in a Lazy Loaded Module

The component (LazyBChildComponent) asking for dependency is in the LazyModuleB. As you can see, once the Injectors from the Element Injector tree fail to provide the service, the request shifts to Module Injector Hierarchy. The starting point for the search is the Module to which LazyBChildComponent belongs i.e. LazyModuleB.



[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Notes on Dependency Resoultion

Module Injector Tree is not a parent of Element Injector Tree. Each Element can have the same parent Element Injector Tree, but a different parent in Module Injector Tree

No Injectors from Eager Modules. They share it with the Root Module.

Separate Injector for Lazy Loaded Modules in Module Ejector Tree

If two Eager Modules, provide the service for the same token, the module, which appears last in the imports array wins. For Example in the following imports, providers of Eager2Module overwrites the providers of the Eager1Module for the same token

|  |  |
| --- | --- |
| 1  2  3 | imports: [BrowserModule, FormsModule, Eager1Module, Eager2Module], |

If Eager Module & Root Module provide the service for the same token, then the Root module wins

Any service with providedIn value of root in the lazy loaded module, become part of the Root Module Injector.

To restrict service to the lazy loaded module, remove it from the providedIn and add it in the provider’s array of the Module.

The Services are singletons within the scope of an injector. When the injector gets a request for a particular service for the first time, it creates a new instance of the service. For all the subsequent requests, it will return the already created instance.

The Injectors are destroyed when Angular removes the associated Module or element.

Where you configure your services, will decide the service scope, service lifetime & bundle size.

You can use [Resolution Modifiers](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/) to modify the behavior of injectors. Refer to the [@Self, @SkipSelf & @Optional Decorators](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/) & [@Host Decorator in Angular](https://www.tektutorialshub.com/angular/host-decorator-in-angular/)

Example of hierarchical dependency injection

You can download the sample application from [Stackblitz](https://stackblitz.com/edit/angular-hierarchical-dependency-injection-example1?file=src/app/app.module.ts)

The App has a RootModule, EagerModule & LazyModule.

All the modules have one service each. Each generates a random number. We affix the number with the name of the service so that you will know the service has generated it.

The code from the AppService. Other services are almost identical

***app.service.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import { Injectable } from '@angular/core';    @Injectable()  export class AppService {    sharedValue: string;      constructor() {      console.log('Shared Service initialised');      this.sharedValue = 'App:' + Math.round(Math.random() \* 100);      console.log(this.sharedValue);    }      public getSharedValue() {      return this.sharedValue;    }  } |

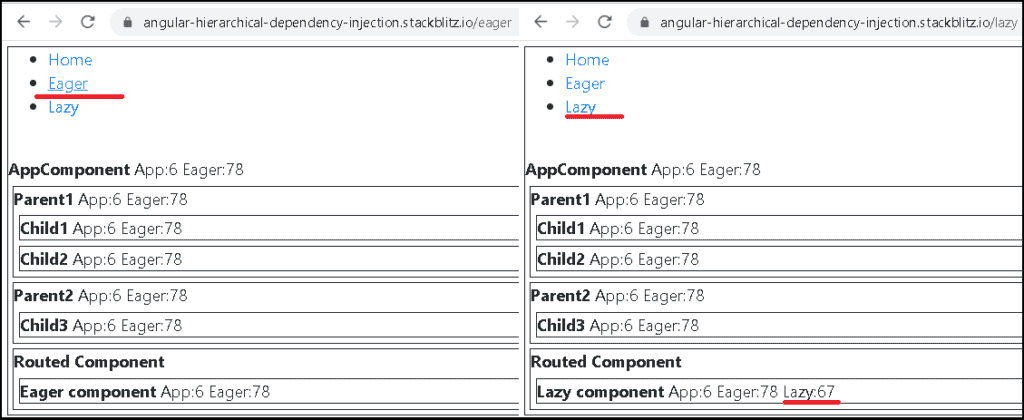
All Components inject all services and display the result. We use the [@Optional() decorator](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/). This will ensure that the Injector returns null instead of an error if the dependency not found.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | import { Component, Optional } from '@angular/core';  import { AppService } from './app.service';  import { EagerService } from './eager/eager.service';  import { LazyService } from './lazy/lazy.service';    @Component({    selector: 'parent1-component',    template: `      <div class="box">        <strong>Parent1</strong>        {{ appValue }}        {{ eagerValue }}        {{ lazyValue }}          <child1-component></child1-component>        <child2-component></child2-component>        </div>    `,    providers: []  })  export class Parent1Component {    appValue;    eagerValue;    lazyValue;      constructor(      @Optional() private appService: AppService,      @Optional() private eagerService: EagerService,      @Optional() private lazyService: LazyService    ) {        this.appValue = appService?.getSharedValue();      this.eagerValue = eagerService?.getSharedValue();      this.lazyValue = lazyService?.getSharedValue();    }  } |

Now, let us play around and see the effect of providing the services at various places.

Provide Services in the respective Modules

Use the Providers array to add the AppService, EagerService & LazyService in their respective NgModules ([Stackblitz](https://stackblitz.com/edit/angular-hierarchical-dependency-injection-example1?file=src/app/app.module.ts)).



AppService & EagerService returns the same value everywhere because they are available in the RootModule Injector.

LazyService is available only in the LazyModule Injector. Hence available only in the LazyComponent.

You can try out the following

Register EagerService in AppModule instead of EagerModule

Add providedIn: 'root' to both EagerService & AppService.

Both will result in a similar result

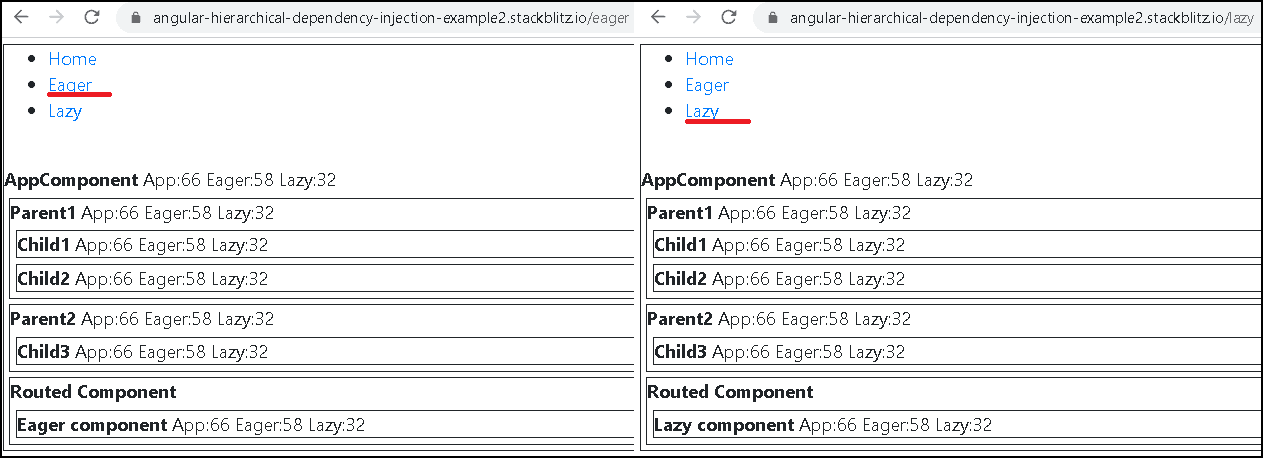
Using ProvidedIn in LazyService

Add the providedIn: 'root' for the LazyService. Now LazyService is added to Root Module Injector.

Remove LazyService from the Provider’s Array of LazyModule

|  |  |
| --- | --- |
| 1  2  3  4 | @Injectable({ providedIn: 'root' })  export class LazyService { |

Since the LazyService is now available Root Module Injector, it will be available across the application as a singleton ([Code](https://stackblitz.com/edit/angular-hierarchical-dependency-injection-example2?file=src/app/lazy/lazy.service.ts)).



You can try out the following

Keep the providedIn: 'root' for the LazyService.

Add LazyService to the Providers array of the LazyModule.

Now LazyService available at two Injectors. In RootModule Injector & in LazyModule Injector. The LazyComponent will use the service from the LazyModule Injector, while the rest of the App uses it from the RootModule Injector. Hence you get two different values

LazyService in AppComponent

Keep the providedIn: 'root' for the LazyService.

Add LazyService to the Providers array of the LazyModule.

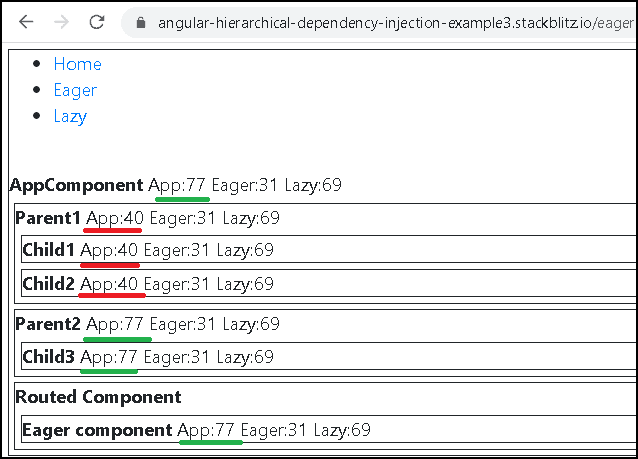
Add LazyService to the Providers array of the AppComponent.

Now LazyService available at three Injectors ([code](https://stackblitz.com/edit/angular-hierarchical-dependency-injection-example3?file=src/app/app.component.ts)). RootModule Injector, LazyModule Injector & AppComponent Injector. The AppComponent Injector is the root of the Element Injector Tree. All Components are children of the AppComponent. Hence all of them get the same value.

Services in a Component

Register the AppService in the Providers array of Parent1Component.

As you can see from the image below, the Parent1Component & all its child components gets their copy of AppService from the Parent1Component while rest of the App gets it from AppModule



How to Create a Singleton Service in Angular

There are two ways in which you can create a Singleton Service

Using the root option of the providedIn property. This works irrespective of your service is in an **eager module** or **lazy loaded module**. Using the providedIn  is the preferred way as it makes the service tree shakeable.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Injectable({    providedIn: 'root'  })  export class AppService {  ...  } |

The second option is to add it in the Providers array of @NgModule.

If the NgModule is **root module** or **eagerly loaded module**, then the AppService is available as a Singleton service to the entire application

But if the NgModule is **lazy-loaded module**, then AppService is available **only in that**[**Lazy loaded module**](https://www.tektutorialshub.com/angular/angular-lazy-loading/) and not outside of it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @NgModule({    imports: [],    declarations: [],    bootstrap: [],    providers: [AppService]     <===  })  export class AppModule {} |

Angular Singleton Service Example

Let us try to create the Singleton Service using an example from [StackBlitz](https://stackblitz.com/edit/angular-singleton-services-example-1?file=src/app/lazy/lazy.module.ts)

The app has three modules. AppModule, EagerModule & LazyModule.

All the modules contain a random generation service. AppService, LazyService, & EagerService. The code for the AppService is as shown below. The LazyService & EagerService also has the same code. We just prefixed the random number with the service name to distinguish it from other services.

***app.service.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | import { Injectable } from '@angular/core';    @Injectable()  export class AppService {    private \_randomNo = '';      constructor() {      console.log('AppService Constructed');      this.\_randomNo = 'App ' + Math.floor(Math.random() \* 1000 + 1);    }      get RandomNo() {      return this.\_randomNo;    }  } |

We have four components. AppComponent & HelloComponent from the Root Module, EagerComponent from EagerModule and LazyComponent from LazyModule

The following code is from the HelloComponent.

***hello.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | import { Component, Input, Optional } from '@angular/core';  import { AppService } from './app.service';  import { EagerService } from './eager/eager.service';  import { LazyService } from './lazy/lazy.service';    @Component({    selector: 'hello',    providers:[],    template: `      Hello Works {{ randomApp }} {{ randomEager }} {{ randomLazy }}    `,    styles: [      `        h1 {          font-family: Lato;        }      `    ]  })  export class HelloComponent {    randomApp = 'App : Not defined';    randomEager = 'Eager : Not defined';    randomLazy = 'Lazy : Not defined';      constructor(      @Optional() private appService: AppService,      @Optional() private eagerService: EagerService,      @Optional() private laztyService: LazyService    ) {      if (appService) this.randomApp = this.appService.RandomNo;      if (eagerService) this.randomEager = this.eagerService.RandomNo;      if (laztyService) this.randomLazy = this.laztyService.RandomNo;    }      ngOnInit() {}  } |

We have injected all three services into HelloComponent. The Optional Decorator ensures that if the service not available then the Angular returns **null**instead of throwing an error.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | constructor(      @Optional() private appService: AppService,      @Optional() private eagerService: EagerService,      @Optional() private laztyService: LazyService    ) |

All the other components also have similar codes. The AppComponent has a navigation menu.

Initially, we start off by removing the providedIn and also making the provider’s array empty. This will make all the component’s display Not defined. No errors thrown as we are using the [optional decorator](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/)

Service in the root module

The AppService is in the root module.

Add it to the Providers array of the @NgModul of the AppModule.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @NgModule({    imports: [BrowserModule,FormsModule,RouterModule.forRoot(routes),EagerModule],    declarations: [AppComponent, HelloComponent],    bootstrap: [AppComponent],    providers: [AppService]  })  export class AppModule {} |

You can see that all components display the same value for the Random No (including those from EagerModule & LazyModule ). You can also see the AppService Constructed in the console only once.

Another way to achieve this is to add the providedIn: 'root' in the @Injectable decorator in the AppService

|  |  |
| --- | --- |
| 1  2  3  4 | @Injectable({ providedIn: 'root' })  export class AppService { |

Service in the eagerly loaded module

The services in the eagerly loaded module can be made singleton in the same way as in the root module.

You can add it in the providedIn :'root' with the @Injectable decorator

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Injectable({ providedIn: 'root' })  export class EagerService { |

Or add it to the providers array of the NgModule decorator of the EagerModule.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @NgModule({    imports: [CommonModule, RouterModule.forChild(routes)],    declarations: [EagerComponent],    providers: [EagerService]  })  export class EagerModule {} |

Service in the lazy loaded module

Adding the LazyService in the providers array of the LazyModule. But this makes the LazyService available only in the LazyModule and not outside of it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @NgModule({    imports: [CommonModule, RouterModule.forChild(routes)],    declarations: [LazyComponent],    providers: [LazyService]  })  export class LazyModule {} |

The only way to achieve it by using the providedIn : 'root' in the Injectable of the LazyService

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Injectable({ providedIn: 'root' })  export class LazyService { |

ProvidedIn root

Use the ProvidedIn root option, when you want to register the application-level singleton service.

The root option registers the service in the Root Module Injector of the Module Injector tree. This will make it available to the entire application. This is irrespective of whether the service is lazy loaded or eagerly loaded.

If it is never used it will not be added in the final build (tree shaking)

Lazy Loaded Service

Using ProvidedIn root adds it to the Root Module Injector and makes it application-wide singleton

Registering the service in a @NgModule will make it available in that Module only (Singleton within the Module Scope)

Using both makes it singleton for the rest of the application, while it creates a separate instance for that Module

ProvidedIn any

Use ProvidedIn: any when you want every lazy-loaded module to get its own instance of the service.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Injectable({     providedIn: 'platform'  })  export class SomeService{  } |

The eagerly loaded modules always share the instance provided by the Root Module Injector. Hence this will not have any effect on them.

ProvidedIn platform

As per the [documents](https://next.angular.io/api/core/Injectable)

A special singleton platform injector shared by all applications on the page.

the platform allows us to add the service to the Providers of the Platform Injector. If you recall, the Platform Injector is the parent of the Root Module Injector in the [Module Injector tree](https://www.tektutorialshub.com/angular/how-dependency-injection-resolution-works-in-angular/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Injectable({     providedIn: 'platform'  })  export class SomeService{  } |

This is useful if you have multiple Angular Apps running on a single page.

This is a useful option if you are using Angular Elements, where they can share a single instance of service between them.

How Angular DI Framework Resolves Dependencies

When a component asks for Dependency, the DI Framework resolves it in two phases.

In the first phase, it starts to look for the Dependency in the current component’s ElementInjector. If it does not provide the Dependency, it will look in the Parent Components ElementInjector. The Request bubbles up until it finds an injector that provides the service or reaches the root ElementInjector.

If ElementInjector does not satisfy the request, Angular looks for the Dependency in the ModuleInjector hierarchy. If Angular still doesn’t find the provider, it throws an error.

The older versions of the Angular created only one Injector tree. But in the later versions, the tree was split into two trees. One is ElementInjector for elements (components, directives & pipes etc) and the other one is ModuleInjector for [Angular Modules](https://www.tektutorialshub.com/angular/angular-modules/).

@Self, @SkipSelf & @Optional Example

We have created an example project in Angular to explain the @Self, @SkipSelf, & @Optional. You can find the Source Code in [StackBlitz](https://stackblitz.com/edit/angular-self-optional-skipself-example?file=src/app/app.component.ts).

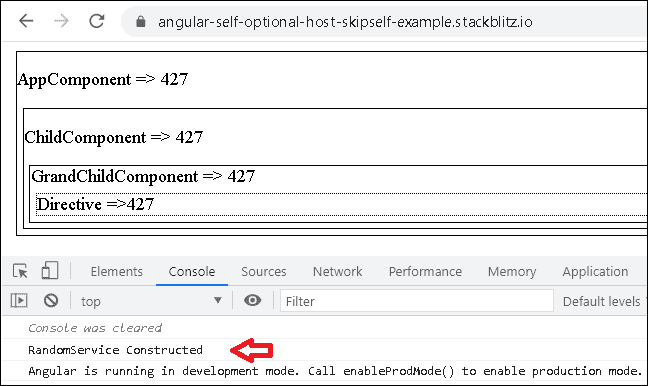
The Code contains a RandomService, which generates a Random Number when initialized. The [Angular Service](https://www.tektutorialshub.com/angular/angular-services/) is added to the Providers array of the AppModule. We can inject this service anywhere in our Application.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Injectable({    providedIn: "root"  })  export class RandomService { |

The project contains three [Angular Components](https://www.tektutorialshub.com/angular/angular-component/) (AppComponent, ChildComponent & GrandChildComponent) all inject the RandomService and displays the Random Number from the Service.

We also have testDirective, which we include in the template of GrandChildComponent. It also displays the Random Number from the Service.

Ensure that the Providers array is empty in all components & directives. Run the App. Angular creates only one instance of the RandomService. That is why all the components and directives show the same number.



Now, let us check how we can modify the above behavior with @Self, @SkipSelf, & @Optional.

First let us start with @Self

@Self

The @Self decorator instructs Angular to look for the dependency only in the local injector. The local injector is the injector that is part of the current component or directive.

Open the GrandChildComponent and add the @Self() on randomService as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | @Component({    selector: "my-grandChild",    template: `      <div class="box">        GrandChildComponent => {{ randomNo }}        <div class="dirbox" testDirective>fdf</div>      </div>    `,    providers: []  })  export class GrandChildComponent {    randomNo;    constructor(@Self() private randomService: RandomService) {      this.randomNo = randomService.RandomNo;    }  } |

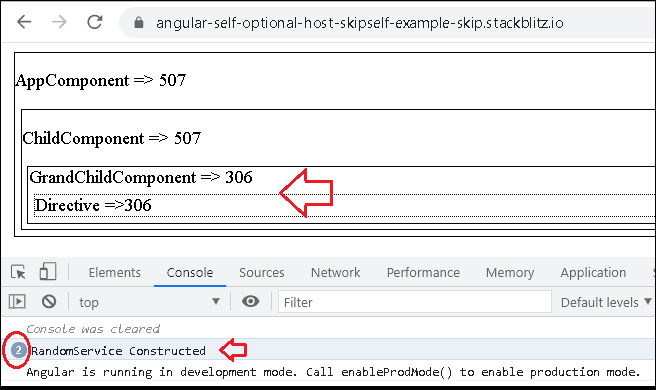
This forces the Angular DI Framework to look for the Dependency attached to the current Component. Since it does find one it will throw the error

**Error: NG0201: No provider for RandomService found in NodeInjector**

Add the RandomService to the providers array of the GrandChildComponent and the error goes away.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | @Component({    selector: "my-grandChild",    template: `      <div class="box">        GrandChildComponent => {{ randomNo }}        <div class="dirbox" testDirective>fdf</div>      </div>    `,    providers: [RandomService]  })  export class GrandChildComponent {    randomNo;    constructor(@Self() private randomService: RandomService) {      this.randomNo = randomService.RandomNo;    }  } |

As you can see from the image Angular creates two instances of RandomService. One from the AppModule and another from the GrandChildComponent. Also, note that testDirective picks up the RandomService provided from the GrandChildComponent and not from the AppModule



@SkipSelf

The @SkipSelf decorator instructs Angular to look for the dependency in the Parent Injector and upwards.

It tells [Angular](https://www.tektutorialshub.com/angular-tutorial/) not to look for the injector in the local injector, but start from the Parent. You can think of this decorator as the opposite of the @Self

Open the GrandChildComponent again. Add the SkipSelf instead of Self decorator.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import { Component, SkipSelf, Self, Optional, Host } from "@angular/core";  import { RandomService } from "./random-service";    @Component({    selector: "my-grandChild",    template: `      <div class="box">        GrandChildComponent => {{ randomNo }}        <div class="dirbox" testDirective>fdf</div>      </div>    `,    providers: [RandomService]  })  export class GrandChildComponent {    randomNo;    constructor(@SkipSelf() private randomService: RandomService) {      this.randomNo = randomService.RandomNo;    }  } |

As you can see from the image, the GrandChildComponent, picks up RandomService instance provided by the Module and not the one provided by itself.

But, the testDirective still picks up the RandomService provided by the GrandChildComponent.



@Optional

Optional marks the dependency as Optional. If the dependency is not found, then it returns null instead of throwing an error

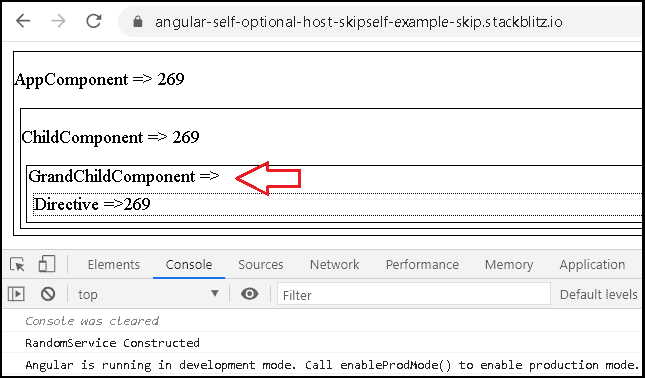
In the GrandChildComponent remove the RandomService from the Providers Array and add the @Self decorator. You will instantly receive the error “No provider for RandomService found in NodeInjector“.

Add the @Optional decorator along with the @Self. Now, the dependency injection will return null instead of an error.

Also, remember to add the ? in randomService?, else you will get the “Cannot read property ‘RandomNo’ of null” error.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import { Component, SkipSelf, Self, Optional, Host } from "@angular/core";  import { RandomService } from "./random-service";    @Component({    selector: "my-grandChild",    template: `      <div class="box">        GrandChildComponent => {{ randomNo }}        <div class="dirbox" testDirective>fdf</div>      </div>    `,    providers: []  })  export class GrandChildComponent {    randomNo;    constructor(@Optional() @Self() private randomService: RandomService) {      this.randomNo = randomService?.RandomNo;    }  } |

As you can see in the image, GrandChildComponent does not receive any values, while testDirective picks up the RandomService provided by the AppModule



Source Code

***app.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import { Component, VERSION } from "@angular/core";  import { RandomService } from "./random-service";    @Component({    selector: "my-app",    providers: [],    viewProviders: [],    template: `      <div class="box">        <p>AppComponent => {{ randomNo }}</p>        <my-child></my-child>      </div>    `  })  export class AppComponent {    randomNo;    constructor(private randomService: RandomService) {      this.randomNo = randomService.RandomNo;    }  } |

***child.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import { Component, SkipSelf, Self, Optional, Host } from "@angular/core";  import { RandomService } from "./random-service";    @Component({    selector: "my-child",    providers: [],    viewProviders: [],    template: `      <div class="box">        <p>ChildComponent => {{ randomNo }}</p>          <my-grandChild></my-grandChild>      </div>    `  })  export class ChildComponent {    randomNo;    constructor(private randomService: RandomService) {      this.randomNo = randomService.RandomNo;    }  } |

***grand-child.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import { Component, SkipSelf, Self, Optional, Host } from "@angular/core";  import { RandomService } from "./random-service";    @Component({    selector: "my-grandChild",    template: `      <div class="box">        GrandChildComponent => {{ randomNo }}        <div class="dirbox" testDirective>fdf</div>      </div>    `,    providers: []  })  export class GrandChildComponent {    randomNo;    constructor(private randomService: RandomService) {      this.randomNo = randomService.RandomNo;    }  } |

***test-directive.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | import {    Directive,    ElementRef,    Input,    OnInit,    SkipSelf,    Self,    Optional,    Host  } from "@angular/core";  import { RandomService } from "./random-service";    @Directive({    selector: "[testDirective]",    providers: []  })  export class testDirective implements OnInit {    @Input() ttClass: string;      constructor(private el: ElementRef, private randomService: RandomService) {}      ngOnInit() {      this.el.nativeElement.innerHTML =        "Directive =>" + this.randomService.RandomNo;    }  } |

***random-service.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import { Injectable } from "@angular/core";    @Injectable({    providedIn: "root"  })  export class RandomService {    private \_randomNo = 0;      constructor() {      console.log("RandomService Constructed");      this.\_randomNo = Math.floor(Math.random() \* 1000 + 1);    }      get RandomNo() {      return this.\_randomNo;    }  } |

***styles.css***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | .box {    margin: 5px;    border: 1px;    border-style: solid;  }    .dirbox {    margin: 5px;    border: 1px;    border-style: dotted;  } |

***app.module.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import { NgModule } from "@angular/core";  import { BrowserModule } from "@angular/platform-browser";  import { FormsModule } from "@angular/forms";    import { AppComponent } from "./app.component";  import { ChildComponent } from "./child.component";  import { GrandChildComponent } from "./grand-child.component";  import { testDirective } from "./test-directive";    @NgModule({    imports: [BrowserModule, FormsModule],    declarations: [      AppComponent,      ChildComponent,      GrandChildComponent,      testDirective    ],    bootstrap: [AppComponent]  })  export class AppModule {} |

@Host

The definition of @Host from the [Angular Docs](https://angular.io/api/core/Host)

*Parameter decorator on a view-provider parameter of a class constructor that tells the DI framework to resolve the view by checking injectors of child elements, and stop when reaching the host element of the current component.*

The definition is a little confusing. So let us try to simplify it a bit.

The @Host property searches for the dependency inside the **component’s template only**.

It starts with the current Injector and continues to search in the Injector hierarchy **until it reaches the host element** of the **current component**.

It **does not search** for the dependency in the Providers of the **host element**.

But **it does search** in the ViewProviders of the **host element**.

Module injector is never searched in the case of @Host flag

It looks somewhat similar to **@Self**. But [@Self](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/#self) only checks in the Injector of the **current component**.

But the @Host checks for the dependency in the **current template**.

@Host Example

Let us look at the Angular Project from [StackBlitz](https://stackblitz.com/edit/angular-host-decorator-example-1?file=src/app/test-directive.ts). It is the same Project from in the tutorial [@Self, @SkipSelf & @Optional Decorators](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/).

It has three Components. AppComponent, ChildComponent & GrandChildComponent. One testDirective directive. And RandomService, which generates the Random number, when instantiated.

All the components & directives display the random number from the RandomService.

Make the Providers empty in all components & directives so that RandomService is always provided by the AppModule. Run the App and you should see the same number displayed by all the components & directives. Also, you will RandomService Constructed message only once in the console window indicating that a single instance of the service is created.

Parent & Child Components with @Host Decorator

In the template of [ChildComponent](https://stackblitz.com/edit/angular-host-decorator-example-1?file=src/app/child.component.ts), we have added GrandChildComponent.

***child.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @Component({    selector: "my-child",    providers: [],    viewProviders: [],    template: `      <div class="box">        <p>ChildComponent => {{ randomNo }}</p>          <my-grandChild></my-grandChild>      </div>    `  })  export class ChildComponent { |

Now, go to the GrandChildComponent and add the @Host decorator on the randomService

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | export class GrandChildComponent {    randomNo;    constructor(@Host() private randomService: RandomService) {      this.randomNo = randomService?.RandomNo;    }  } |

Immidetaly you will see the message.

**No provider for RandomService found in NodeInjector**

The GrandChildComponent is part of the following template of ChildComponent. This Template is hosted inside the ChildComponent. Hence ChildComponent is the host component.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <div class="box">        <p>ChildComponent => {{ randomNo }}</p>          <my-grandChild></my-grandChild>      </div> |

@Host first looks for the Dependency in the template. It starts with the Providers of the GrandChildComponent.

Next, it will move to ChildComponent, which is the Host Component. Here it will only look in the ViewProviders and not in Providers array.

Hence there are two ways in which you can remove this error

Add the RandomService to the Providers of the GrandChildComponent. Because that is where DI looks first for dependency.

The second option is to add RandomService to viewProviders array of the ChildComponent.

Directives & @Host Decorator

The Components are nothing but Directives with a View. Hence Directives works in the same way

Goto the testDirective and add the @host decorator on randomService. The error immediately shows up in the console window. No provider for *RandomService* found in *NodeInjector*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | export class testDirective implements OnInit {    constructor(      private el: ElementRef,      @Host() private randomService: RandomService    ) {} |

The testDirective is part of this template from GrandChildComponent. The Host of the template is GrandChildComponent.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | template: `      <div class="box">        GrandChildComponent => {{ randomNo }}          <div class="dirbox" testDirective>fdf</div>        </div>    `, |

@Host will look for the Dependency in this order

Providers array of testDirective

ViewProviders of the GrandChildComponent

Hence adding the *RandomService* in any of those places will resolve the error.

Multiple Directives

Consider the following [StackBlitz example](https://stackblitz.com/edit/angular-host-decorator-example-2?file=src/app/child.component.ts). Here we have three directives. All of them has a dependency on randomService.

Now go to cDirective and add @Host on RandomService.

To understand how @Host resolves the Dependency, take a look at the template, where we have used cDirective

The template from the ChildComponent (Hence our Host Component). The aDir, bDir & cDir all part of this template. As mentioned earlier @Host only looks at this template while resolving the dependency.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <div class="box">        <p>ChildComponent => {{ randomNo }}</p>          <div aDir>          <div bDir>            <div cDir></div>          </div>        </div>      </div> |

@Host will look for the Dependency in this order

Providers of cDirective

Providers of bDirective

Providers of aDirective

ViewProviders of ChildComponent

Put RandomService in any of those place and error will go away

What if we add @Host in bDirective instead of cDirective. The checking will be done in the following order. cDirective is not checked as it is the child of bDirective

Providers of bDirective

Providers of aDirective

ViewProviders of ChildComponen

Content Projection & @Host Decorator

Now, look at the [StackBlitz](https://stackblitz.com/edit/angular-host-content-projection-example?file=src/app/app.component.ts) example. The App has three components (AppComponent, ChildComponent & GrandChildComponent) and a service (RandomService). Here we are making use of [Content Projection](https://www.tektutorialshub.com/angular/ng-content-content-projection-in-angular/) to project the GrandChildComponent into the ChildComponent.

Open the GrandChildComponent and add the @Host decorator on randomService.

Like in our previous examples, let us go and look at the template where you will find the GrandChildComponent. You will find it in template of the AppComponent.

***app.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <div class="box">        <p>AppComponent => {{ randomNo }}</p>        <my-child>          <my-grandChild></my-grandChild>        </my-child>      </div> |

Hence the above is our template and the host is AppComponent.

@Host will look for the Dependency in this order

Providers of GrandChildComponent

Providers of ChildComponent

ViewProviders of AppComponent

ViewProviders Example

Look at the example app at [stackblitz](https://stackblitz.com/edit/angular-view-providers-example1?file=src/app/child.component.ts)

The app has RandomService, which generates a random number when initialized.

***random-service.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import { Injectable } from "@angular/core";    @Injectable({    providedIn: "root"  })  export class RandomService {    private \_randomNo = 0;      constructor() {      console.log("RandomService Constructed");      this.\_randomNo = Math.floor(Math.random() \* 1000);    }      get RandomNo() {      return this.\_randomNo;    }  } |

ChildComponent displays the random no from the RandomService.

It also has [ng-content](https://www.tektutorialshub.com/angular/ng-content-content-projection-in-angular/), where the parent can inject content. The Parent component is going to inject the GrandChildComponent here.

ChildComponent also displays the GrandChildComponent as View

***child.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import { Component, SkipSelf, Self, Optional, Host } from '@angular/core';  import { RandomService } from './random-service';    @Component({    selector: 'my-child',    providers: [],    viewProviders: [],    template: `      <div class="box">        <p>ChildComponent => {{ randomNo }}</p>        <ng-content> </ng-content>          <strong>View Child</strong>        <my-grandChild></my-grandChild>      </div>    `  })  export class ChildComponent {    randomNo;    constructor(private randomService: RandomService) {      this.randomNo = randomService.RandomNo;    }  } |

GrandChildComponent just displays the random no from the RandomService. We use [@Optional](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/)() decorator ensures no error is thrown if the dependency is not found.

***grand-child.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import { Component, SkipSelf, Self, Optional, Host } from "@angular/core";  import { RandomService } from "./random-service";    @Component({    selector: "my-grandChild",    template: `      <div class="box">        GrandChildComponent => {{ randomNo }}      </div>    `,    providers: [],    viewProviders: [],  })  export class GrandChildComponent {    randomNo;    constructor(@Optional() private randomService: RandomService) {      this.randomNo = randomService?.RandomNo;    }  } |

In the AppComponent, we display the ChildComponent. We also project the GrandChildComponent inside the ChildComponent using [projected content](https://www.tektutorialshub.com/angular/ng-content-content-projection-in-angular/).

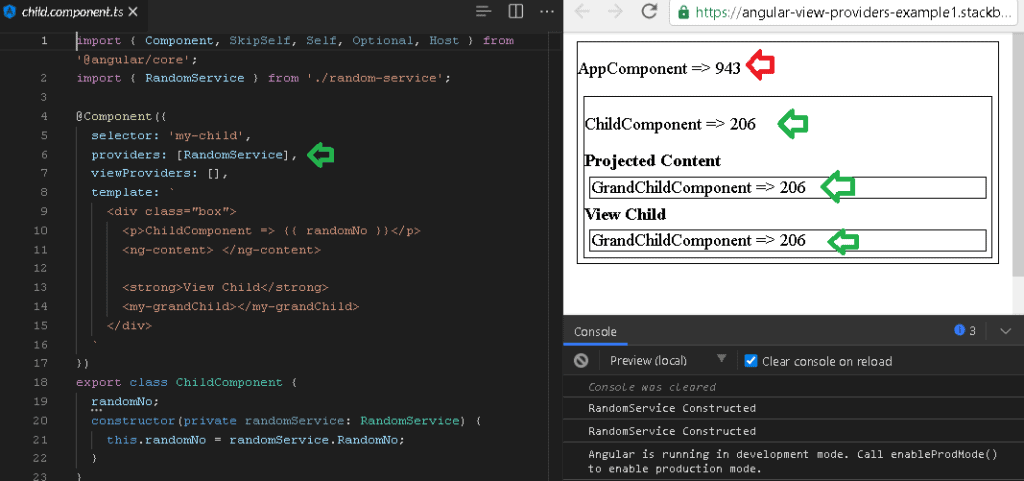
***app.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | import { Component, VERSION } from '@angular/core';  import { RandomService } from './random-service';    @Component({    selector: 'my-app',    providers: [],    viewProviders: [],    template: `      <div class="box">      <div class="box">        <p>AppComponent => {{ randomNo }}</p>        <my-child>          <strong>Projected Content</strong>          <my-grandChild></my-grandChild>        </my-child>      </div>        `  })  export class AppComponent {    randomNo;    constructor(private randomService: RandomService) {      this.randomNo = randomService.RandomNo;    }  } |

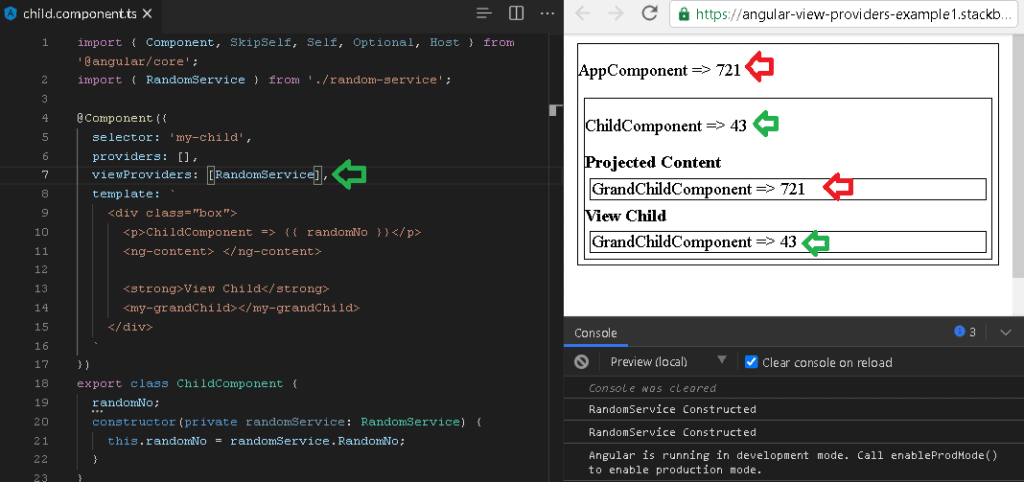
Run the app and you will see all the components receive the same value for the random no. Because the RandomService is provided from the root module injector.

Also, note that ChildComponent displays the GrandChildComponent twice. Once as a Content Child (Projected Content) and also a View Child.

Add the RandomService to the Providers array of the ChildComponent. As you can see from the image below, ChildComponent and all its children (content children and view children) gets the instance of the RandomService provided by the ChildComponent.



Now, move the RandomService from Providers to ViewProviders. As you can see in the image below View Child still get the RandomService from ChildComponent, but the Projected Content does not. Projected Content gets the service from the Root Module.



The providers allow all children to use the services. While the viewProviders limit it to children other than projected content.

Use case for ViewProviders

This is useful when you develop libraries.

For Example, you have made some-great-comp , which user will be using it to project their user-component into it.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <some-great-comp>     <user-component></user-component>  </some-great-comp> |

Here you do not want services that you used in your some-great-comp interfere with the user-component. Hence you provide your services in the ViewProviders.

# Angular Forms Validation

Validators in Reactive Forms

What is a Validator

A Validator is a function that checks the instance of [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/), [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) or a [FormArray](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/) and returns a list of errors. If the Validator returns a null means that validation has passed

How to add a Validator to Reactive Forms

We configure the validators as the second and third argument to the [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/), [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) or [FormArray](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/) in the component class. The second argument is a collection of **sync validators** and the third argument is a collection of an **async validators**.

sync validators runs validations and returns immediately. They either return a list of errors or null if no errors found.

async validators: returns a **Promise** or **Observable**. They either return a list of errors or null if no errors are found.

Built-in Validators

The Angular ReactiveForms Module provides several Built-in validators out of the box. They are required, minlength, maxlength & pattern etc.

Reactive Forms Validation Example

W learned [how to create Angular Reactive Forms](https://www.tektutorialshub.com/angular/angular-reactive-forms/) in the previous tutorial. We will now add some of the built-in validators to that example.

Model

Here is the contactForm model from the previous tutorial.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | contactForm = new FormGroup({      firstname: new FormControl(''),      lastname: new FormControl(''),      email:new FormControl(''),      gender: new FormControl(''),      isMarried: new FormControl(''),      country: new FormControl(''),      address:new FormGroup({        city: new FormControl(''),        street: new FormControl(''),        pincode:new FormControl('')      })    }) |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Disabling the Browser validation

First, we need to disable browser validator by adding the [novalidate](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form) attribute to the *<form>* element as shown below. If this attribute is present then the form is not validated by the built-in HTML5 validation when submitted.

|  |  |
| --- | --- |
| 1  2  3 | <form [formGroup]="contactForm" (ngSubmit)="onSubmit()" novalidate> |

Adding in Built-in Validators

The mentioned earlier, the Angular has provided several built-in validators out of the box.

Required Validator

The required validator is a sync validator, which returns true only if the formcontrol has a non-empty value entered. The second argument of the FormControl takes the Sync Validator.

|  |  |
| --- | --- |
| 1  2  3 | firstname: new FormControl('',[Validators.required]), |

Minlength Validator

Minlength validator requires the control value must not have less number of characters than the value specified in the validator.

For Example, minlength validator ensures that the firstname value has at least 10 characters.

|  |  |
| --- | --- |
| 1  2  3 | firstname: new FormControl('',[Validators.required,Validators.minLength(10)]), |

Maxlength Validator

This Validator requires that the number of characters must not exceed the value specified in the validator.

|  |  |
| --- | --- |
| 1  2  3 | lastname: new FormControl('',[Validators.maxLength(15)]), |

Pattern Validator

This Validator requires that the control value must match the regex pattern provided in the attribute. For example, the pattern ^[a-zA-Z]+$ ensures that the only letters are allowed (even spaces are not allowed). Let us apply this pattern to the lastName

|  |  |
| --- | --- |
| 1  2  3 | lastname: new FormControl('',[Validators.maxLength(15), Validators.pattern("^[a-zA-Z]+$")]), |

Email Validator

This Validator requires that the control value must be a valid email address. We apply this to the email field

|  |  |
| --- | --- |
| 1  2  3 | email:new FormControl('',[Validators.email,Validators.required]), |

After adding all the validators, our final contactForm will look like this.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | contactForm = new FormGroup({      firstname: new FormControl('',[Validators.required,Validators.minLength(10)]),      lastname: new FormControl('',[Validators.required, Validators.maxLength(15), Validators.pattern("^[a-zA-Z]+$")]),      email:new FormControl('',[Validators.email,Validators.required]),      gender: new FormControl('',[Validators.required]),      isMarried: new FormControl('',[Validators.required]),      country: new FormControl('',[Validators.required]),      address:new FormGroup({        city: new FormControl('',[Validators.required]),        street: new FormControl('',[Validators.required]),        pincode:new FormControl('',[Validators.required])      })    }) |

Ad by Valueimpression

Disable Submit button

We have successfully added the validators. Now, we need to disable the submit button if our form is not valid.

The Angular Forms API exposes the state of the forms through the FormGroup, FormControl & [FormArray](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/) instances. The FormGroup control has a property valid, which is set to true if all of its child controls are valid.

The contactForm represents the top-level FormGroup. We use it to set the disabled attribute of the submit button.

|  |  |
| --- | --- |
| 1  2  3 | <button type="submit" [disabled]="!contactForm.valid">Submit</button> |

Displaying the Validation/Error messages

We need to provide a short and meaningful error message to the user. We do that by using the error object returned by the FormControl instance

Every form element has a FormControl instance associated with it. It exposes the state of form element like valid, dirty, touched etc.

There are two ways in which you can get the reference to the FormControl.

One way is to use the contactForm variable. We can use contactForm.controls.firstname.valid to find out if the firstname is valid.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <div      \*ngIf="!contactForm.controls.firstname?.valid && (contactForm.controls.firstname?.dirty      ||contactForm.controls.firstname?.touched)">        First Name is not valid    </div> |

The other way to is to define getter function for each FormControl instance in the component class.

|  |  |
| --- | --- |
| 1  2  3  4  5 | get firstname() {     return this.contactForm.get('firstname');  } |

and then use it in the template as follows

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div \*ngIf="!firstname.valid && (firstname.dirty ||firstname.touched)">        First Name is not valid    </div> |

Dirty & touched

Apart from checking valid we are also checking for the dirty & touched. Because we do not want the application to display the error when the form is displayed for the first time. We want to display errors only after the user has attempted to change the value. The dirty & touched properties help us do that.

dirty: A control is dirty if the user has changed the value in the UI.  
touched: A control is touched if the user has triggered a blur event on it.

Error message

The error message “First Name is not valid ” is not helpful. The firstname has two validators. required and minlength

Any errors generated by the failing validation is updated in the errors object. The errors object returns the error object or null if there are no errors.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <div      \*ngIf="!firstname?.valid && (firstname?.dirty ||firstname?.touched)">      <div [hidden]="!firstname.errors.required">        First Name is required      </div>      <div [hidden]="!firstname.errors.minlength">        Min Length is 10      </div>    </div> |

Final Code

**app.component.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109 | import { Component, ViewChild, ElementRef } from '@angular/core';  import { FormGroup, FormControl, Validators } from '@angular/forms'      @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'Angular Reactive forms';        contactForm = new FormGroup({      firstname: new FormControl('',[Validators.required,Validators.minLength(10)]),      lastname: new FormControl('',[Validators.required, Validators.maxLength(15), Validators.pattern("^[a-zA-Z]+$")]),      email:new FormControl('',[Validators.email,Validators.required]),      gender: new FormControl('',[Validators.required]),      isMarried: new FormControl('',[Validators.required]),      country: new FormControl('',[Validators.required]),      address:new FormGroup({        city: new FormControl('',[Validators.required]),        street: new FormControl('',[Validators.required]),        pincode:new FormControl('',[Validators.required])      })    })      get firstname() {      return this.contactForm.get('firstname');    }      get lastname() {      return this.contactForm.get('lastname');    }      get email() {      return this.contactForm.get('email');    }      get gender() {      return this.contactForm.get('gender');    }      get isMarried() {      return this.contactForm.get('isMarried');    }      get country() {      return this.contactForm.get('country');    }      get city() {      return this.contactForm.get("address").get('city');    }      get street() {      return this.contactForm.get("address").get('street');    }      get pincode() {      return this.contactForm.get("address").get('pincode');    }        countryList: country[] = [      new country("1", "India"),      new country('2', 'USA'),      new country('3', 'England')    ];            onSubmit() {      console.log(this.contactForm.value);    }        }      export class contact {    firstname:string;    lastname:string;    gender:string;    isMarried:boolean;    country:string;    address: {      city:string;      street:string;      pincode:string;    }  }      export class country {    id: string;    name: string;      constructor(id: string, name: string) {      this.id = id;      this.name = name;    }  } |

**app.component.html**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152 | <form [formGroup]="contactForm" (ngSubmit)="onSubmit()" novalidate>      <p>      <label for="firstname">First Name </label>      <input type="text" id="firstname" name="firstname" formControlName="firstname">    </p>      <div      \*ngIf="!firstname?.valid && (firstname?.dirty ||firstname?.touched)">      <div [hidden]="!firstname.errors.required">        First Name is required      </div>      <div [hidden]="!firstname.errors.minlength">        Min Length is 10      </div>    </div>      <p>      <label for="lastname">Last Name </label>      <input type="text" id="lastname" name="lastname" formControlName="lastname">    </p>      <div \*ngIf="!lastname.valid && (lastname.dirty ||lastname.touched)">      <div [hidden]="!lastname.errors.pattern">        Only characters are allowed      </div>      <div [hidden]="!lastname.errors.maxLength">        Max length allowed is {{lastname.errors.maxlength?.requiredLength}}      </div>      <div [hidden]="!lastname.errors.required">        Last Name is required      </div>    </div>      <p>      <label for="email">Email </label>      <input type="text" id="email" name="email" formControlName="email">    </p>    <div \*ngIf="!email.valid && (email.dirty ||email.touched)">      <div [hidden]="!email.errors.required">        email is required      </div>      <div [hidden]="!email.errors.email">        invalid email id      </div>    </div>        <p>      <label for="gender">Geneder </label>      <input type="radio" value="male" id="gender" name="gender" formControlName="gender"> Male      <input type="radio" value="female" id="gender" name="gender" formControlName="gender"> Female    </p>    <div \*ngIf="!gender.valid && (gender.dirty ||gender.touched)">      <div [hidden]="!gender.errors.required">        gender is required      </div>    </div>      <p>      <label for="isMarried">Married </label>      <input type="checkbox" id="isMarried" name="isMarried" formControlName="isMarried">    </p>    <div \*ngIf="!isMarried.valid && (isMarried.dirty ||isMarried.touched)">      <div [hidden]="!isMarried.errors.required">        isMarried is required      </div>    </div>        <p>      <label for="country">country </label>      <select id="country" name="country" formControlName="country">        <option [ngValue]="c.id" \*ngFor="let c of countryList">          {{c.name}}        </option>      </select>    </p>    <div \*ngIf="!country.valid && (country.dirty ||country.touched)">      <div [hidden]="!country.errors.required">        country is required      </div>    </div>          <div formGroupName="address">        <div class="form-group">        <label for="city">City</label>        <input type="text" class="form-control" name="city" formControlName="city">      </div>      <div \*ngIf="!city.valid && (city.dirty ||city.touched)">        <div [hidden]="!city.errors.required">          city is required        </div>      </div>          <div class="form-group">        <label for="street">Street</label>        <input type="text" class="form-control" name="street" formControlName="street">      </div>      <div \*ngIf="!street.valid && (street.dirty ||street.touched)">        <div [hidden]="!street.errors.required">          street is required        </div>      </div>        <div class="form-group">        <label for="pincode">Pin Code</label>        <input type="text" class="form-control" name="pincode" formControlName="pincode">      </div>      <div \*ngIf="!pincode.valid && (pincode.dirty ||pincode.touched)">        <div [hidden]="!pincode.errors.required">          pincode is required        </div>      </div>      </div>      <p>{{contactForm.valid}} </p>      <p>      <button type="submit" [disabled]="!contactForm.valid">Submit</button>    </p>      </form>          <!-- <div ngModelGroup="address">      <p>      <label for="city">City</label>      <input type="text" id="city" name="city">    </p>      <p>      <label for="street">Street</label>      <input type="text" id="street" name="street">    </p>    <p>      <label for="pincode">Pin Code</label>      <input type="text" id="pincode" name="pincode">    </p>    </div> --> |

Custom Validator in Angular Reactive Form

Built-in validators are useful but do not cover all use cases. This is where we use the custom validator. It is very easy to create a custom validator in Angular.

Ad by Valueimpression

How to Build Custom Validator

Building a custom Validator is as easy as creating a Validator function. It is a function, which must implement [ValidatorFn](https://angular.io/api/forms/ValidatorFn) Interface.

ValidatorFn

The [ValidatorFn](https://angular.io/api/forms/ValidatorFn) is an Interface, which defines the signature of the Validator function.

|  |  |
| --- | --- |
| 1  2  3  4  5 | interface ValidatorFn {    (control: AbstractControl): ValidationErrors | null  } |

The function takes the [AbstractControl](https://angular.io/api/forms/AbstractControl). This is the base class for [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/), [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/), and [FormArray](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/). The validator function must return a list of errors i.e [ValidationErrors](https://angular.io/api/forms/ValidationErrors) or null if the validation has passed

Custom Validator Example

[Create a new angular application](https://www.tektutorialshub.com/angular/angular-create-first-application/). Add the following code in app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | <h1>Custom Validator in Angular</h1>    <h2>Reactive Form</h2>    <form [formGroup]="myForm" (ngSubmit)="onSubmit()" novalidate>      <div>      <label for="numVal">Number :</label>      <input type="text" id="numVal" name="numVal" formControlName="numVal">    </div>      <p>Is Form Valid : {{myForm.valid}} </p>      <p>      <button type="submit" [disabled]="!myForm.valid">Submit</button>    </p>    </form> |

[***Source Code***](https://stackblitz.com/edit/angular-custom-validator-in-angular-reactive-form-1)

Our example app has numVal form field. We want it to be greater than 10.

Angular does not have any built-in validator for that. Hence let us build a custom Validator gte

Create a new file gte.validator.ts under the app folder.

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import { AbstractControl, ValidationErrors } from '@angular/forms'    export function gte(control: AbstractControl): ValidationErrors | null {        const v=+control.value;        if (isNaN(v)) {        return { 'gte': true, 'requiredValue': 10 }      }        if (v <= 10) {        return { 'gte': true, 'requiredValue': 10 }      }        return null    } |

[***Source Code***](https://stackblitz.com/edit/angular-custom-validator-in-angular-reactive-form-1)

First, import the AbstractControl and ValidationErrors from the @angular/forms

|  |  |
| --- | --- |
| 1  2  3 | import { AbstractControl, ValidationErrors } from '@angular/forms' |

The validator function must adhere to the [ValidatorFn](https://angular.io/api/forms/ValidatorFn) Interface. It should receive the AbstractControl as its parameter. It can be [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/), FormGroup or FormArray.

The function must validate the control value and return ValidationErrors if any errors are found otherwise null.

|  |  |
| --- | --- |
| 1  2  3 | export function gte(control: AbstractControl): ValidationErrors | null { |

The ValidationErrors is a key-value pair object of type [key: string]: any and it defines the broken rule. The key is the string and should contain the name of the broken rule. The value can be anything, but usually set to true.

The validation logic is very simple. Check if the value of the control is a number using the [isNaN](https://www.tektutorialshub.com/typescript/nan-in-typescript/) method. Also, check if the value is less than or equal to 10. If both the rules are valid and then return null

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | const v=+control.value;        if (isNaN(v)) {        return { 'gte': true, 'requiredValue': 10 }      }        if (v <= 10) {        return { 'gte': true, 'requiredValue': 10 }      }        return null |

If the validation fails then return the ValidationErrors. You can use anything for the key, but it is advisable to use the name of the validator i.e gte as the key. Also, assign true as value. You can as well assign a string value.

|  |  |
| --- | --- |
| 1  2  3 | return { 'gte': true, 'requiredValue': 10 } |

You can return more than one key-value pair as shown in the above example. The second key requiredValue returns the value 10. We use this in the template to show the error message.

Using the Custom Validator

To use this validator first, import it in the component class.

|  |  |
| --- | --- |
| 1  2  3 | import { gte } from './gte.validator'; |

Add the validator to the Validator collection of the [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | myForm = new FormGroup({      numVal: new FormControl('', [gte]),    }) |

The complete app.component.ts is shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | import { Component } from '@angular/core';  import { FormGroup, FormControl, AbstractControl, ValidationErrors } from '@angular/forms'  import { gte } from './gte.validator';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {      constructor() {    }      myForm = new FormGroup({      numVal: new FormControl('', [gte]),    })      get numVal() {      return this.myForm.get('numVal');    }      onSubmit() {      console.log(this.myForm.value);    }  } |

[***Source Code***](https://stackblitz.com/edit/angular-custom-validator-in-angular-reactive-form-1)

Accessing the Errors from Custom Validator

We need to provide a meaningful error message to the user.

Validators return ValidationErrors. They are added to the control’s errors collection of the control. The valid property of the control is set to false.

Hence we check if the valid property. We also check the dirty and touched property. Because we do not want to display the error message when the form is displayed for the first time.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <div>      <label for="numVal">Number :</label>      <input type="text" id="numVal" name="numVal" formControlName="numVal">      <div \*ngIf="!numVal.valid && (numVal.dirty ||numVal.touched)">        <div \*ngIf="numVal.errors.gte">          The number should be greater than {{numVal.errors.requiredValue}}        </div>      </div>      </div> |

We check if the gte is true and display the error message. Note that gte is the name of the key we used while creating the validator.

We also make use of the requiredValue to show a meaningful message to the user.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div \*ngIf="numVal.errors.gte">     The number should be greater than {{numVal.errors.requiredValue}}  </div> |

Custom Validator with Parameter

Here is the code of greater than validator (gte) from the [Custom Validators in Angular Reactive Form](https://www.tektutorialshub.com/angular/custom-validator-in-angular-reactive-form/) tutorial. The validator checks if the given value is greater than 10 and if not return ValidationErrors.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | export function gte(control: AbstractControl): ValidationErrors | null {        const v:number=+control.value;        if (isNaN(v)) {        return { 'gte': true, 'requiredValue': 10 }      }        if (v <= 10) {        return { 'gte': true, 'requiredValue': 10 }      }        return null    } |

The problem with the above validator is that the value 10 is hardcoded. Hence, we will be not able to reuse it. If we want to resue it, we need to pass the number as the parameter.

Let us add the parameter val:number to the validator as shown below.

|  |  |
| --- | --- |
| 1  2  3 | export function gte(control: AbstractControl,val:number): ValidationErrors | null { |

The compiler immediately throws an error as shown below.

|  |  |
| --- | --- |
| 1  2  3 | error TS2345: Argument of type '((control: AbstractControl, val: number) => ValidationErrors)[]' is not assignable to parameter of type 'ValidatorFn | ValidatorFn[] | AbstractControlOptions'. |

That is because, the Validator must implement [ValidatorFn](https://angular.io/api/forms/ValidatorFn) Interface. It can have only one parameter i.e control: AbstractControl

|  |  |
| --- | --- |
| 1  2  3  4  5 | interface ValidatorFn {    (control: AbstractControl): ValidationErrors | null  } |

Passing Parameters to a Custom Validator

To pass a parameter, we need to create a factory function or a function that returns a function. The example code is as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | export function gte(val: number): ValidatorFn {      return (control: AbstractControl): ValidationErrors | null => {        let v: number = +control.value;        if (isNaN(v)) {        return { 'gte': true, 'requiredValue': val }      }        if (v <= +val) {        return { 'gte': true, 'requiredValue': val }      }        return null;      }    } |

First, we create a factory function. It receives the val as the argument. It must return the function of the type ValidatorFn

|  |  |
| --- | --- |
| 1  2  3 | export function gte(val: number): ValidatorFn { |

The get must return a function ValidatorFn

|  |  |
| --- | --- |
| 1  2  3  4  5 | return (control: AbstractControl): ValidationErrors | null => {      //Validaton code here  } |

Using the Validator

Now, add the validator to the Validator collection of the FormControl as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5 | myForm = new FormGroup({      numVal: new FormControl('', [gte(10)]),    }) |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Accessing the Errors in Template

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | <h1>Custom Validator with Parameter in Angular</h1>    <h2>Reactive Form</h2>    <form [formGroup]="myForm" (ngSubmit)="onSubmit()" novalidate>      <div>      <label for="numVal">Number :</label>      <input type="text" id="numVal" name="numVal" formControlName="numVal">      <div \*ngIf="!numVal.valid && (numVal.dirty ||numVal.touched)">        <div \*ngIf="numVal.errors.gte">          The number should be greater than {{numVal.errors.requiredValue}}        </div>      </div>      </div>        <p>Is Form Valid : {{myForm.valid}} </p>      <p>      <button type="submit" [disabled]="!myForm.valid">Submit</button>    </p>      </form> |

Summary

We learned how to pass a parameter to a custom validator. First, we create a factory function, which accepts the parameter. The factory function returns the validator function. Using this technique we can pass as many as parameters to a [custom validator in Angular.](https://www.tektutorialshub.com/angular/custom-validator-in-angular-reactive-form/)

Custom Validator

Here is our greater than [custom validator](https://www.tektutorialshub.com/angular/custom-validator-in-angular-reactive-form/) gte from the previous tutorial. The gte is a function that returns [ValidatorFn](https://angular.io/api/forms/ValidatorFn) function.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import { AbstractControl, ValidationErrors, ValidatorFn } from '@angular/forms'    export function gte(val: number): ValidatorFn {      return (control: AbstractControl): ValidationErrors | null => {        let v: number = +control.value;        if (isNaN(v)) {        return { 'gte': true, 'requiredValue': val }      }        if (v <= +val) {        return { 'gte': true, 'requiredValue': val }      }        return null;    }  } |

We pass the parameter to the validator in the component class as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5 | myForm = new FormGroup({    numVal: new FormControl('', [gte(10)]),  }) |

Validator Service

Let us now see how we can inject service into the above validator. First, move the logic from the gte validator to a separate service.

Create a new service gte.service.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import { Injectable } from '@angular/core';    @Injectable({    providedIn: 'root',  })  export class gteService {      gte(num:any, requiredValue:Number) : Boolean {        if (isNaN(num)) {        return false;      }        if (num <= +requiredValue) {        return false;      }        return true;    }  } |

The gteService is very simple.

The gte method takes val and requiredValue as the parameter. It checks if the val is a number and is greater than requiredValue. If yes returns true else returns false.

Injecting Service

There are two ways you can inject service into the validator. One is to create a wrapper service. The other option is to inject service directly into the validator.

Wrapper Service

Open the gte.validator.ts. Create the gteValidatorService class. In the constructor inject the gteService. Copy the validator functiongte into the class and use the gteService as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | import { AbstractControl, ValidationErrors, ValidatorFn, Validator, NG\_VALIDATORS, FormControl } from '@angular/forms'  import { gteService } from './gte.service';  import { Directive, OnInit, forwardRef, Input, Injectable } from '@angular/core';    @Injectable({    providedIn: 'root',  })  export class gteValidatorService {      constructor(private gteService: gteService) {    }      gte(val: number): ValidatorFn {        return (control: AbstractControl): ValidationErrors | null => {          let v: number = +control.value;        if (!this.gteService.gte(v,val)) {          return { 'gte': true, 'requiredValue': val }        }          return null;      }    }  } |

Inject the gteValidatorService in the component class and use the validator as shown below.

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | import { Component } from '@angular/core';  import { FormGroup, FormControl, AbstractControl, ValidationErrors, ValidatorFn } from '@angular/forms'  import { gteValidatorService } from './gte.validator';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {      constructor(private gteValidator:gteValidatorService) {    }        myForm = new FormGroup({      numVal: new FormControl('', [this.gteValidator.gte(10)]),    })      get numVal() {      return this.myForm.get('numVal');    }        onSubmit() {      console.log(this.myForm.value);    }  } |

Inject Service directly into the Validator

Another option is to directly inject the service into the Validator as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import { AbstractControl, ValidationErrors, ValidatorFn, Validator, NG\_VALIDATORS, FormControl } from '@angular/forms'  import { gteService } from './gte.service';  import { Directive, OnInit, forwardRef, Input, Injector } from '@angular/core';    export function gte(val: number): ValidatorFn {      return (control: AbstractControl): ValidationErrors | null => {        let injector = Injector.create([ { provide: gteService, useClass:gteService,deps: []}])      let service = injector.get(gteService);        let v: number = +control.value;      if (!service.gte(v, val)) {        return { 'gte': true, 'requiredValue': val }      }        return null;      }  } |

We use the Injectorto inject the instance of the service.

|  |  |
| --- | --- |
| 1  2  3  4 | let injector = Injector.create([ { provide: gteService, useClass:gteService,deps: []}])      let service = injector.get(gteService); |

Summary

In this article, we learned how to inject the service into the validator function. One option is to create a Validator service. Copy the validator function into the Validator service. Another option is to inject the service directly into the validator function using the inject method.

Template-driven Form Validation

Validations in [Template-driven forms](https://www.tektutorialshub.com/angular/angular-template-driven-forms/) are provided by the Validation directives. The [Angular Forms Module](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/) comes with several built-in validators. You can also create your own custom Validator.

Template

Consider the following template-driven form. It has firstname, lastname, email, gender & istoc form fields.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)">      <p>      <label for="firstname">First Name </label>      <input type="text" id="firstname" name="firstname" [(ngModel)]="contact.firstname">    </p>      <p>      <label for="lastname">Last Name </label>      <input type="text" id="lastname" name="lastname" [(ngModel)]="contact.lastname">    </p>      <p>      <label for="email">email </label>      <input type="text" id="email" name="email" [(ngModel)]="contact.email">    </p>      <p>      <label for="gender">Geneder </label>      <input type="radio" value="male" id="gender" name="gender" [(ngModel)]="contact.gender"> Male      <input type="radio" value="female" id="gender" name="gender" [(ngModel)]="contact.gender"> Female    </p>      <p>      <label for="isToc">Accept TOC</label>      <input type="checkbox" id="isToc" name="isToc" [(ngModel)]="contact.isToc">    </p>      <p>      <button type="submit">Submit</button>    </p>    </form> |

**Component Class**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | import { Component, ViewChild, ElementRef, OnInit } from '@angular/core';  import { NgForm } from '@angular/forms';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit  {    title = 'Template driven forms';      @ViewChild('contactForm',null) contactForm: NgForm;      contact:contact;      ngOnInit() {        this.contact = {        firstname:"",        lastname:"",        gender:"male",        isToc:true,        email:"",      };      }      onSubmit() {      console.log(this.contactForm.value);    }    }    export class contact {    firstname:string;    lastname:string;    gender:string;    isToc:boolean;    email:string;  } |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Disabling the Browser validation

First, we need to disable browser validator interfering with the Angular validator. To do that we need to add novalidate attribute on <form> element as shown below

|  |  |
| --- | --- |
| 1  2  3 | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)" novalidate> |

Built-in Validators

The Built-in validators use the HTML5 validation attributes like required, minlength, maxlength & pattern. Angular interprets these validation attributes and add the validator functions to [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) instance.

Adding in Built-in Validators

Required Validation

The required validator returns true only if the form control has non-empty value entered. Let us add this validator to all fields

|  |  |
| --- | --- |
| 1  2  3 | <input type="text" id="firstname" name="firstname" required [(ngModel)]="contact.firstname"> |

Minlength Validation

This Validator requires the control value must not have less number of characters than the value specified in the validator.

For Example, minlength validator ensures that the firstname value has at least 10 characters.

|  |  |
| --- | --- |
| 1  2  3 | <input type="text" id="firstname" name="firstname" required minlength="10" [(ngModel)]="contact.firstname"> |

Maxlength Validation

This Validator requires that the number of characters must not exceed the value of the attribute.

|  |  |
| --- | --- |
| 1  2  3 | <input type="text" id="lastname" name="lastname" required maxlength="15" [(ngModel)]="contact.lastname"> |

Pattern Validation

This Validator requires that the control value must match the regex pattern provided in the attribute. For example, the pattern ^[a-zA-Z]+$ ensures that the only letters are allowed (even spaces are not allowed). Let us apply this pattern to the lastName

|  |  |
| --- | --- |
| 1  2  3  4 | <input type="text" id="lastname" name="lastname" required maxlength="15"      pattern="^[a-zA-Z]+$" [(ngModel)]="contact.lastname"> |

Email Validation

This Validator requires that the control value must be a valid email address. We apply this to the email field

|  |  |
| --- | --- |
| 1  2  3 | <input type="text" id="email" name="email" required email [(ngModel)]="contact.email"> |

Disable Submit button

Now, we have successfully added the validators. You will notice that the click submit button still submits the form.

We need to disable the submit button if our form is not valid.

Angular forms module keep track of the state of our form and each of its form elements. These states are exposed to the user through FormGroup, FormArray & FormControl objects.

We get the reference to the top-level FormGroup instance by creating a template variable and bind it to ngForm. We have already done it when we had added the #contactForm="ngForm" in our form tag.

The [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) has a valid property, which is set to true if all of its child controls are valid. We use it to set the disabled attribute of the submit button.

|  |  |
| --- | --- |
| 1  2  3 | <button type="submit" [disabled]="!contactForm.valid">Submit</button> |

So long as contactForm.valid remains false, the submit button remains disabled.

Displaying the Validation/Error messages

We need to provide a short and meaningful error message to the user.

Angular creates a FormControl for each and every field, which has ngModel directive applied. The FormControl exposes the state of form element like valid, dirty, touched, etc.

There are two ways in which you can get the reference to the FormControl.

One way is to use the contactForm variable. We can use the contactForm.controls.firstname.valid to find out if the firstname is valid.

The other way is to create a new local variable for each FormControl For Example, the following firstname="ngModel" creates the firstname variable with the FormControl instance.

|  |  |
| --- | --- |
| 1  2  3  4 | <input type="text" id="firstname" name="firstname" required minlength="10"              #firstname="ngModel" [(ngModel)]="contact.firstname"> |

Now, we have a reference to the firstname FormControl instance, we can check its status. We use the valid property to check if the firstname has any errors.

valid: returns either invalid status or null which means a valid status

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div \*ngIf="!firstname?.valid && (firstname?.dirty || firstname?.touched)">     Invalid First Name  </div> |

Why check dirty and touched?

We do not want the application to display the error when the form is displayed for the first time. We want to display errors only after the user has attempted to change the value. The dirty & touched properties help us do that.

dirty: A control is dirty if the user has changed the value in the UI.  
touched: A control is touched if the user has triggered a blur event on it.

Error message

The error message ” “Invalid First Name” ” is not helpful. The firstname has two validators. required and minlength

Any errors generated by the failing validation is updated in the errors object. The errors object returns the error object or null if there are no errors.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <div \*ngIf="!firstname?.valid && (firstname?.dirty || firstname?.touched)">    Invalid First Name    <div \*ngIf="firstname.errors.required">       First Name is required    </div>    <div \*ngIf="firstname.errors.minlength">      First Name Minimum Length is {{firstname.errors.minlength?.requiredLength}}    </div>  </div> |

Note that the minlength validators return the {{firstname.errors.minlength?.requiredLength}}, which we use the display the error message.

Final Template

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81 | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)" novalidate>      <p>      <label for="firstname">First Name </label>      <input type="text" id="firstname" name="firstname" required minlength="10" #firstname="ngModel"        [(ngModel)]="contact.firstname">    </p>    <div \*ngIf="!firstname?.valid && (firstname?.dirty || firstname?.touched)" class="error">      <div \*ngIf="firstname.errors.required">        First Name is required      </div>      <div \*ngIf="firstname.errors.minlength">        First Name Minimum Length is {{firstname.errors.minlength?.requiredLength}}      </div>    </div>      <p>      <label for="lastname">Last Name </label>      <input type="text" id="lastname" name="lastname" required maxlength="15" #lastname="ngModel"              pattern="^[a-zA-Z]+$"  [(ngModel)]="contact.lastname">    </p>    <div \*ngIf="!lastname?.valid && (lastname?.dirty || lastname?.touched)" class="error">      <div \*ngIf="lastname.errors.required">        Last Name is required      </div>      <div \*ngIf="lastname.errors.maxlength">        Last Name Minimum Length is {{lastname.errors.maxlength?.requiredLength}}      </div>      <div \*ngIf="lastname.errors.pattern">        Only characters are allowed      </div>    </div>          <p>      <label for="email">email </label>      <input type="text" id="email" name="email" required email #email="ngModel" [(ngModel)]="contact.email">    </p>    <div \*ngIf="!email?.valid && (email?.dirty || email?.touched)" class="error">      <div \*ngIf="email.errors.required">        Email is required      </div>      <div \*ngIf="email.errors.email">        Invalid Email Address      </div>    </div>      <p>      <label for="gender">Geneder </label>      <input type="radio" value="male" id="gender" name="gender" #gender="ngModel" required [(ngModel)]="contact.gender">      Male      <input type="radio" value="female" id="gender" name="gender" #gender="ngModel" required        [(ngModel)]="contact.gender"> Female    </p>    <div \*ngIf="!gender?.valid && (gender?.dirty || gender?.touched)" class="error">      <div \*ngIf="gender.errors.required">        Gender is required      </div>    </div>        <p>      <label for="isToc">Accept TOC</label>      <input type="checkbox" id="isToc" name="isToc" required #isToc="ngModel" [(ngModel)]="contact.isToc">    </p>    <div \*ngIf="!isToc?.valid && (isToc?.dirty || isToc?.touched)" class="error">      <div \*ngIf="isToc.errors.required">        Please accept the TOC      </div>    </div>      <p>      <button type="submit" [disabled]="!contactForm.valid">Submit</button>    </p>      <p>{{contactForm.valid}} </p>    </form> |

Summary

Angular template-driven form validation uses the directives known as validators. The validators handle form validations and display validation messages. The Angular comes up with several built-in validators for this purpose. They are minlength, maxlength, email, pattern, required, etc.

Custom Validator in Template Driven Forms

[Create a new Angular Project](https://www.tektutorialshub.com/angular/angular-create-first-application/). Copy the following code toapp.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',  })  export class AppComponent {      constructor() {    }    } |

Copy the following code app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | <h1>Custom Validator in Template driven forms</h1>    <h2>Template Form</h2>    <form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)" novalidate>      <label for="numVal">Number :</label>    <input type="text" name="numVal" ngModel #numVal="ngModel">      <p>Is Form Valid : {{myForm.valid}} </p>      <p>Form  : {{ myForm.value | json}} </p>    <p>      <button type="submit" [disabled]="!myForm.valid">Submit</button>    </p>    </form> |

It has only one input field numVal. Let us create a validator to ensure that the value of the numVal is greater than 10.

Built-in Validators

The [Angular Forms](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/) Module already has a few built-in validators. They are listed below. But we do not have a greater than validator.

[Required validator](https://www.tektutorialshub.com/angular/template-driven-form-validation-in-angular/#required-validation)

[Min length Validator](https://www.tektutorialshub.com/angular/template-driven-form-validation-in-angular/#minlength-validation)

[Max length Validator](https://www.tektutorialshub.com/angular/template-driven-form-validation-in-angular/#maxlength-validation)

[Pattern Validator](https://www.tektutorialshub.com/angular/template-driven-form-validation-in-angular/#pattern-validation)

[Email Validator](https://www.tektutorialshub.com/angular/template-driven-form-validation-in-angular/#email-validation)

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

How to Build Custom Validator in template-driven form

Building a Validator in [template-driven forms](https://www.tektutorialshub.com/angular/angular-template-driven-forms/) is similar to building an Angular directive. The directive must implement the Validator interface.

Validator Interface

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | interface Validator {    validate(control: AbstractControl): ValidationErrors | null    registerOnValidatorChange(fn: () => void)?: void  } |

The directive must implement the validate function. Notice that the validate function has the same signature as the [ValidatorFn](https://angular.io/api/forms/ValidatorFn) Interface. Whenever the Validator directive is invoked angular looks for the validate method and invokes it.

Validate Function

A Validator is just a function, which must implement [ValidatorFn](https://angular.io/api/forms/ValidatorFn) Interface.

|  |  |
| --- | --- |
| 1  2  3  4  5 | interface ValidatorFn {    (control: AbstractControl): ValidationErrors | null  } |

The function takes the [AbstractControl](https://angular.io/api/forms/AbstractControl). This is the base class for [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/), [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/), and [FormArray](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/). The validator function must return a list of errors i.e [ValidationErrors](https://angular.io/api/forms/ValidationErrors) or null if the validation has passed

Custom Validator Example

Create the gte.validator.ts and copy the following code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | import { Validator, NG\_VALIDATORS, FormControl } from '@angular/forms'  import { Directive, OnInit, forwardRef } from '@angular/core';      @Directive({    selector: '[gteValidator]',    providers: [      { provide: NG\_VALIDATORS, useExisting: gteValidatorDirective, multi: true }    ]  })  export class gteValidatorDirective implements Validator, OnInit {      ngOnInit() {    }      validate(c: FormControl) {        let v: number = +c.value;        if (isNaN(v)) {        return { 'gte': true, 'requiredValue': 10 }      }        if (v <= +10) {        return { 'gte': true, 'requiredValue': 10 }      }        return null;    }  } |

We decorate the [directive](https://www.tektutorialshub.com/angular/angular-directives/) using @Directive decorator.

We use the directive as an attribute in the HTML template. The attribute needs a name or selector. We assign the name as gteValidator in the selector metadata section of the directive decorator.

|  |  |
| --- | --- |
| 1  2  3 | selector: '[gteValidator]', |

The Angular knows nothing about the Validation capabilities of our directive. Hence we need to register it in [Angular Providers](https://www.tektutorialshub.com/angular/angular-providers/) metadata using the special injection token [NG\_VALIDATORS](https://angular.io/api/forms/NG_VALIDATORS). We also set multi:true because there can be more validation directives.

|  |  |
| --- | --- |
| 1  2  3 | { provide: NG\_VALIDATORS, useExisting: gteValidatorDirective, multi: true } |

The directive class must implement the validate method. The validate method must honor the [ValidatorFn](https://angular.io/api/forms/ValidatorFn) Interface.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | validate(c: FormControl) {        let v: number = +c.value;        if (isNaN(v)) {        return { 'gte': true, 'requiredValue': 10 }      }        if (v <= +10) {        return { 'gte': true, 'requiredValue': 10 }      }        return null;    } |

It is a simple function, which checks if the value is a number and is less than 10. It returns null if it passes all checks.

If Validation fails it returns the ValidationErrors. It is a key-value pair object of type [key: string]: any and it defines the broken rule. The key is the string and should contain the name of the broken rule. The value can be anything, but usually set to true.

We return the following key-value pair when the validation fails

|  |  |
| --- | --- |
| 1  2  3 | return { 'gte': true, 'requiredValue': 10 } |

The 'gte': true: indicates that the validation has failed. 'requiredValue': 10 is used by the template to display that the expected value is greater than 10.

Using the Custom Validator

Since this is a template-driven form., we do not have to do anything in the component class. In the HTML template just add the attribute gteValidator as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <label for="numVal">Number :</label>      <input type="text" name="numVal" ngModel #numVal="ngModel" gteValidator>      <div \*ngIf="!numVal.valid && (numVal.dirty ||numVal.touched)">      <div \*ngIf="numVal.errors.gte">        The number should be greater than {{numVal.errors.requiredValue}}      </div>    </div> |

Validators return ValidationErrors. They are added to the control’s errors collection of the control. The valid property of the control is set to false.

Hence we check if the valid property. We also check the dirty and touched property. Because we do not want to display the error message when the form is displayed for the first time.

We check if the gte is true and display the error message. Note that gte is the name of the key we used while creating the validator.

We also make use of requiredValue to show a meaningful message to the user.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div \*ngIf="numVal.errors.gte">        The number should be greater than {{numVal.errors.requiredValue}}      </div> |

Passing Parameter to Validator

We have hardcoded the value of 10 in the above example. This will make our validator difficult to reuse. If we want to resue it, we need to pass the number to be checked as the parameter.

Since they are directives, we can use [Input decorator](https://angular.io/api/core/Input) to pass the parameter to the Validator.

Open the template add the special attribute gteNum="20"

|  |  |
| --- | --- |
| 1  2  3 | <input type="text" name="numVal" ngModel #numVal="ngModel" gteValidator gteNum="20" > |

You can read the gteNum from the template using the Input decorator as shown below

|  |  |
| --- | --- |
| 1  2  3 | @Input("gteNum") gteNum:number |

Now, you can remove the hardcoded value 10 and use the gteNum instead.

The complete validator code is as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | import { Validator, NG\_VALIDATORS, FormControl } from '@angular/forms'  import { Directive,  Input } from '@angular/core';    @Directive({    selector: '[gteValidator]',    providers: [      { provide: NG\_VALIDATORS, useExisting: gteValidatorDirective, multi: true }    ]  })  export class gteValidatorDirective implements Validator {      @Input("gteNum") gteNum:number      validate(c: FormControl) {        let v: number = +c.value;        if (isNaN(v)) {        return { 'gte': true, 'requiredValue': this.gteNum }      }        if (v <= +this.gteNum) {        return { 'gte': true, 'requiredValue': this.gteNum }      }        return null;    }    } |

Injecting Service into Validator

The validator may depend on some external service to validate the value. For Example, it may need to fetch data from the back end server.

Let us move the validation logic in the above validator to a separate service. Create a service gte.service.ts and copy the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import { Injectable } from '@angular/core';    @Injectable({    providedIn: 'root',  })  export class gteService {      gte(num:any, requiredValue:Number) : Boolean {        if (isNaN(num)) {        return false;      }        if (num <= +requiredValue) {        return false;      }        return true;    }  } |

In the validation directive, create a constructor method and inject the service. The complete code is as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | import { Validator, NG\_VALIDATORS, FormControl } from '@angular/forms'  import { Directive,  Input } from '@angular/core';  import { gteService } from 'projects/injectService1/src/app/gte.service';    @Directive({    selector: '[gteValidator]',    providers: [      { provide: NG\_VALIDATORS, useExisting: gteValidatorDirective, multi: true }    ]  })  export class gteValidatorDirective implements Validator {      @Input("gteNum") gteNum:number      constructor(private gteService:gteService) {    }      validate(c: FormControl) {        let v: number = +c.value;        if (this.gteService.gte(v,this.gteNum)) {        return { 'gte': true, 'requiredValue': this.gteNum }      }        return null;    }  } |

Summary

We learned how to create a custom validator in template-driven forms. We also talked about how to pass parameter and inject service into our directive.

How to Create Async Validator

Creating a Async Validator is simple as creating a function, which must obey the following rules

The function must implement the [AsyncValidatorFn](https://angular.io/api/forms/AsyncValidator) Interface, which defines the signature of the validator function.

The function must return either an [observable](https://www.tektutorialshub.com/angular/angular-observable-tutorial-using-rxjs/)or a promise

Return null for valid, or an ValidationErrors if the input is invalid

AsyncValidatorFn

The [AsyncValidatorFn](https://angular.io/api/forms/AsyncValidator) is an Interface, which defines the signature of the validator function.

|  |  |
| --- | --- |
| 1  2  3  4  5 | interface AsyncValidatorFn {    (control: AbstractControl): Promise<ValidationErrors | null> | Observable<ValidationErrors | null>  } |

The function takes the [AbstractControl](https://angular.io/api/forms/AbstractControl). This is the base class for [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/), [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/), and [FormArray](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/). The validator function must return a list of errors i.e [ValidationErrors](https://angular.io/api/forms/ValidationErrors) or null if the validation has passed. The only difference it has with the Sync Validator is the return type. It must return either a promise or an [observable](https://www.tektutorialshub.com/angular/angular-observable-tutorial-using-rxjs/).

Async Validator Example

We build gte validator in [how to create a custom validator in Angular](https://www.tektutorialshub.com/angular/custom-validator-in-angular-reactive-form/) tutorial. In this Async Validator Example, let us convert that validator to Async Validator.

[Create a new Angular Application](https://www.tektutorialshub.com/angular/angular-create-first-application/). Add the gte.validator.ts and copy the following code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | import { AbstractControl, ValidationErrors } from '@angular/forms'    import { Observable, of } from 'rxjs';    export function gte(control: AbstractControl):           Observable<ValidationErrors> | null {        const v:number=+control.value;        console.log(v)        if (isNaN(v)) {        return of({ 'gte': true, 'requiredValue': 10 })      }        if (v <= 10) {        return of({ 'gte': true, 'requiredValue': 10 })      }        return of(null)    } |

First, import the AbstractControl and ValidationErrors from the @angular/forms. Since we need to return an observable, we also need to import Observable from the rxjs library.

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

The validator function must follow the [AsyncValidatorFn](https://angular.io/api/forms/AsyncValidator) Interface. It should receive the AbstractControl as its parameter. It can be [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/), [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) or [FormArray](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/).

The function must validate the control value and return ValidationErrors if any errors are found otherwise null. It must return them as [observable](https://www.tektutorialshub.com/angular/angular-observable-tutorial-using-rxjs/).

The ValidationErrors is a key-value pair object of type [key: string]: any and it defines the broken rule. The key is the string and should contain the name of the broken rule. The value can be anything, but usually set to true.

The validation logic is very simple. Check if the value of the control is a number using the [isNaN](https://www.tektutorialshub.com/typescript/nan-in-typescript/) method. Also, check if the value is less than or equal to 10. If both the rules are valid and then return null

If the validation fails then return the ValidationErrors. You can use anything for the key, but it is advisable to use the name of the validator i.e gte as the key. Also, assign true as value. You can as well assign a string value.

You can return more than one key-value pair as shown in the above example. The second key requiredValue returns the value 10. We use this in the template to show the error message.

We use the [of](https://www.tektutorialshub.com/angular/rxjs-observable-using-create-of-from-in-angular/#of-operator) operator convert the result into an observable and return it

Using the Async Validator

To use this validator first, import it in the component class.

|  |  |
| --- | --- |
| 1  2  3 | import { gte } from './gte.validator'; |

Add the validator to the Async Validator collection of the [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) as shown below. The async validator is the third argument to the FormControl.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | myForm = new FormGroup({      numVal: new FormControl('', [gte]),    })      // Without FormGroup    // this.myForm = new FormGroup({    //   numVal: new FormControl('', null, [gte]),    // }) |

That’s it. The complete app.component.ts code as show below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | import { Component } from '@angular/core';  import { FormGroup, FormControl } from '@angular/forms'  import { gte } from './gte.validator';      @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {      constructor() {    }        myForm = new FormGroup({      numVal: new FormControl('',null, [gte]),    })      get numVal() {      return this.myForm.get('numVal');    }        onSubmit() {      console.log(this.myForm.value);    } |

The complete app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | <h1>Async Validator in Angular</h1>    <h2>Reactive Form</h2>    <form autocomplete="off" [formGroup]="myForm" (ngSubmit)="onSubmit()" novalidate>      <div>      <label for="numVal">Number :</label>      <input type="text" id="numVal" name="numVal" formControlName="numVal">      <div \*ngIf="!numVal.valid && (numVal.dirty ||numVal.touched)">        <div \*ngIf="numVal.errors.gte">          The number should be greater than {{numVal.errors.requiredValue}}        </div>      </div>      </div>        <p>Is Form Valid : {{myForm.valid}} </p>      <p>      <button type="submit" [disabled]="!myForm.valid">Submit</button>    </p>      </form> |

app.module.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | 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 { } |

The use case for Async Validators

We use the async validator when we need to send an HTTP call to the server to check if the data is valid.

The following code shows how you can send a HTTP Request to verify the data.

If the data is Valid we will return nothing, else we return the ValidationErrors  i.e. ({ 'InValid': true }).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import { AbstractControl, ValidationErrors } from '@angular/forms'  import { Observable, pipe } from 'rxjs';  import { map, debounceTime } from 'rxjs/operators';    export function validate(control: AbstractControl): Observable<ValidationErrors> | null {      const value: string = control.value;      return this.http.get(this.baseURL + 'checkIfValid/?value=' + value)      .pipe(        debounceTime(500),        map( (data:any) =>  {            if (!data.isValid) return ({ 'InValid': true })        })      )    } |

You can also [pass parameter to the validator](https://www.tektutorialshub.com/angular/custom-validator-with-parameters-in-angular/) & [Inject Service into the validator](https://www.tektutorialshub.com/angular/inject-service-into-validator-in-angular/).

You can also use the [combineLatest](https://www.learnrxjs.io/learn-rxjs/operators/combination/combinelatest) to merge data from more than one observable and validate the input.

Validation Recap

We assign a validator’s to a form filed, using the second argument of the [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) as shown below. You can also attach an [Async Validator](https://www.tektutorialshub.com/angular/angular-async-validator-example/) as the third argument.

|  |  |
| --- | --- |
| 1  2  3  4 | this.contactForm = new FormGroup({      userName: new FormControl('',[Validators.required,customValidator]), |

The above syntax using the [FormBuilder](https://www.tektutorialshub.com/angular/angular-formbuilder-in-reactive-forms/).

|  |  |
| --- | --- |
| 1  2  3  4 | this.contactForm = this.builder.group({        userName: ["", [Validators.required,customValidator]], |

The Validator will run only when we change the value of userName and Validates only the userName field.

Cross Field Validation

When we validate the multiple fields, we need to ensure that our validation logic runs for each of those fields.

Hence we attach the validator to the [Formgroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) instead of [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/). The Validator runs whenever we modify any of the fields in the FormGroup.

Example

Let us create a matchPassword [custom validator](https://www.tektutorialshub.com/angular/custom-validator-in-angular-reactive-form/) to compare the password & confirm Password fields.

Since we attach it to a [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/), it gets the instance of FormGroup as its parameter. We can use the get method to get the values of both password & confirm FormControls. If they do not match then return the ValidationErrors. Return null if it values passes the Validation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | matchPassword(control: AbstractControl): ValidationErrors | null {        const password = control.get("password").value;      const confirm = control.get("confirm").value;          if (password != confirm) { return { 'noMatch': true } }        return null      } |

We attach the matchPassword Validator to FormGroup using its second argument as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | this.mainForm = this.builder.group({        userName: ["", [Validators.required]],        password: ["", [Validators.required, Validators.minLength(5)]],        confirm: ["", [Validators.required]]      }, { validator: this.matchPassword }); |

The FormGroup also allows us the add more than one validator using the Validators.compose method.

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | this.mainForm = this.builder.group({        userName: ["", [Validators.required]],        password: ["", [Validators.required, Validators.minLength(5)]],        confirm: ["", [Validators.required]]      }, {        validator: Validators.compose(          [            this.matchPassword,            Validators.required          ]        )      }); |

Passing Parameter

You can also pass the parameter to the Multiple Field Validator.

In the following example, we pass the name of the

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | this.mainForm = this.builder.group({        userName: ["", [Validators.required]],        password: ["", [Validators.required, Validators.minLength(5)]],        confirm: ["", [Validators.required]]      }, { validator: this.matchPassword2('password', 'confirm') }); |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | matchPassword2(firstControl, secondControl): ValidatorFn {        return (control: AbstractControl): ValidationErrors | null => {          const password = control.get(firstControl).value;        const confirm = control.get(secondControl).value;          if (password != confirm) { return { 'noMatch': true } }          return null        }    } |

Adding the Validators Using the SetValidators

Syntax

The setValidators programmatically adds the sync validators. This method will remove all the previously added sync or async validators.

setValidators(newValidator: ValidatorFn | ValidatorFn[]): void

Examples:

|  |  |
| --- | --- |
| 1  2  3 | this.myform.controls["mobile"].setValidators(Validators.required); |

|  |  |
| --- | --- |
| 1  2  3 | this.myform.controls["mobile"].setValidators([Validators.required,Validators.minLength(10)]); |

setAsyncValidators

The setAsyncValidators programmatically add the Async validators.

|  |  |
| --- | --- |
| 1  2  3 | setAsyncValidators(newValidator: AsyncValidatorFn | AsyncValidatorFn[]): void |

setValidators overwrites all existing Validators. Hence it is very important to include all the validators that we want in the setValidators method

Removing Validators Using clearValidators

There is no option that exists, which can remove an individual validator. Use clearValidators to remove all the validators of a control.

|  |  |
| --- | --- |
| 1  2  3 | this.myForm.controls['controlName'].clearValidators() |

Update Validation Status

Removing or adding the validators does not change the validity status of the form or the control immediately. The Validators run only when we change the value of the field.

We can force angular to run the validations using the updateValueAndValidity method.

|  |  |
| --- | --- |
| 1  2  3 | this.myForm.controls['controlName'].updateValueAndValidity() |

SetValidators Example

The following example, shows how to use the SetValidators in Angular

We have two fields email & mobile.

The user needs to choose, how he wants the system to notify him, using the drop-down field notifyVia. The drop-down has two options email & Mobile.

If the user chooses email, then we need to make the email field as a Required field. If he chooses the Mobile, then we must make the mobile field as Required field.

We subscribe to the [valueChanges](https://www.tektutorialshub.com/angular/valuechanges-in-angular-forms/) event of the notifyVia to listen for changes and invoke the changeValidators method.

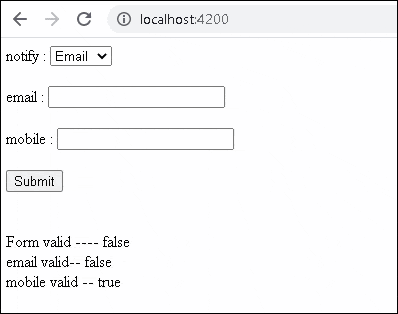
In the changeValidators method, we check the value of notifyVia and add or remove the required validator using the setValidators. We also add the email validator (for email field) or MinLength validator (for mobile field). To remove the validator, we use the method clearValidators()

Finally, we use the updateValueAndValidity method, which forces the angular to update the validity status of the control.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | import { Component } from '@angular/core';  import { FormBuilder, FormGroup, FormControl, Validators } from '@angular/forms';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',  })  export class AppComponent {    title = 'setValidators';      myform:FormGroup;      notifyOptions = ["Email" ,"SMS"]      constructor(private fb: FormBuilder) {        this.myform = this.fb.group({        email: new FormControl(''),        mobile: new FormControl(''),        notifyVia: new FormControl('',Validators.required),      });        this.myform.get("notifyVia").valueChanges        .subscribe(data=> {          this.changeValidators()        })    }        changeValidators() {        console.log(this.myform.get("notifyVia").value)        if (this.myform.get("notifyVia").value=="Email") {        this.myform.controls["email"].setValidators([Validators.required,Validators.email]);        this.myform.controls["mobile"].clearValidators();      } else {        this.myform.controls["email"].clearValidators();        this.myform.controls["mobile"].setValidators([Validators.required,Validators.minLength(10)]);      }        this.myform.get("email").updateValueAndValidity();      this.myform.get("mobile").updateValueAndValidity();        }  } |

The component template

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | <form [formGroup]="myform">      notify :  <select formControlName="notifyVia">    <option \*ngFor="let item of notifyOptions" [ngValue]="item">{{item}}</option>  </select>  <br>  <br>      email :  <input type="text" formControlName= "email"/>  <br>  <br>  mobile :  <input type="text" formControlName= "mobile"/>  <br>  <br>    <button type="submit" >Submit </button>  </form>    <br>  <br>    Form valid ---- {{myform.valid}}  <br>    email valid-- {{myform.controls['email'].valid}}  <br>    mobile valid -- {{myform.controls['mobile'].valid}}  <br> |



# HTTP

Using Angular HttpClient

The HttpClient is a separate model in Angular and is available under the @angular/common/http package. The following steps show you how to use the HttpClient in an Angular app.

Import HttpClient Module in Root Module

We need to import it into our root module app.module. Also, we need to add it to the imports metadata array.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import { NgModule } from '@angular/core';  import { HttpClientModule } from '@angular/common/http';    @NgModule({      declarations: [          AppComponent      ],      imports: [          HttpClientModule      ],      providers: [],      bootstrap: [AppComponent]  })  export class AppModule { } |

Import Required Module in Component/Service

Then you should import HttpClient the @angular/common/http in the component or service.

|  |  |
| --- | --- |
| 1  2  3 | import { HttpClient } from '@angular/common/http'; |

Inject HttpClient service

Inject the HttpClient service in the constructor.

|  |  |
| --- | --- |
| 1  2  3  4 | constructor(public http: HttpClient) {  } |

Call the HttpClient.Get method

Use HttpClient.Get method to send an [HTTP Request](https://www.tektutorialshub.com/http/http-get-and-post-methods/). The request is sent when we Subscribe to the get() method. When the response arrives map it the desired object and display the result.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public getData() {    this.HttpClient.get<any[]>(this.baseUrl+'users/'+this.userName+'/repos')             .subscribe(data => {                 this.repos= data;             },             error => {             }    );  } |

What is Observable?

The Angular HTTPClient makes use of [observable](https://www.tektutorialshub.com/angular/angular-observable-tutorial-using-rxjs/). Hence it is important to understand the basics of it

Observable help us to manage async data. You can think of Observables as an array of items, which arrive asynchronously over time.

The observables implement the [observer design pattern](https://en.wikipedia.org/wiki/Observer_pattern), where observables maintain a list of dependents. We call these dependents as observers. The observable notifies them automatically of any state changes, usually by calling one of their methods.

Observer subscribes to an Observable. The observer reacts when the value of the Observable changes. An Observable can have multiple subscribers and all the subscribers are notified when the state of the Observable changes.

When an Observer subscribes to an observable, it needs to pass (optional) the three callbacks. next(),  error()  &  complete(). The observable invokes the next() callback, when it receives an value. When the observable completes it invokes the complete() callback. And when the error occurs it invokes the error() callback with details of error and subscriber finishes.

The Observables are used extensively in Angular. The new HTTPClient Module and Event system are all Observable based.

The Observables are proposed feature for the next version of Javascript. The Angular uses a Third-party library called [Reactive Extensions](http://reactivex.io/) or RxJs to implement the Observables. You can learn about RxJs from these [RxJx tutorials](http://reactivex.io/tutorials.html)

Observables Operators

Operators are methods that operate on an Observable and return a new observable. Each Operator modifies the value it receives. These operators are applied one after the other in a chain.

The RxJs provides several Operators, which allows you to filter, select, transform, combine and compose Observables. Examples of Operators are map, filter, take, merge, etc

How to use RxJs

The RxJs is a very large library. Hence Angular exposes a stripped-down version of Observables. You can import it using the following import statement

|  |  |
| --- | --- |
| 1  2  3 | import { Observable } from 'rxjs'; |

The above import imports only the necessary features. It does not include any of the Operators.

To use observables operators, you need to import them. The following code imports the map & catchError operators.

|  |  |
| --- | --- |
| 1  2  3 | import { map, catchError } from 'rxjs/operators'; |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

HTTP GET

The HttpClient.get sends the HTTP Get Request to the API endpoint and parses the returned result to the desired type. By default, the body of the response is parsed as JSON. If you want any other type, then you need to specify explicitly using the observe & responseType options.

You can read more about [Angular HTTP Get](https://www.tektutorialshub.com/angular/angular-http-get-example-using-httpclient/)

Syntax

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | get(url: string,        options: {            headers?: HttpHeaders | { [header: string]: string | string[]; };            params?: HttpParams | { [param: string]: string | string[]; };            observe?: "body|events|response|";            responseType: "arraybuffer|json|blob|text";            reportProgress?: boolean;            withCredentials?: boolean;}       ): Observable<> |

Options

under the options, we have several configuration options, which we can use to configure the request.

headers It allows you to add HTTP headers to the outgoing requests.

observe The HttpClient.get method returns the body of the response parsed as JSON (or type specified by the responseType). Sometimes you may need to read the entire response along with the headers and status codes. To do this you can set the observe property to the **response**.

The allowed options are

a response which returns the entire response

body which returns only the body

events which return the response with events.

params Allows us to Add the [URL parameters or Get Parameters](https://www.tektutorialshub.com/angular/angular-pass-url-parameters-query-strings/) to the Get Request

reportProgress This is a boolean property. Set this to true, if you want to get notified of the progress of the Get Request. This is a pretty useful feature when you have a large amount of data to download (or upload) and you want the user to notify of the progress.

responseType Json is the default response type. In case you want a different type of response, then you need to use this parameter. The Allowed Options are arraybuffer, blob, JSON, and text.

withCredentials It is of boolean type. If the value is true then HttpClient.get will request data with credentials (cookies)

HTTP Post

The HttpClient.post() sends the HTTP POST request to the endpoint. Similar to the get(), we need to subscribe to the post() method to send the request. The post method parsed the body of the response as JSON and returns it. This is the default behavior. If you want any other type, then you need to specify explicitly using the observe & responseType options.

You can read [Angular HTTP Post](https://www.tektutorialshub.com/angular/angular-http-post-example/)

The syntax of the HTTP Post is similar to the HTTP Get.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | post(url: string,       body: any,       options: {          headers?: HttpHeaders | { [header: string]: string | string[]; };          observe?: "body|events|response|";          params?: HttpParams | { [param: string]: string | string[]; };          reportProgress?: boolean;          responseType: "arraybuffer|json|blob|text";          withCredentials?: boolean;       }  ): Observable |

The following is an example of HTTP Post

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | addPerson(person:Person): Observable<any> {      const headers = { 'content-type': 'application/json'}      const body=JSON.stringify(person);      this.http.post(this.baseURL + 'people', body,{'headers':headers , observe: 'response'})        .subscribe(         response=> {              console.log("POST completed sucessfully. The response received "+response);          },          error => {              console.log("Post failed with the errors");          },          () => {              console.log("Post Completed");          }  } |

HTTP PUT

The HttpClient.put() sends the HTTP PUT request to the endpoint. The syntax and usage are very similar to the HTTP POST method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | put(url: string,       body: any,       options: {          headers?: HttpHeaders | { [header: string]: string | string[]; };          observe?: "body|events|response|";          params?: HttpParams | { [param: string]: string | string[]; };          reportProgress?: boolean;          responseType: "arraybuffer|json|blob|text";          withCredentials?: boolean;       }  ): Observable |

HTTP PATCH

The HttpClient.patch() sends the HTTP PATCH request to the endpoint. The syntax and usage are very similar to the HTTP POST method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | patch(url: string,       body: any,       options: {          headers?: HttpHeaders | { [header: string]: string | string[]; };          observe?: "body|events|response|";          params?: HttpParams | { [param: string]: string | string[]; };          reportProgress?: boolean;          responseType: "arraybuffer|json|blob|text";          withCredentials?: boolean;       }  ): Observable |

HTTP DELETE

The HttpClient.delete() sends the HTTP DELETE request to the endpoint. The syntax and usage are very similar to the HTTP GET method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | delete(url: string,        options: {            headers?: HttpHeaders | { [header: string]: string | string[]; };            params?: HttpParams | { [param: string]: string | string[]; };            observe?: "body|events|response|";            responseType: "arraybuffer|json|blob|text";            reportProgress?: boolean;            withCredentials?: boolean;}       ): Observable<> |

HttpClient Example

Now, We have a basic understanding of HttpClient model & observables, let us build an HttpClient example app.

Create a new Angular app

|  |  |
| --- | --- |
| 1  2  3 | ng new httpClient |

import HttpClientModule

In the app,module.ts import the HttpClientModule module as shown below. We also add it to the imports array

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { HttpClientModule } from '@angular/common/http';  import { AppRoutingModule } from './app-routing.module';  import { AppComponent } from './app.component';    @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule,      AppRoutingModule,      HttpClientModule    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

Component

Now, open the app.component.ts and copy the following code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | import { Component, OnInit } from '@angular/core';  import { HttpClient } from '@angular/common/http';    export class Repos {    id: string;    name: string;    html\_url: string;    description: string;  }    @Component({    selector: 'app-root',    templateUrl: './app.component.html',  })  export class AppComponent implements OnInit {      userName: string = "tektutorialshub"    baseURL: string = "https://api.github.com/";    repos: Repos[];        constructor(private http: HttpClient) {    }      ngOnInit() {      this.getRepos()    }        public getRepos() {        return this.http.get<Repos[]>(this.baseURL + 'users/' + this.userName + '/repos')        .subscribe(          (response) => {                           //Next callback            console.log('response received')            console.log(response);            this.repos = response;          },          (error) => {                              //Error callback            console.error('Request failed with error')            alert(error);          },          () => {                                   //Complete callback            console.log('Request completed')          })    }  } |

Import HTTPClient

HTTPClient is a service, which is a major component of the HTTP Module. It contains methods like GET, POST, PUT etc. We need to import it.

|  |  |
| --- | --- |
| 1  2  3 | import { HttpClient } from '@angular/common/http'; |

Repository Model

The model to handle our data.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | export class Repos {    id: string;    name: string;    html\_url: string;    description: string;  } |

Inject HttpClient

Inject the HttpClient service into the component. You can learn more about [Dependency injection in Angular](https://www.tektutorialshub.com/angular/angular-dependency-injection/)

|  |  |
| --- | --- |
| 1  2  3  4 | constructor(private http: HttpClient) {    } |

Subscribe to HTTP Get

The GetRepos method, we invoke the get() method of the HttpClient Service.

The HttpClient.get method allows us to cast the returned response object to a type we require. We make use of that feature and supply the type for the returned value http.get<repos[]>

The get() method returns an observable. Hence we subscribe to it.

|  |  |
| --- | --- |
| 1  2  3  4  5 | public getRepos() {     return this.http.get<Repos[]>(this.baseURL + 'users/' + this.userName + '/repos')        .subscribe( |

When we subscribe to any observable, we optionally pass the three callbacks. next(),  error()  &  complete(). In this example we pass only two callbacks next() & error().

Receive the Response

We receive the data in the next() callback. By default, the Angular reads the body of the response as JSON and casts it to an object and returns it back. Hence we can use directly in our app.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | (response) => {                           //Next callback            console.log('response received')            console.log(response);            this.repos = response;          }, |

Handle the errors

We handle the errors in error callback.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | (error) => {                              //Error callback            console.error('Request failed with error')            alert(error);          }, |

Template

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | <h1 class="heading"><strong>HTTPClient </strong> Example</h1>      <table class='table'>    <thead>      <tr>        <th>ID</th>        <th>Name</th>        <th>HTML Url</th>        <th>description</th>      </tr>    </thead>    <tbody>      <tr \*ngFor="let repo of repos;">        <td>{{repo.id}}</td>        <td>{{repo.name}}</td>        <td>{{repo.html\_url}}</td>        <td>{{repo.description}}</td>      </tr>    </tbody>  </table>    <pre>{{repos | json}}</pre> |

Finally, we display the list of GitHub Repos using the[ngFor Directive](https://www.tektutorialshub.com/angular/angular-ngfor-directive/)

Summary

We learned how to create a simple HTTP Service in this tutorial. We also looked at the basics of Observables which is not extensively used in Angular

HTTP Get Example

Create a new Angular App.

|  |  |
| --- | --- |
| 1  2  3 | ng new httpGet |

Import HttpClientModule

To make HTTP Get request, we need to make use of the [HttpClientModule](https://angular.io/api/common/http/HttpClientModule), which is part of the package @angular/common/http. Open the app.module.ts and import it. Also, import the FormsModule

You must also include it in the the imports array as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { HttpClientModule } from '@angular/common/http';  import { FormsModule } from '@angular/forms'    import { AppRoutingModule } from './app-routing.module';  import { AppComponent } from './app.component';  import { GitHubService } from './github.service';    @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule,      AppRoutingModule,      HttpClientModule,      FormsModule    ],    providers: [GitHubService],    bootstrap: [AppComponent]  })  export class AppModule { } |

Model

Create repos.ts file and add the following code. This is a simplified model for the GitHub repository.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | export class repos {      id: string;      name: string;      html\_url: string;      description: string;  } |

HTTP GET Service

Let us create a service to handle the HTTP Request. Create a new file github.service.ts and copy the following code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import { Injectable } from '@angular/core';    import { HttpClient, HttpParams, HttpHeaders } from '@angular/common/http';  import { Observable, throwError } from 'rxjs';  import { map, catchError} from 'rxjs/operators';    import { repos } from './repos';    @Injectable()  export class GitHubService {      baseURL: string = "https://api.github.com/";      constructor(private http: HttpClient) {    }      getRepos(userName: string): Observable<any> {      return this.http.get(this.baseURL + 'users/' + userName + '/repos')    }    } |

First, we import the required libraries. The [HttpClient](https://angular.io/api/common/http/HttpClient) is the main service, which Performs the HTTP requests like GET, PUT, POST, etc. We need to inject this into our GitHubService. Also, import [HttpParams](https://www.tektutorialshub.com/angular/angular-pass-url-parameters-query-strings/) which helps us to [add Query Parameters in an HTTP Request](https://www.tektutorialshub.com/angular/angular-pass-url-parameters-query-strings/). Import HTTP Headers using the HttpHeaders which allows us to add HTTP Headers to the request.

|  |  |
| --- | --- |
| 1  2  3 | import { HttpClient, HttpParams, HttpHeaders } from '@angular/common/http'; |

The HttpClient service makes use of [RxJs observable](https://www.tektutorialshub.com/angular/angular-observable-tutorial-using-rxjs/), Hene we import Observable, throwError & RxJs Operators like map & catchError

|  |  |
| --- | --- |
| 1  2  3  4 | import { Observable, throwError } from 'rxjs';  import { map, catchError} from 'rxjs/operators'; |

The URL endpoint is hardcoded in our example, But you can make use of a [config file](https://www.tektutorialshub.com/angular/angular-runtime-configuration/) to store the value and read it using the [APP\_INITIALIZER](https://www.tektutorialshub.com/angular/angular-how-to-use-app-initializer/) token

|  |  |
| --- | --- |
| 1  2  3 | baseURL: string = "https://api.github.com/"; |

We inject the HttpClient using the [Dependency Injection](https://www.tektutorialshub.com/angular/angular-dependency-injection/)

|  |  |
| --- | --- |
| 1  2  3  4 | constructor(private http: HttpClient) {    } |

Finally, we use the get method of the httpclient to make an HTTP Get request to GitHub.

The https://api.github.com/users/<username>?repos endpoint returns the list of Repositories belonging to the user <userName>

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | //Any Data Type    getRepos(userName: string): Observable<any> {      return this.http.get(this.baseURL + 'users/' + userName + '/repos')    } |

Note that httpclient.getmethod returns the observable. Hence we need to subscribe to it to get the data.

Component

The following is the code from app.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | import { Component } from '@angular/core';    import { GitHubService } from './github.service';  import { repos } from './repos';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',  })  export class AppComponent {    userName: string = "tektutorialshub"    repos: repos[];      loading: boolean = false;    errorMessage;      constructor(private githubService: GitHubService) {    }      public getRepos() {      this.loading = true;      this.errorMessage = "";      this.githubService.getRepos(this.userName)        .subscribe(          (response) => {                           //next() callback            console.log('response received')            this.repos = response;          },          (error) => {                              //error() callback            console.error('Request failed with error')            this.errorMessage = error;            this.loading = false;          },          () => {                                   //complete() callback            console.error('Request completed')      //This is actually not needed            this.loading = false;          })    }  } |

We subscribe to the getRepos() method in our component class. Only when we subscribe to the observable, the HTTP GET request is sent to the back end server.

|  |  |
| --- | --- |
| 1  2  3  4 | this.githubService.getRepos(this.userName)        .subscribe(); |

When we subscribe to any observable, we optionally pass the three callbacks. next(),  error()  &  complete().

Next() callback is where we get the result of the observable. In this example the list of repositories for the given user.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | (response) => {                           //next() callback            console.log('response received')            this.repos = response;          } |

The observable can also result in an error. It will invoke the error() callback and pass the error object. The observables stop after emitting the error signal.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | (error) => {                              //error() callback            console.error('Request failed with error')            this.errorMessage = error;            this.loading = false;          }, |

When the observable completes, it will call the complete() callback. **There is no need for this call back as the subscription completes when the data is received.**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | () => {                                   //complete() callback            console.log('Request completed')            this.loading = false;          }) |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Loading Indicator

We create a variable loading=true just before subscribing to the GETrequest. When the observable completes or an error occurs, we make it false. This helps us to show some kind of loading indicator to users, while we wait for the response.

Template

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | <h1 class="heading"><strong>HTTP </strong>Demo</h1>    <div class="form-group">    <label for="userName">GitHub User Name</label>    <input type="text" class="form-control" name="userName" [(ngModel)]="userName">  </div>    <div class="form-group">    <button type="button" (click)="getRepos()">Get Repos</button>  </div>    <div \*ngIf="loading">loading...</div>    <div \*ngIf="errorMessage" class="alert alert-warning">    <strong>Warning!</strong> {{errorMessage | json}}  </div>      <table class='table'>    <thead>      <tr>        <th>ID</th>        <th>Name</th>        <th>HTML Url</th>        <th>description</th>      </tr>    </thead>    <tbody>      <tr \*ngFor="let repo of repos;">        <td>{{repo.id}}</td>        <td>{{repo.name}}</td>        <td>{{repo.html\_url}}</td>        <td>{{repo.description}}</td>      </tr>    </tbody>  </table>    <pre>{{repos | json}}</pre> |

The template is very simple

We first ask for the userName. We use the two-way data binding to sync userName [(ngModel)]="userName" with the userName property in the component class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <div class="form-group">    <label for="userName">GitHub User Name</label>    <input type="text" class="form-control" name="userName" [(ngModel)]="userName">  </div> |

getRepos() method subscribes to the HTTP get method.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div class="form-group">    <button type="button" (click)="getRepos()">Get Repos</button>  </div> |

We show a loading message until the observable returns response or an error.

|  |  |
| --- | --- |
| 1  2  3 | <div \*ngIf="loading">loading...</div> |

Show the error message.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div \*ngIf="errorMessage" class="alert alert-warning">    <strong>Warning!</strong> {{errorMessage | json}}  </div> |

The last line shows the response as it is received.

|  |  |
| --- | --- |
| 1  2  3 | <pre>{{repos | json}}</pre> |

HTTP Get in Action

Now, run the app, you should able to make a successful GET Request.



Get Syntax

The above code is a very simple example of the HTTP get() method. The complete syntax of the get() method is as shown below. It has second argument options, where we can pass the HTTP headers, parameters, and other options to control how the get() method behaves.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | get(url: string,        options: {            headers?: HttpHeaders | { [header: string]: string | string[]; };            params?: HttpParams | { [param: string]: string | string[]; };            observe?: "body|events|response|";            responseType: "arraybuffer|json|blob|text";            reportProgress?: boolean;            withCredentials?: boolean;}       ): Observable<> |

headers : use this to send the HTTP Headers along with the request

params: set query strings / URL parameters

observe: This option determines the return type.

responseType: The value of responseType determines how the response is parsed.

reportProgress: Whether this request should be made in a way that exposes [progress events](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/readyState).

withCredentials: Whether this request should be sent with outgoing credentials (cookies).

observe

The GET method returns one of the following

Complete response

body of the response

[events](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/readyState).

By default, it returns the body as shown in our example app.

Complete Response

The following code will return the complete response and not just the body

|  |  |
| --- | --- |
| 1  2  3  4  5 | getReposRawResponse(userName: string): Observable<any> {    return this.http.get(this.baseURL + 'users/' + userName + '/repos', { observe: 'response' })  } |

The complete response is as follows.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | {    "headers": {      "normalizedNames": {},      "lazyUpdate": null    },    "status": 200,    "statusText": "OK",    "url": "https://api.github.com/users/tektutorialshub/repos",    "ok": true,    "type": 4,    "body": [      {        "id": 102269857,          \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Removed for clarity \*\*\*\*\*      }    ]  } |

events

You can also listen to progress events by using the { observe: 'events', reportProgress: true }. You can read about [observe the response](https://brianflove.com/2018-09-03/angular-http-client-observe-response/)

|  |  |
| --- | --- |
| 1  2  3  4  5 | getReposRawResponse(userName: string): Observable<any> {    return this.http.get(this.baseURL + 'users/' + userName + '/repos', { observe: 'events', reportProgress: true })  } |

Response Type

The responseType determines how the response is parsed. it can be one of the arraybuffer, json blob or text. The default behavior is to parse the response as JSON.

Strongly typed response

Instead of any, we can also use a type as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5 | getReposTypedResponse(userName: string): Observable<repos[]> {      return this.http.get<repos[]>(this.baseURL + 'users/' + userName + '/repos')    } |

String as Response Type

The API may return a simple text rather than a JSON. Use responsetype: 'text' to ensure that the response is parsed as a string.

|  |  |
| --- | --- |
| 1  2  3  4  5 | getReposTypedResponse(userName: string): Observable<repos[]> {      return this.http.get<repos[]>(this.baseURL + 'users/' + userName + '/repos', { responsetype: 'text'})    } |

Catching Errors

The API might fail with an error. You can catch those errors using catchError. You either handle the error or throw it back to the component using the throw err

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | getReposCatchError(userName: string): Observable<repos[]> {      return this.http.get<repos[]>(this.baseURL + 'usersY/' + userName + '/repos')        .pipe(          catchError((err) => {            console.error(err);            throw err;          }          )        )    } |

Read more about error handling from [Angular HTTP interceptor error handling](https://www.tektutorialshub.com/angular/angular-http-error-handling/)

Transform the Response

You can make use of the map, filter RxJs Operators to manipulate or transform the response before sending it to the component.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | getReposMap(userName: string): Observable<repos[]> {      return this.http.get<repos[]>(this.baseURL + 'users/' + userName + '/repos')        .pipe(          map((data) => {             //You can perform some transformation here             return data;          }),          catchError((err, caught) => {            console.error(err);            throw err;          }          )        )    } |

URL Parameters

The [URL Parameters or Query strings](https://www.tektutorialshub.com/angular/angular-pass-url-parameters-query-strings/) can be added to the request easily using the [HttpParams](https://www.tektutorialshub.com/angular/angular-pass-url-parameters-query-strings/) option. All you need to do is to create a new HttpParams class and add the parameters as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | //URL Parameter    getReposUrlParameter(userName: string): Observable<repos[]> {        const params = new HttpParams()        .set('sort', "description")        .set('page',"2");        return this.http.get<repos[]>(this.baseURL + 'users/' + userName + '/repos', { 'params': params })        .pipe(          map((response) => {            return response;          }),          catchError((err, caught) => {            console.error(err);            throw err;          }          )        )    } |

The above code sends the GET request to the URL https://api.github.com/users/tektutorialshub/repos?sort=description&page=2

The following code also works.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | getReposUrlParameter(userName: string): Observable<repos[]> {      return this.http.get<repos[]>(this.baseURL + 'users/' + userName + '/repos?sort=description&page=2')        .pipe(          map((response) => {            return response;          }),          catchError((err, caught) => {            console.error(err);            throw err;          }          )        )    } |

HTTP Headers

You can also add HTTP Headers using the HttpHeaders option as shown below. You can make use of the [Http Interceptor to set the common headers](https://www.tektutorialshub.com/angular/angular-httpclient-http-interceptor/).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | //HTTP Headers    getReposHeaders(userName: string): Observable<repos[]> {        const params = new HttpParams()        .set('sort', "description")        .set('page',"2");        const headers = new HttpHeaders()        .set('Content-Type', 'application/json')          return this.http.get<repos[]>(this.baseURL + 'users/' + userName + '/repos', { 'params': params, 'headers': headers })        .pipe(          map((response) => {            return response;          }),          catchError((err, caught) => {            console.error(err);            throw err;          }          )        )    } |

Send Cookies

You can send cookies with every request using the withCredentials=true as shown below. You can make use of the[Http Interceptor](https://www.tektutorialshub.com/angular/angular-httpclient-http-interceptor/) to set the withCredentials=true for all requests.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | //With Credentials      getReposWithCookies(userName: string): Observable<repos[]> {          const params = new HttpParams()          .set('sort', "description")          .set('page',"2");          const headers = new HttpHeaders()          .set('Content-Type', 'application/json')            return this.http.get<repos[]>(this.baseURL + 'users/' + userName + '/repos', { 'params': params, 'headers': headers, withCredentials: true })          .pipe(            map((response) => {              return response;            }),            catchError((err, caught) => {              console.error(err);              throw err;            }            )          )      } |

Summary

This guide explains how to make use of HTTP get in Angular using an example app HTTP Post Example

Create a new Angular App.

|  |  |
| --- | --- |
| 1  2  3 | ng new httpPost |

Import HttpClientModule

Import the HttpClientModule & FormsModule in app.module.ts. Also, add it to the imports array.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';    import { HttpClientModule } from '@angular/common/http';  import { FormsModule } from '@angular/forms'    import { AppRoutingModule } from './app-routing.module';  import { AppComponent } from './app.component';    @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule,      AppRoutingModule,      HttpClientModule,      FormsModule,    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

Faking Backend

In the [HTTP Get example](https://www.tektutorialshub.com/angular/angular-http-get-example-using-httpclient/), we made use of the publicly available GitHub API. For this example, we need a backend server, which will accept the post request.

There are few ways to create a fake backend. You can make use of an [in-memory web API](https://github.com/angular/in-memory-web-api) or the [JSON server](https://github.com/typicode/json-server). For this tutorial, we will make use of the JSON Server.

Install the JSON-server globally using the following npm command

|  |  |
| --- | --- |
| 1  2  3 | npm install -g json-server |

create a db.json file with some data. The following example contains data of people with id & name fields.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | {    "people": [      {        "id": 1,        "name": "Don Bradman"      },      {        "id": 2,        "name": "Sachin Tendulkar"      }    ]  } |

Start the server

|  |  |
| --- | --- |
| 1  2  3 | json-server --watch db.json |

The json-server starts and listens for requests on port 3000.

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**



Browse the URL <http://localhost:3000/> and you should be able to see the home page

The URL<http://localhost:3000/people> lists the people from the db.json. You can now make GET POST PUT PATCH DELETE OPTIONS against this URL

Model

Now, back to our app and create a Person model class under person.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | export class Person {    id:number    name:string  } |

HTTP Post Service

Now, let us create a Service, which is responsible to send HTTP Requests. Create a new file api.service.ts and copy the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | import { HttpClient, HttpHeaders } from '@angular/common/http';  import { Person } from './person';  import { Injectable } from '@angular/core';  import { Observable } from 'rxjs';    @Injectable({providedIn:'root'})  export class ApiService {      baseURL: string = "http://localhost:3000/";      constructor(private http: HttpClient) {    }      getPeople(): Observable<Person[]> {      console.log('getPeople '+this.baseURL + 'people')      return this.http.get<Person[]>(this.baseURL + 'people')    }      addPerson(person:Person): Observable<any> {      const headers = { 'content-type': 'application/json'}      const body=JSON.stringify(person);      console.log(body)      return this.http.post(this.baseURL + 'people', body,{'headers':headers})    }    } |

The URL endpoint of our json-server is hardcoded in our example, But you can make use of a [config file](https://www.tektutorialshub.com/angular/angular-runtime-configuration/) to store the value and read it using the [APP\_INITIALIZER](https://www.tektutorialshub.com/angular/angular-how-to-use-app-initializer/) token

|  |  |
| --- | --- |
| 1  2  3 | baseURL: string = "http://localhost:3000/"; |

We inject the HttpClient using the [Dependency Injection](https://www.tektutorialshub.com/angular/angular-dependency-injection/)

|  |  |
| --- | --- |
| 1  2  3  4 | constructor(private http: HttpClient) {  } |

The getPeople() method sends an [HTTP GET](https://www.tektutorialshub.com/angular/angular-http-get-example-using-httpclient/) request to get the list of persons. Refer to the tutorial [Angular HTTP GET Example](https://www.tektutorialshub.com/angular/angular-http-get-example-using-httpclient/) to learn more.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | getPeople(): Observable<Person[]> {      console.log('getPeople '+this.baseURL + 'people')      return this.http.get<Person[]>(this.baseURL + 'people')    } |

In the addPerson method, we send an HTTP POST request to insert a new person in the backend.

Since we are sending data as JSON, we need to set the 'content-type': 'application/json' in the HTTP header. The JSON.stringify(person) converts the person object into a JSON string.

Finally, we use the http.post() method using URL, body & headers as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | addPerson(person:Person): Observable<any> {      const headers = { 'content-type': 'application/json'}      const body=JSON.stringify(person);      console.log(body)      return this.http.post(this.baseURL + 'people', body,{'headers':headers})    } |

The post() method returns an [observable](https://www.tektutorialshub.com/angular/angular-observable-tutorial-using-rxjs/). Hence we need to subscribe to it.

Component

Template

The template is very simple.

We ask for the name of the person, which we want to add to our backend server. The two-way data binding ([(ngModel)]="person.name") keeps the person object in sync with the view.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | <h1>{{title}}</h1>    <div>    <div>      <label>Name: </label>      <input [(ngModel)]="person.name" />    </div>    <div>      <button (click)="addPerson()">Add</button>    </div>  </div>    <table class='table'>    <thead>      <tr>        <th>ID</th>        <th>Name</th>      </tr>    </thead>    <tbody>      <tr \*ngFor="let person of people;">        <td>{{person.id}}</td>        <td>{{person.name}}</td>      </tr>    </tbody>  </table> |

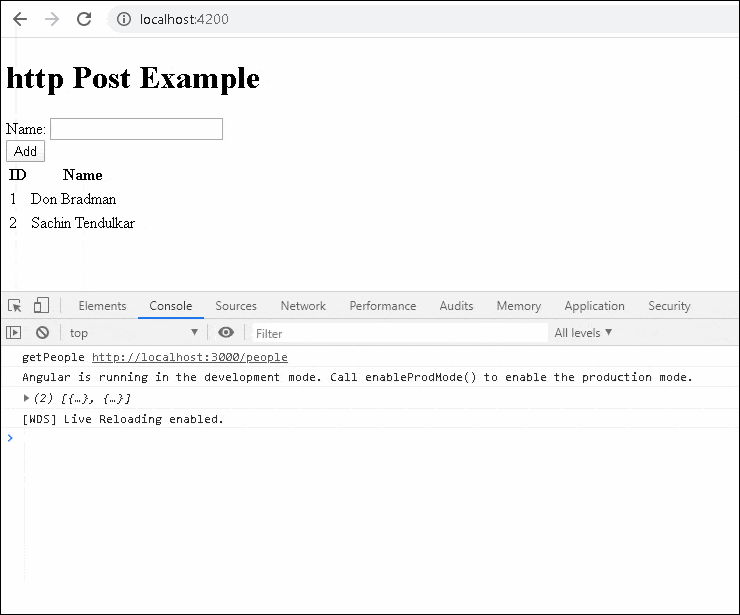
Code

In the refreshPeople() method, we subscribe to the getPeople() method of our ApiService to make an [HTTP get() request](https://www.tektutorialshub.com/angular/angular-http-get-example-using-httpclient/) to get the list of people.

Under the addPerson() method, we subscribe to the apiService.addPerson(). Once the post request finishes, we call refreshPeople() method to get the updated list of people.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | import { Component, OnInit } from '@angular/core';  import { ApiService } from './api.service';  import { Person } from './person';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {      title = 'httpGet Example';    people:Person[];    person = new Person();      constructor(private apiService:ApiService) {}      ngOnInit() {      this.refreshPeople()    }      refreshPeople() {      this.apiService.getPeople()        .subscribe(data => {          console.log(data)          this.people=data;        })      }      addPerson() {      this.apiService.addPerson(this.person)        .subscribe(data => {          console.log(data)          this.refreshPeople();        })    }    } |

HTTP Post in Action



HTTP Post syntax

The above code is a very simple example of the HTTP post() method. The complete syntax of the post() method is as shown below. The first two arguments are URL and body. It has the third argument options, where we can pass the HTTP headers, parameters, and other options to control how the post() method behaves.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | post(url: string,       body: any,       options: {          headers?: HttpHeaders | { [header: string]: string | string[]; };          observe?: "body|events|response|";          params?: HttpParams | { [param: string]: string | string[]; };          reportProgress?: boolean;          responseType: "arraybuffer|json|blob|text";          withCredentials?: boolean;       }  ): Observable |

headers : use this to send the HTTP Headers along with the request

params: set query strings / URL parameters

observe: This option determines the return type.

responseType: The value of responseType determines how the response is parsed.

reportProgress: Whether this request should be made in a way that exposes [progress events](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/readyState).

withCredentials: Whether this request should be sent with outgoing credentials (cookies).

observe

The POST method returns one of the following

Complete response

body of the response

[events](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/readyState).

By default, it returns the body.

Complete Response

The following code will return the complete response and not just the body

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | addPerson(person:Person): Observable<any> {      const headers = { 'content-type': 'application/json'}      const body=JSON.stringify(person);        return this.http.post(this.baseURL + 'people', body,{'headers':headers , observe: 'response'})  } |

events

You can also listen to progress events by using the { observe: 'events', reportProgress: true }. You can read about [observe the response](https://brianflove.com/2018-09-03/angular-http-client-observe-response/)

|  |  |
| --- | --- |
| 1  2  3 | return this.http.post(this.baseURL + 'people', body,{'headers':headers, observe: 'response',reportProgress: true}) |

Response Type

The responseType determines how the response is parsed. it can be one of the arraybuffer, json blob or text. The default behavior is to parse the response as JSON.

Strongly typed response

Instead of any, we can also use a type as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | addPerson(person:Person): Observable<Person> {      const headers = { 'content-type': 'application/json'}      const body=JSON.stringify(person);      console.log(body)      return this.http.post<Person>(this.baseURL + 'people', body,{'headers':headers})    } |

String as Response Type

The API may return a simple text rather than a JSON. Use responsetype: 'text' to ensure that the response is parsed as a string.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | addPerson(person:Person): Observable<Person> {      const headers = { 'content-type': 'application/json'}      const body=JSON.stringify(person);        return this.http.post<Person>(this.baseURL + 'people', body,{'headers':headers, responsetype: 'text'})    } |

Catching Errors

The API might fail with an error. You can catch those errors using catchError. You either handle the error or throw it back to the component using the throw err

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | addPerson(person:Person): Observable<Person> {      const headers = { 'content-type': 'application/json'}      const body=JSON.stringify(person);        return this.http.post<Person>(this.baseURL + 'people', body,{'headers':headers})         .pipe(           catchError((err) => {             console.error(err);             throw err;           }         )    } |

Read more about error handling from [Angular HTTP interceptor error handling](https://www.tektutorialshub.com/angular/angular-http-error-handling/)

Transform the Response

You can make use of the map, filter RxJs Operators to manipulate or transform the response before sending it to the component.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | addPerson(person:Person): Observable<Person> {      const headers = { 'content-type': 'application/json'}      const body=JSON.stringify(person);        return this.http.post<Person>(this.baseURL + 'people', body,{'headers':headers})         .pipe(            map((data) => {              //You can perform some transformation here             return data;           }),           catchError((err) => {             console.error(err);             throw err;           }         )    } |

URL Parameters

The [URL Parameters or Query strings](https://www.tektutorialshub.com/angular/angular-pass-url-parameters-query-strings/) can be added to the request easily using the [HttpParams](https://www.tektutorialshub.com/angular/angular-pass-url-parameters-query-strings/) option. All you need to do is to create a new HttpParams class and add the parameters as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | addPerson(person:Person): Observable<Person> {      const headers = { 'content-type': 'application/json'}       const params = new HttpParams()        .set('para1', "value1")        .set('para2',"value2");      const body=JSON.stringify(person);        return this.http.post<Person>(this.baseURL + 'people', body,{'headers':headers, 'params': params})      } |

The above code sends the GET request to the URL http://localhost:3000/people?para1=value1&para2=value2

The following code also works.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | addPerson(person:Person): Observable<Person> {      const headers = { 'content-type': 'application/json'}          const body=JSON.stringify(person);        return this.http.post<Person>(this.baseURL + 'people?para1=value1&para2=value2', body,{'headers':headers))      } |

HTTP Headers

You can also add HTTP Headers using the HttpHeaders option as shown below. You can make use of the [Http Interceptor to set the common headers](https://www.tektutorialshub.com/angular/angular-httpclient-http-interceptor/). Our example code already includes an HTTP header

Send Cookies

You can send cookies with every request using the withCredentials=true as shown below. You can make use of the[Http Interceptor](https://www.tektutorialshub.com/angular/angular-httpclient-http-interceptor/) to set the withCredentials=true for all requests.

|  |  |
| --- | --- |
| 1  2  3 | return this.http.post<Person>(this.baseURL + 'people?para1=value1&para2=value2', body,{'headers':headers, withCredentials=true)) |

. In the next tutorial, we will look at the HTTP post method.

The URL Parameters

In the [Angular HttpClient GET Example](https://www.tektutorialshub.com/angular/angular-httpclient/) article, we created a GitHubService. The Service issued GET Request to GitHub API Endpoint to retrieve the List of Repositories belonging to a particular User.

The GitHub API also has a set of parameters, which allows us to specify how we want to sort, which page to retrieve, No of Entries per page and type of the Repository to retrieve, etc.

For Example

|  |  |
| --- | --- |
| 1  2  3 | https://api.github.com/users/tekTutorialsHub/repos?sort=description&page=2 |

The Above query will return the result sorted on the description and retrieves only the second page. The string sort=description&page=2 after the question mark is called **URL Parameter** or **Query strings /Query Parameters**. The Question mark is used as a separator. The URL Parameters are also known as the **GET params**.

HttpParams()

We add the URL parameters using the helper class HttpParams.  The HttpParams is passed as one of the arguments to [HttpClient.get](https://www.tektutorialshub.com/angular/angular-http-get-example-using-httpclient/) method.

To use HttpParams, you need to import it first as shown below.

|  |  |
| --- | --- |
| 1  2  3 | import { HttpClient,HttpParams } from '@angular/common/http'; |

Then create an instance of the HttpParams class.

|  |  |
| --- | --- |
| 1  2  3  4  5 | const params = new HttpParams()    .set('page', PageNo)    .set('sort', SortOn); |

And then call the [httpClient.get](https://www.tektutorialshub.com/angular/angular-http-get-example-using-httpclient/) method passing the params as the argument.

|  |  |
| --- | --- |
| 1  2  3 | return this.httpClient.get<repos[]>(this.baseURL + 'users/' + userName + '/repos',{params}) |

The following are the list of method available in HttpParams class

HttpParams.set

|  |  |
| --- | --- |
| 1  2  3 | set(param: string, value: string): HttpParams |

Construct a new body with a new value for the given parameter name. If the parameter already exists then it is replaced else it is added.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | params = new HttpParams()      .set('page', '2')      .set('page', '3')      .set('sort', 'name');    console.log(params.toString()); //Returns page=3&sort=name |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

HTTPParams is immutable

The HttpParams object is immutable. Every time you call a set method on Params object, it will create and return a new instance of the Params.

For Example

The following code does not work

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | let params = new HttpParams();    params.set('page', PageNo);  params.set('sort', SortOn); |

Each call to set method does not add the options to the existing HttpParams instance, but creates a new instance from the existing instance and returns it.

To work around, you can use the code as follows

|  |  |
| --- | --- |
| 1  2  3  4  5 | Let params = new HttpParams()      .set('page', PageNo)      .set('sort', SortOn); |

Or you can try this

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | let params = new HttpParams();    params=params.set('page', PageNo);  params=params.set('sort', SortOn); |

HttpParams.append

|  |  |
| --- | --- |
| 1  2  3 | append(param: string, value: string): HttpParams |

Construct a new body with an appended value for the given parameter name. Always appends the value irrespective of whether the parameter exists. The page parameter is appended twice in the following example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | params = new HttpParams()      .set('page', '2')      .append('page', '3')      .set('sort', 'name');    console.log(params.toString()); //Returns page=2&page=3&sort=name |

The URL Is constructed as page=2&page=3&sort=name

You can also use the append method similar to the Set method

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | let params = new HttpParams();    params=params.append('page', PageNo);  params=params.append('sort', SortOn); |

HttpParams.has

|  |  |
| --- | --- |
| 1  2  3 | has(param: string): boolean |

Returns true if the given parameter name already exists in the HttpParams

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | params = new HttpParams()      .set('sort', 'name');    if (!params.has('page')) {      params = params.set('page', PageNo)  } |

HttpParams.get

|  |  |
| --- | --- |
| 1  2  3 | get(param: string): string | null |

Get the first value for the given parameter name, or null if it’s not present.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | params = new HttpParams()      .set('page', '2')      .append('page', '3')      .set('sort', 'name');    console.log(params.get('page')); // Returns 2 The First occurance of Page |

HttpParams.getAll

|  |  |
| --- | --- |
| 1  2  3 | getAll(param: string): string[] | null |

Get all values as for the given parameter name, or null if it’s not present.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | params = new HttpParams()      .set('page', '2')      .append('page', '3')      .set('sort', 'name');    console.log(params.getAll('page')); // Returns ["2", "3"] All occurance of Page |

HttpParams.keys

|  |  |
| --- | --- |
| 1  2  3 | keys(): string[] |

Get all the parameter names for this body.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | let params = new HttpParams()      .set('page', '2')      .set('sort', 'name');    console.log(params.keys()); //Returns ["page", "sort"] |

HttpParams.delete

|  |  |
| --- | --- |
| 1  2  3 | delete(param: string, value?: string): HttpParams |

Deletes the parameter and returns the new parameter collection.  You can delete using the parameter name or by using the name & value. If no argument is specified, then all parameters are deleted.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | params = new HttpParams()      .set('page', '2')      .Append('page', '3')      .set('sort', 'name');    params = params.delete('page', '3'); //Deletes the parameter page with value 3    params = params.delete('page'); //Delete the all the parameter of page    params = params.delete(''); //Delete all parameters |

HttpParams.toString

|  |  |
| --- | --- |
| 1  2  3 | toString(): string |

Serialize the body to an encoded string, where key-value pairs (separated by =) are separated by &s.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | params = new HttpParams()      .set('page', '2')      .Append('page', '3')      .set('sort', 'name');    console.log(params.toString()); //Returns page=2&page=3&sort=name |

Http Parameters from a string

Another way to pass the value is to use the fromString shortcut

|  |  |
| --- | --- |
| 1  2  3 | let params = new HttpParams({fromString: 'page=' + PageNo + '&sort=' + SortOn}); |

Http Parameters from an object

|  |  |
| --- | --- |
| 1  2  3 | let params = new HttpParams({ fromObject: { page: PageNo, sort: SortOn } }); |

Without params

You can also add the parameters directly to the URL, without going through the HttpParams as shown below.

|  |  |
| --- | --- |
| 1  2  3  4 | //You an also do it this way.  return this.httpClient.get<repos[]>(this.baseURL + 'users/' + userName + '/repos?'+'page='+PageNo+'&sort='+SortOn); |

Angular Httpparams Example

We are updating the project, which was created in the tutorial [Angular Http GET Example](https://www.tektutorialshub.com/angular/angular-http-get-example-using-httpclient/).

**app.module**

Import the httpClientModule from the @angular/common/http package.

|  |  |
| --- | --- |
| 1  2  3 | import {HttpClientModule} from '@angular/common/http'; |

Declare it in Imports metadata array in app.module.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule,      HttpClientModule,      FormsModule    ],    providers: [GitHubService],    bootstrap: [AppComponent]  }) |

Passing the URL Parameters

Open the github.service.ts.

Import the HttpClient & HttpParams from the @angular/common/http. We also require the Observable module from RxJs

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | import { Injectable } from '@angular/core';  import { HttpClient, HttpParams } from '@angular/common/http';  import { repos} from './repos';  import { Observable } from 'rxjs'; |

Inject HttpClient in the Constructor

|  |  |
| --- | --- |
| 1  2  3  4 | constructor(private httpClient:HttpClient){  } |

In the GetRepos method create the params object

|  |  |
| --- | --- |
| 1  2  3  4  5 | const params = new HttpParams()      .set('page', PageNo)      .set('sort', SortOn); |

And use the params when calling the [httpClient.get](https://www.tektutorialshub.com/angular/angular-http-get-example-using-httpclient/) method as shown below

|  |  |
| --- | --- |
| 1  2  3 | return this.httpClient.get<repos[]>(this.baseURL + 'users/' + userName + '/repos',{params}) |

The complete github.service.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | import { Injectable } from '@angular/core';  import { HttpClient, HttpParams } from '@angular/common/http';  import { Observable} from 'rxjs/Rx';    import { repos} from './repos';      @Injectable()  export class GitHubService {       baseURL= "https://api.github.com/";       constructor(private httpClient: HttpClient){     }       getRepos(userName: string, PageNo: string, SortOn: string): Observable<repos[]> {              let params = new HttpParams()                  .set('page', PageNo)                  .set('sort', SortOn);            console.log(params.toString());            return this.httpClient.get<repos[]>(this.baseURL + 'users/' + userName + '/repos', {params});     }    } |

**App.component.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | import { Component } from '@angular/core';  import { Observable} from 'rxjs/Rx';    import { GitHubService } from './github.service';    import { repos} from './repos';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',  })      export class AppComponent  {        userName = 'tektutorialshub';      pageNo  = '1';      sortOn = 'description';        repos: repos[];        loading = false;      errorMessage = '';        constructor(private githubService: GitHubService) {      }        public getRepos() {          this.loading = true;          this.errorMessage = '';          this.githubService.getRepos(this.userName,this.pageNo,this.sortOn)              .subscribe((response) => {this.repos = response;},                         (error) => {                             this.errorMessage = error.message; this.loading = false;                          },                          () => {this.loading = false;})        }    } |

App.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | <div class="container">        <h1 class="heading"><strong>HTTP </strong>Demo</h1>          <div class="form-group">         <label for="userName">User Name</label>         <input type="text" class="form-control" name="userName" [(ngModel)]="userName">           <label for="pageNo">Page No</label>         <input type="text" class="form-control" name="pageNo" [(ngModel)]="pageNo">           <label for="sortOn">Sorted On</label>         <input type="text" class="form-control" name="sortOn" [(ngModel)]="sortOn">      </div>        <div class="form-group">          <button type="button" (click)="getRepos()">Get Repos</button>      </div>        <div \*ngIf="loading">loading...</div>        <div \*ngIf="errorMessage" class="alert alert-warning">         <strong>Warning!</strong> {{errorMessage}}      </div>        <div class='table-responsive'>              <table class='table'>                  <thead>                      <tr>                          <th>ID</th>                          <th>Name</th>                          <th>HTML Url</th>                          <th>description</th>                      </tr>                  </thead>                  <tbody>                      <tr \*ngFor="let repo of repos;">                          <td>{{repo.id}}</td>                          <td>{{repo.name}}</td>                          <td>{{repo.html\_url}}</td>                          <td>{{repo.description}}</td>                      </tr>                  </tbody>              </table>          </div>            <pre>{{repos | json}}</pre>    </div> |

Summary

We learned how to pass Get Parameters or request parameters when we invoke the HTTP get Request using httpClient.get method

HttpHeaders

We add HTTP Headers using the [HttpHeaders](https://angular.io/api/common/http/HttpHeaders)helper class. It is passed as one of the arguments to the GET, POST, PUT, DELETE, PATCH & OPTIONS request.

To use HttpHeaders in your app, you must import it into your component or service

|  |  |
| --- | --- |
| 1  2  3 | import { HttpHeaders } from '@angular/common/http'; |

Then create an instance of the class

|  |  |
| --- | --- |
| 1  2  3  4  5 | const headers= new HttpHeaders()    .set('content-type', 'application/json')    .set('Access-Control-Allow-Origin', '\*'); |

And then call the httpClient.get method passing the headers as the argument

|  |  |
| --- | --- |
| 1  2  3 | return this.httpClient.get(this.baseURL + 'users/' + userName + '/repos', { 'headers': headers }) |

Note that httpHeaders are immutable. i.e every method on HttpHeaders object does not modify it but returns a new HttpHeaders object.

The httpHeaders class has several methods using which you can manipulate the headers.

set

set(name: string, value: string | string[]): HttpHeaders

The Sets method returns a new instance after modifying the given header. If the header already exists, its value is replaced with the given value in the returned object.

|  |  |
| --- | --- |
| 1  2  3  4  5 | const headers = new HttpHeaders()     .set('content-type', 'application/json')     .set('Access-Control-Allow-Origin', '\*'); |

httpHeaders are immutable

The HTTP headers are immutable. The following example does not work as each set method returns a new header and does not update the original header.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | let headers = new HttpHeaders()  headers .set('content-type', 'application/json')  headers .set('Access-Control-Allow-Origin', '\*')  console.log(headers); |

To workaround, you can use the code as follows

|  |  |
| --- | --- |
| 1  2  3  4  5 | const headers= new HttpHeaders()    .set('content-type', 'application/json')    .set('Access-Control-Allow-Origin', '\*'); |

You can also use the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | let headers = new HttpHeaders()  headers=headers.set('content-type','application/json')  headers=headers.set('Access-Control-Allow-Origin', '\*');  console.log(headers) |

append

append(name: string, value: string | string[]): HttpHeaders

The append method appends a new value to the existing set of values for a header and returns a new instance. The append method does not check if the value exists.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | let headers = new HttpHeaders()    headers=headers.append('content-type','application/json')  headers=headers.append('Access-Control-Allow-Origin', '\*')  headers=headers.append('content-type','application/x-www-form-urlencoded')    console.log(headers) |

The above results in content-type header in the request header as content-type: application/json,application/x-www-form-urlencoded

has

has(name: string): boolean

Returns true if the given header with the name already exists in the HttpHeaders. The following code checks if the content-type header present in the request header. If not it adds it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | let headers = new HttpHeaders()      headers=headers.append('Access-Control-Allow-Origin', '\*')      if (!headers.has('content-type')) {        headers=headers.append('content-type','application/json')      } |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

get

get(name: string): string | null

Get the first value for the given header name, or null if it’s not present.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | let headers = new HttpHeaders()     .set('content-type','application/json')     .set('Access-Control-Allow-Origin', '\*')    const h =headers.get('content-type')    if (h==null) {    console.log('content type header not present')  } else {    console.log(h)      //returns 'application/json'  } |

getAll

getAll(name: string): string[] | null

Get all the headers for the given header name, or null if it’s not present.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | let headers = new HttpHeaders()     .set('content-type','application/json')     .set('Access-Control-Allow-Origin', '\*')     .append('content-type','application/x-www-form-urlencoded')    const h =headers.getAll('content-type')  console.log(h)    \*\*\* output  0: "application/json"  1: "application/x-www-form-urlencoded" |

Keys

keys(): string[]

Get all the headers for this request.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | let headers = new HttpHeaders()      .set('content-type','application/json')      .set('Access-Control-Allow-Origin', '\*')      .append('content-type','application/x-www-form-urlencoded')    const h =headers.keys()  console.log(h)    \*\*\*output  0: "content-type"  1: "Access-Control-Allow-Origin" |

delete

delete(name: string, value?: string | string[]): HttpHeaders

Deletes the header and returns the new headers.  You can delete using the header name or by using the name & value.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | let headers = new HttpHeaders()      .set('content-type','application/json')      .set('Access-Control-Allow-Origin', '\*')      .append('content-type','application/x-www-form-urlencoded')    headers=headers.delete("content-type","application/json")  //delete content-type='application/json'    headers=headers.delete("content-type")   //delete all content-type headers |

HttpHeaders from object

The following code shows how you can create HttpHeaders from an object.

|  |  |
| --- | --- |
| 1  2  3  4 | let headers = new HttpHeaders({ 'Access-Control-Allow-Origin': '\*','content-type': 'application/json'}  )  console.log(headers) |

Using HTTP Interceptor

Most headers we add to the HTTP Request in the entire application are likely to remain the same. Adding them to every GET, POST, PUT, etc requests are cumbersome. Instead, you can make use of the [HTTP Interceptors](https://www.tektutorialshub.com/angular/angular-httpclient-http-interceptor/) to intercept every request and add the commonly used headers. Refer to our tutorial on [how to set HttpHeaders using HTTP Interceptors](https://www.tektutorialshub.com/angular/angular-httpclient-http-interceptor/#setting-the-new-headers)

HttpHeaders Example

Refer to our tutorial on [HTTP Post example](https://www.tektutorialshub.com/angular/angular-http-post-example/).

The code requires you to set up a fake backend server using json-server. Install JSON server using the following command.

|  |  |
| --- | --- |
| 1  2  3 | npm install -g json-server |

Create a db.json file with some data.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | {    "people": [      {        "id": 1,        "name": "Don Bradman"      },      {        "id": 2,        "name": "Sachin Tendulkar"      }    ]  } |

Start the server with the following command. The server will run on the port http://localhost:3000/

|  |  |
| --- | --- |
| 1  2  3 | json-server --watch db.json |

Code

person.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | export class Person {    id:number    name:string  } |

app.module.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';    import { HttpClientModule } from '@angular/common/http';  import { FormsModule } from '@angular/forms'    import { AppRoutingModule } from './app-routing.module';  import { AppComponent } from './app.component';          @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule,      AppRoutingModule,      HttpClientModule,      FormsModule,    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

app.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | import { Component, OnInit } from '@angular/core';  import { ApiService } from './api.service';  import { Person } from './person';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {      title = 'http Headers Example';    people:Person[];    person = new Person();      constructor(private apiService:ApiService) {}      ngOnInit() {      this.refreshPeople()    }      refreshPeople() {      this.apiService.getPeopleFromObject()        .subscribe(data => {          this.people=data;        })      }      addPerson() {      this.apiService.addPerson(this.person)        .subscribe(data => {            this.person = new Person();          this.refreshPeople();        })      }    } |

app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | <h1>{{title}}</h1>    <div>    <div>      <label>Name: </label>      <input [(ngModel)]="person.name" />    </div>    <div>      <button (click)="addPerson()">Add</button>    </div>  </div>    <button (click)="refreshPeople()">Refresh</button>    <table class='table'>    <thead>      <tr>        <th>ID</th>        <th>Name</th>      </tr>    </thead>    <tbody>      <tr \*ngFor="let person of people;">        <td>{{person.id}}</td>        <td>{{person.name}}</td>      </tr>    </tbody>  </table> |

app.routing.module.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import { NgModule } from '@angular/core';  import { Routes, RouterModule } from '@angular/router';      const routes: Routes = [];    @NgModule({    imports: [RouterModule.forRoot(routes)],    exports: [RouterModule]  })  export class AppRoutingModule { } |

app.service.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159 | import { HttpClient, HttpHeaders } from '@angular/common/http';  import { Person } from './person';  import { Injectable } from '@angular/core';  import { Observable } from 'rxjs';    @Injectable({ providedIn: 'root' })  export class ApiService {      baseURL: string = "http://localhost:3000/";      constructor(private http: HttpClient) {    }        getPeople(): Observable<Person[]> {      console.log('getPeople ' + this.baseURL + 'people')      return this.http.get<Person[]>(this.baseURL + 'people')    }      //Adding headers    getPeopleWithHeaders(): Observable<Person[]> {      const headers = { 'content-type': 'application/json'}      console.log(headers)      return this.http.get<Person[]>(this.baseURL + 'people',{'headers':headers})    }      //Set method    getPeopleWithSet(): Observable<Person[]> {      const headers = new HttpHeaders()        .set('content-type','application/json')        .set('Access-Control-Allow-Origin', '\*');      console.log(headers)      return this.http.get<Person[]>(this.baseURL + 'people',{'headers':headers})    }      //This wont work    getPeopleWithImmutable(): Observable<Person[]> {      const headers = new HttpHeaders()      headers.set('content-type','application/json')      headers.set('Access-Control-Allow-Origin', '\*');        console.log(headers)      return this.http.get<Person[]>(this.baseURL + 'people',{'headers':headers})    }      getPeopleWithImmutable1(): Observable<Person[]> {      let headers = new HttpHeaders()      headers=headers.set('content-type','application/json')      headers=headers.set('Access-Control-Allow-Origin', '\*');        console.log(headers)      return this.http.get<Person[]>(this.baseURL + 'people',{'headers':headers})    }        getPeopleAppend(): Observable<Person[]> {      let headers = new HttpHeaders()      headers=headers.append('content-type','application/json')      headers=headers.append('Access-Control-Allow-Origin', '\*')      headers=headers.append('content-type','application/x-www-form-urlencoded')      headers=headers.append('customer-header', 'custom')      console.log(headers)      return this.http.get<Person[]>(this.baseURL + 'people',{'headers':headers})    }      getPeopleHas(): Observable<Person[]> {      let headers = new HttpHeaders()      //headers=headers.append('content-type','application/json')      headers=headers.append('Access-Control-Allow-Origin', '\*')      if (!headers.has('content-type')) {        headers=headers.append('content-type','application/json')      }        console.log(headers)      return this.http.get<Person[]>(this.baseURL + 'people',{'headers':headers})    }        getPeopleGet(): Observable<Person[]> {      let headers = new HttpHeaders()        .set('content-type','application/json')        .set('Access-Control-Allow-Origin', '\*')        const h =headers.get('content-type')      if (h==null) {        console.log('content type header not present')      } else {        console.log(h)      }          return this.http.get<Person[]>(this.baseURL + 'people',{'headers':headers})    }      getPeopleGetAll(): Observable<Person[]> {      let headers = new HttpHeaders()        .set('content-type','application/json')        .set('Access-Control-Allow-Origin', '\*')        .append('content-type','application/x-www-form-urlencoded')        const h =headers.getAll('content-type')      console.log(h)          return this.http.get<Person[]>(this.baseURL + 'people',{'headers':headers})    }      getPeopleKeys(): Observable<Person[]> {      let headers = new HttpHeaders()        .set('content-type','application/json')        .set('Access-Control-Allow-Origin', '\*')        .append('content-type','application/x-www-form-urlencoded')        const h =headers.keys()      console.log(h)          return this.http.get<Person[]>(this.baseURL + 'people',{'headers':headers})    }      getPeopleDelete(): Observable<Person[]> {      let headers = new HttpHeaders()        .set('content-type','application/json')        .set('Access-Control-Allow-Origin', '\*')        .append('content-type','application/x-www-form-urlencoded')          headers=headers.delete('content-type','application/json')        //headers=headers.delete("content-type")          console.log(headers)          return this.http.get<Person[]>(this.baseURL + 'people',{'headers':headers})    }        getPeopleFromObject(): Observable<Person[]> {        let headers = new HttpHeaders({ 'Access-Control-Allow-Origin': '\*','content-type': 'application/json'}  )        console.log(headers)          return this.http.get<Person[]>(this.baseURL + 'people',{'headers':headers})    }        addPerson(person:Person): Observable<Person> {      const headers = { 'content-type': 'application/json'}      const body=JSON.stringify(person);      console.log(body)      return this.http.post<Person>(this.baseURL + 'people', body,{'headers':headers})    }  } |

Summary

We learned how to add/modify the HTTP Headers using the HttpHeaders in Angular.

What is angular Http interceptor

The Angular HTTP interceptors sit between our application and the backend. When the application makes a request, the interceptor catches the request before it is sent to the backend. By Intercepting requests, we will get access to request headers and the body. This enables us to transform the request before sending it to the Server.

When the response arrives from the back end the Interceptors can transform it before passing it to our application.

One of the main benefits of the Http Interceptors is to add the Authorization Header to every request. We could do this manually, but that is a lot of work and error-prone. Another benefit is to catch the errors generated by the request and log them.

How to Create Http Interceptor

To Implement the Interceptor, you need to create an [injectable service](https://www.tektutorialshub.com/angular/angular-dependency-injection/), which implements the HttpInterceptorinterface.

|  |  |
| --- | --- |
| 1  2  3 | @Injectable() export class AppHttpInterceptor implements HttpInterceptor { |

This class must implement the method Intercept.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {      //do whatever you want with the HttpRequest      return next.handle(req);  } |

This class is then provided in the Root Module using the HTTP\_INTERCEPTORS token:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | providers: [      {          provide: HTTP\_INTERCEPTORS,          useClass: AppHttpInterceptor,          multi: true      }  ], |

Ad by Valueimpression

HttpInterceptor Interface

At the heart of the Interceptor, logic is the [HttpInterceptor Interface](https://angular.io/api/common/http/HttpInterceptor). we must Implement it in our Interceptor Service.

The interface contains a single method Intercept with the following signature

|  |  |
| --- | --- |
| 1  2  3 | intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> |

You can define more than one Interceptor. The Interceptors are called in the order they are defined in [provider metadata](https://www.tektutorialshub.com/angular/angular-providers/).

HttpRequest

The first argument is [HttpRequest](https://angular.io/api/common/http/HttpRequest).

The HttpRequest is an outgoing HTTP request which is being intercepted. It contains URL, method, headers, body, and other request configuration.

The HttpRequest is a immutable class. Which means that we can’t modify the original request. To make changes we need to clone the Original request using the HttpRequest.clone method

HttpHandler

The second argument is [httpHandler](https://angular.io/api/common/http/HttpHandler)

The HttpHandler dispatches the HttpRequest to the next Handler using the method HttpHandler.handle. The next handler could be another Interceptor in the chain or the Http Backend.

Http Interceptor Example

Open the GitHubService app, which we created in the previous tutorial. You can download it from [GitHub](https://github.com/tekTutorialsHub/HTTP). The Final code is in the folder **HttpInterceptors**. The initial code in **HttpGetParameters** folder.

Create the Interceptor

Create AppHttpInterceptor.ts under the src/app folder and copy the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import {Injectable} from "@angular/core";  import {HttpEvent, HttpHandler, HttpInterceptor,HttpRequest} from "@angular/common/http";  import {Observable} from "rxjs/Observable";    @Injectable()  export class AppHttpInterceptor implements HttpInterceptor {      constructor() {      }        intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {          console.log(req);          return next.handle(req);      }  } |

Now let us look at each code in detail

First, we have Imported the following module.

|  |  |
| --- | --- |
| 1  2  3  4  5 | import {Injectable} from "@angular/core";  import {HttpEvent, HttpHandler, HttpInterceptor,HttpRequest} from "@angular/common/http";  import {Observable} from "rxjs/Observable"; |

Create a class AppHttpInterceptor which implements HttpInterceptor Interface.

|  |  |
| --- | --- |
| 1  2  3 | export class AppHttpInterceptor implements HttpInterceptor { |

Then create an Intercept method that takes HttpRequest and HttpHandler as the argument.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {      //Do whatever you want to do with the Request      console.log(req);      return next.handle(req);  } |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

In the method body, you can modify the HttpRequest object. Once done, you can call the HttpHandler.handle method of the HttpHandler with the HttpRequest object. The HttpHandler.handle method invokes the next interceptor or sends the request to the backend server.

**App.Module**

The Complete code from App Module.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { HttpClientModule,HTTP\_INTERCEPTORS} from '@angular/common/http';  import { FormsModule } from '@angular/forms';    import { AppComponent } from './app.component';    import { GitHubService } from './github.service';  import {AppHttpInterceptor} from './AppHttpInterceptor';    @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule,      HttpClientModule,      FormsModule    ],    providers: [GitHubService,      {      provide: HTTP\_INTERCEPTORS,      useClass: AppHttpInterceptor,      multi: true    }  ],    bootstrap: [AppComponent]  })  export class AppModule { } |

First, we need to import the HttpClientModule & HTTP\_INTERCEPTORS from @angular/common/http.

|  |  |
| --- | --- |
| 1  2  3 | import { HttpClientModule,HTTP\_INTERCEPTORS} from '@angular/common/http'; |

Next, register AppHttpInterceptor as the [Provider](https://www.tektutorialshub.com/angular/angular-providers/) for the HTTP\_INTERCEPTORS.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | providers: [GitHubService,      {      provide: HTTP\_INTERCEPTORS,      useClass: AppHttpInterceptor,      multi: true    } |

Run the Application. Open the developer console and see the output of console.log(req).

Setting the new headers

We are able to Intercept the request and log it to the console in the above example. Now we will modify the HTTP Headers and Custom Headers.

Adding the Content-Type

To Modify the request we need to clone it. The HttpRequest.clone method allows us to modify the specific properties of the request while copying others. In the following example we are adding the new header content-type to the request.

|  |  |
| --- | --- |
| 1  2  3 | req = req.clone({ headers: req.headers.set('Content-Type', 'application/json') }); |

The headers object is also immutable. Hence we need to clone it using the headers.set method. The header.set method clones the current header and adds/modifies the new header value and returns the cloned header.

You can also use the headers.append method as shown below. Note that the append method always appends the header even if the value is already present.

|  |  |
| --- | --- |
| 1  2  3 | req = req.clone({ headers: req.headers.append('Content-Type', 'application/json') }); |

You can also make use of the setHeaders shortcut as shown below

req = req.clone( {setHeaders: {‘Content-Type’: ‘application/json’}} );

You may want to check if the header already exists using headers.has() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | if (!req.headers.has('Content-Type')) {      req = req.clone({ headers: req.headers.set('Content-Type', 'application/json') });  }  ﻿ |

Check the current value of the header.

|  |  |
| --- | --- |
| 1  2  3 | req.headers.get('Accept') |

And remove a header.

|  |  |
| --- | --- |
| 1  2  3 | req = req.clone({ headers: req.headers.delete('Content-Type','application/json') }); |

Adding the Authorisation token

Add authorization token.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | const token: string =authService.Token; //Get token from some service  if (token) {      req = req.clone({ headers: req.headers.set('Authorization', 'Bearer ' + token) });  } |

Intercepting the Response

The response of the back-end server can be intercepted using the various Rxjs Operators. The [map](https://www.learnrxjs.io/learn-rxjs/operators/transformation/map) can be used to modify the response before sending it to the application. The [do](https://www.learnrxjs.io/learn-rxjs/operators/utility/do) operator is useful for logging the events or time requests. The [catch](https://www.learnrxjs.io/learn-rxjs/operators/error_handling/catch) operator can be used to catch the error. The [retry](https://www.learnrxjs.io/learn-rxjs/operators/error_handling/retry) operator can be used to retry the failed operation.

Logging

The following example code shows the use of do operator. The do operator is invoked whenever certain events take place on an Observable.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {            req = req.clone({ headers: req.headers.append('Content-Type', 'application/json')});          const started = Date.now();            return next.handle(req)          .do(event => {              console.log(event);              const elapsed = Date.now() - started;              console.log(`Request for ${req.urlWithParams} took ${elapsed} ms.`);              if (event instanceof HttpResponse) {                  console.log(`Response Received`);              };          });      } |

In the above example, do is invoked twice. First time when the request is sent to the server (event={type: 0}). The second time when the response is received (event instanceof HttpResponse).

Modify Response

The following code shows the use of the map operator, which allows us to transform the response. The response can be modified using the method clone (the response object is immutable). Then return the cloned response. The example below replaces the entire response body with the new body and returns the response.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {            return next.handle(req)              .map(resp => {                    const myBody = [{ 'id': '1',                                    'name': 'TekTutorialsHub',                                    'html\_url': 'www.tektutorialshub.com',                                    'description': 'description'                                  }];                    // on Response                  if (resp instanceof HttpResponse) {                      console.log(resp);                      console.log(resp.body);                      resp = resp.clone<any>({ body: myBody});                      return resp;                  }              });      } |

Catching the Error

The errors can be caught with the catch operator. The catch callback gets the HttpErrorResponse as its argument, which represents an error object. It contains information about headers, status, statusText  & URL, etc.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {            const token: string = 'invald token';          req = req.clone({ headers: req.headers.set('Authorization', 'Bearer ' + token) });            return next.handle(req)              .map(resp => {                  // on Response                  if (resp instanceof HttpResponse) {                      // Do whatever you want with the response.                      return resp;                  }              }).catch(err => {                  // onError                  console.log(err);                  if (err instanceof HttpErrorResponse) {                      console.log(err.status);                      console.log(err.statusText);                      if (err.status === 401) {                          // redirect the user to login page                          // 401 unauthorised user                      }                  }                  return Observable.of(err);              });      } |

Cancel the current Request

We can also cancel the current request by returning the EMPTY observable.

The following code snippet checks if the user is logged in. If not then it will not send the request to server.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import { EMPTY } from 'rxjs';    intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {    if (NotLoggedIn) {      return EMPTY;    }      return next.handle(request);  } |

Change the Requested URL

You can change the requested URL before it sent to the server. The HttpRequest contains the url property, which you can change before sending the request.

This is useful when you want to add the base URL of all the requests, change HTTP to HTTPS etc.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | const baseURL="https://www.tektutorialsHub.com/";    intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {       const newReq = req.clone({        url: baseURL + req.url;      });       return next.handle(httpsReq);  } |

References

[HTTP Interceptors](https://angular.io/api/common/http/HttpInterceptor)

**Read More**

[**Angular HTTP Client Tutorial**](https://www.tektutorialshub.com/angular/angular-httpclient/)

[**HTTP GET Example**](https://www.tektutorialshub.com/angular/angular-http-get-example-using-httpclient/)

[**HTTP POST Example**](https://www.tektutorialshub.com/angular/angular-http-post-example/)

[**Passing URL Parameters (Query strings)**](https://www.tektutorialshub.com/angular/angular-pass-url-parameters-query-strings/)

[**HTTP Headers Example**](https://www.tektutorialshub.com/angular/angular-httpheaders/)

Summary

We learned how to intercept HTTP request & response using the new [HttpClientModule.](https://www.tektutorialshub.com/angular/angular-httpclient/)  The Interceptor can be useful for adding custom headers to the outgoing request, logging the incoming response, etc.

# Angular Router

What is Routing

Routing allows you to move from one part of the application to another part or one View to another View.

The Angular Router Module

The Router is a separate module in Angular. It is in its own library package, @angular/router. The Router Module provides the necessary service providers and directives for navigating through application views.

Using Angular Router you can

Navigate to a specific view by typing a URL in the address bar

Pass optional parameters to the View

Bind the clickable elements to the View and load the view when the user performs application tasks

Handles back and forward buttons of the browser

Allows you to dynamically load the view

Protect the routes from unauthorized users using Guards

Ad by Valueimpression

Components of Angular Router Module

Router

The Angular Router is an object that enables navigation from one component to the next component as users perform application tasks like clicking on menus links, buttons or clicking on back/forward button on the browser. We can access the router object and use its methods like navigate() or navigateByUrl(), to navigate to a route

Route

Route tells the Angular Router which view to display when a user clicks a link or pastes a URL into the browser address bar. Every Route consists of a path and a component it is mapped to. The Router object parses and builds the final URL using the Route

Routes

Routes is an array of Route objects our application supports

RouterOutlet

The outerOutlet is a directive (<router-outlet>) that serves as a placeholder, where the Router should display the view

RouterLink

The RouterLink is a directive that binds the HTML element to a Route. Clicking on the HTML element, which is bound to a RouterLink, will result in navigation to the Route. The RouterLink may contain parameters to be passed to the route’s component.

RouterLinkActive

RouterLinkActive is a directive for adding or removing classes from an HTML element that is bound to a RouterLink. Using this directive, we can toggle CSS classes for active RouterLinks based on the current RouterState

ActivatedRoute

The ActivatedRoute is an object that represents the currently activated route associated with the loaded Component.

RouterState

The current state of the router including a tree of the currently activated routes together with convenience methods for traversing the route tree.

RouteLink Parameters array

The Parameters or arguments to the Route. It is an array which you can bind to RouterLink directive or pass it as an argument to the Router.navigate method.

How to configure Angular Router

To Configure the Router in Angular, you need to follow these steps

Set the <base href>

Define routes for the view

Register the Router Service with Routes

Map HTML Element actions to Route

Choose where you want to display the view

Set the <base href>

The HTML <base> element specifies the base URL to use for all relative URLs contained within a document.

The Angular Router uses the HTML5 style of Routing (or PathLocationStrategy) as the default option. The router makes use of the browser’s history API for navigation and URL interaction.

|  |  |
| --- | --- |
| 1  2  3 | <base href="/"> |

To make HTML5 routing to work, we need to set up the “**base href”** in the DOM. This is done in app’s index.html file immediately after the head tag.

Define the routes

Next, create an array of route objects. Each route maps path (URL Segment) to the component

|  |  |
| --- | --- |
| 1  2  3 | const appRoutes={ path: 'product', component: ProductComponent } |

Where

**path:** The URL path segment of the route. We will use this value to refer to this route elsewhere in the app

**component:** The component to be loaded.

This route tells angular to render ProductComponent when the user navigate to the URL “/product”

Register the Routes

Import the Router Module from @angular/router library in the root module of the application

|  |  |
| --- | --- |
| 1  2  3 | import { RouterModule } from '@angular/router'; |

Then, install the routes using the RouterModule.forRoot method, passing the routes as the argument in the imports array

|  |  |
| --- | --- |
| 1  2  3 | imports: [RouterModule.forRoot(routes)], |

Map Action to Routes

Next, we need to bind the click event of the link, image or button to a route. This is done using the routerlink directive

|  |  |
| --- | --- |
| 1  2  3 | <li><a [routerLink]="['product']">Product</a></li> |

The routerLink directive accepts an array of route names along with parameters. This array is called as **Link Parameters array**.

When the application requests navigation to the route “product”, the router looks in the routes array and activates the instance of the component associated with the route “product”, which is ProductComponent. The browser address location & history is also updated to /product

Choose where you want to display

Finally, we need to tell the angular where to display the view. This is done using the RouterOutlet directive as shown. We will add the following directive to the root component.

|  |  |
| --- | --- |
| 1  2  3 | <router-outlet></router-outlet> |

Angular Router: Sample Application

Let’s build a sample application with four components and build a navigation system to route for each one of them

**HomeComponent:** This component will display the Welcome message. This is also our default component.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import {Component} from '@angular/core';    @Component({      template: `<h1>Welcome!</h1>                <p>This is Home Component </p>               `  })    export class HomeComponent {  } |

**ContactComponent:** Displays the contact message.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import {Component} from '@angular/core';    @Component({       template: `<h1>Contact Us</h1>                  <p>TekTutorialsHub </p>                  `  })  export class ContactComponent {  } |

**ProductComponent:** Displays the list of products. The Products are retrieved from the Angular 2 Service using Dependency injection.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | import { Component, OnInit } from '@angular/core';    import { ProductService } from './product.service';  import { Product } from './product';    @Component({    templateUrl: './product.component.html',  })    export class ProductComponent  {       products:Product[];       constructor(private productService:ProductService){     }       ngOnInit() {       this.products=this.productService.getProducts();     }    } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | <h1>Product List</h1>  <div class='table-responsive'>      <table class='table'>          <thead>              <tr>                  <th>ID</th>                  <th>Name</th>                  <th>Price</th>              </tr>          </thead>          <tbody>              <tr \*ngFor="let product of products;">                  <td>{{product.productID}}</td>                  <td><a [routerLink]="['detail',product.productID]">{{product.name}} </a> </td>                  <td>{{product.price}}</td>              </tr>          </tbody>      </table>  </div>    <router-outlet></router-outlet> |

**ErrorComponent:** The ErrorComponent is displayed, when the user navigates to a nonexistent path. This is basically a 404 error page.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import {Component} from '@angular/core';    @Component({      template: `<h1>Page not found</h1>                 <p>This is a Error Page</p>                `  })    export class ErrorComponent {  } |

**Product Service**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | import { Observable } from 'rxjs/Observable';  import {Product} from './Product'      export class ProductService{        public getProducts() {            let products:Product[];            products=[              new Product(1,'Memory Card',500),              new Product(2,'Pen Drive',750),              new Product(3,'Power Bank',100)          ]            return products;      }          public getProduct(id) {          let products:Product[]=this.getProducts();          return products.find(p => p.productID==id);      }      } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | export class Product {        constructor(productID:number,    name: string ,   price:number) {          this.productID=productID;          this.name=name;          this.price=price;      }        productID:number ;      name: string ;      price:number;    } |

Index.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | <!doctype html>  <html>  <head>    <base href="/">    <meta charset="utf-8">    <title>Angular 2 Routing</title>        <meta name="viewport" content="width=device-width, initial-scale=1">    <link rel="icon" type="image/x-icon" href="favicon.ico">    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" rel="stylesheet">      ...  </head>  <body>    <app-root>Loading...</app-root>  </body>  </html> |

Note the **<base href=”/”>** right after the head tag. This makes the browser know where is the root of our application is and helps it to construct the URL’s

Routes

Now, we have our components ready. The next step is to create our routes.

It is good practice to create all our route configuration in a separate file. So create **app.routes.ts** under the app folder.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import { Routes } from '@angular/router';    import { HomeComponent} from './home.component'  import { ContactComponent} from './contact.component'  import { ProductComponent} from './product.component'  import { ErrorComponent} from './error.component'    export const appRoutes: Routes = [    { path: 'home', component: HomeComponent },    { path: 'contact', component: ContactComponent },    { path: 'product', component: ProductComponent },    { path: '', redirectTo: 'home', pathMatch: 'full' },    { path: '\*\*', component: ErrorComponent }  ]; |

First, we import Routes from router module

Next, we need to import all the components, that requires routing We have imported Home, Contact, Product, and Error Components

Finally, we have defined a constant (appRoutes) that contains the **Routes** that we wish to create. The **Routes** is an array of **route configuration object** (or route object).

Each route has several configurable properties.

Our First route is

|  |  |
| --- | --- |
| 1  2  3 | { path: 'home', component: HomeComponent }, |

The first parameter is the path, which represents the URL path segment. The second parameter is the component to display. The above route configuration means, when you navigate to /home (URL path segment), then the HomeComponent gets displayed.

Note that path does not contain the leading slash

The next two routes are similar to the home route

|  |  |
| --- | --- |
| 1  2  3  4 | { path: 'contact', component: ContactComponent },  { path: 'product', component: ProductComponent }, |

Default Route

The fourth route is

|  |  |
| --- | --- |
| 1  2  3 | { path: '', redirectTo: 'home', pathMatch: 'full' }, |

The path is empty, indicates the default route. The default route is redirected to the home path using the RedirectTo argument. This route means that, when you navigate to the root of your application /, you are redirected to the home path (/home), which in turn displays the HomeComponent.

Note, that we have pathMatch argument set to ‘full’. The pathMatch tells the Router how to match the URL.

When it is set to full, the path is matched to the entire URL

Every route ends in an empty space for ex: /contact/’’. If pathMatch is not set to full then the router will apply the redirect, which results in the error.

Wild Card Route

The next route is wildcard route

|  |  |
| --- | --- |
| 1  2  3 | { path: '\*\*', component: ErrorComponent } |

The “\*\*” matches every URL. The Router will display the ErrorComponent.

Order matters: First one wins

Note that the order of the route is important. The Routes are matched in the order they are defined. The Router always returns the first matching route (first-match-wins strategy)

Since the wildcard route (\*\*) matches every URL and should be placed last.

Now, we have set up our routes. Now we will add these routes to our application.

Register the Routes

Routes are registered in root module of the application. I.e. app.module.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule } from '@angular/forms';  import { HttpModule } from '@angular/http';    import { RouterModule } from '@angular/router';    import { AppComponent } from './app.component';  import { HomeComponent} from './home.component'  import { ContactComponent} from './contact.component'  import { ProductComponent} from './product.component'  import { ErrorComponent} from './error.component'    import { ProductService } from './product.service';    import { appRoutes } from './app.routes';    @NgModule({    declarations: [      AppComponent,HomeComponent,ContactComponent,ProductComponent,ErrorComponent    ],    imports: [      BrowserModule,      FormsModule,      HttpModule,      RouterModule.forRoot(appRoutes)                           /\*path location strategy \*/      /\*RouterModule.forRoot(appRoutes, { useHash: true }) \*/   /\*Hashlocationstrategy \*/    ],    providers: [ProductService],    bootstrap: [AppComponent]  })  export class AppModule { } |

First, we import the RouterModule

|  |  |
| --- | --- |
| 1  2  3 | import { RouterModule } from '@angular/router'; |

Next, import all the components

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | import { AppComponent } from './app.component';  import { HomeComponent} from './home.component'  import { ContactComponent} from './contact.component'  import { ProductComponent} from './product.component'  import { ErrorComponent} from './error.component' |

Next import the routes, which we configured from app.routes

|  |  |
| --- | --- |
| 1  2  3 | import { routes } from './app.routes'; |

Finally, we add the RouterModule to the import array, passing the routes we have configured via the forRoot method

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | imports: [      BrowserModule,      FormsModule,      HttpModule,      RouterModule.forRoot(routes)  ], |

Note that we are using the **forRoot method**.

**the forRoot method** is used, when you want to provide the service and also want to configure the service at the same time

The **routermodule.forroot** method returns the Router Service configured with the routes passed in the argument and also registers the Router service. It also registers the other providers that the routing module requires.

When the application is bootstrapped, the Router service looks at the current browser URL and performs the initial navigation.

When the user changes the URL either clicking on a link in the page or by entering a URL in the address bar, the router looks for a corresponding Route from the Routes array and renders the associated component.

Defining The Navigation

The next step is to define the navigation

Open the app.component.html. The AppComponent is only handling navigation. It will display the menu option, which user can click to navigate to a view

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | <div class="container">    <nav class="navbar navbar-default">    <div class="container-fluid">      <div class="navbar-header">        <a class="navbar-brand" [routerLink]="['/']"><strong> {{title}} </strong></a>      </div>      <ul class="nav navbar-nav">          <li><a [routerLink]="['home']">Home</a></li>          <li><a [routerLink]="['product']">Product</a></li>          <li><a [routerLink]="['contact']">Contact us</a></li>      </ul>    </div>  </nav>    <router-outlet></router-outlet>    </div> |

We are using bootstrap to style our component

|  |  |
| --- | --- |
| 1  2  3  4  5 | <li><a [routerLink]="['home']">Home</a></li>  <li><a [routerLink]="['product']">Product</a></li>  <li><a [routerLink]="['contact']">Contact us</a></li> |

We use **the routerLink directive** to bind anchor tag elements to the route

RouterLink is an [attribute](https://www.tektutorialshub.com/angular/angular-directives/)directive. We enclose it a square bracket. The routerLink is then bound to template expression, which returns a**link parameters array**.

The **Link Parameters array** is the Parameters or arguments to the Route. The Angular Router module constructs the URL using the link parameters array

When the user clicks on the link, the Router service uses the path to locating the route associated with the path and activates the component

Display the component using Router-outlet

Finally, we need to tell Angular where to display the Component. This is done using the **Router-outlet directive**

The RouterOutlet is a directive that tells the Angular where in our page we want to display the view.  
We do not have to import the RouterOutlet and RouterLink directives. These directives are imported when we imported RouterModule in our app.module

|  |  |
| --- | --- |
| 1  2  3 | <router-outlet></router-outlet> |

That’s it

Running the Application

Type the in the address bar http://localhost:4200, you should see the HomeComponent is rendered, which is the default root

Type the invalid URL and you should see the ErrorComponent rendered.

Click on the menu options or Type the Back & forward button in the browser. Everything should work as intended.

Location Strategies in Angular Router

Being a Single Page Application, the Angular applications should not send the URL to the server and should not reload the page, every time user requests for a new page.

The URLs are strictly local in Angular Apps. The [Angular router](https://www.tektutorialshub.com/angular/angular-routing-navigation/) navigates to the new component and renders its template and updates the history and URL for the view. All this happens locally in the browser.

There are two ways, by which Angular achieves this. These are called Location Strategies.

The Location Strategy defines how our URL/Request is resolved. It also determines how your URL will look like

Angular supports two Location Strategies:

**HashLocationStrategy**  
Where URL looks like http://localhost:4200/#/product

**PathLocationStrategy**  
Where URL looks like http://localhost:4200/product

Before going further lets first understand what is client-side routing is

Client-Side Routing

In a Multi-page web application, Every time the application needs to a display a page it has to send a request to the webserver. You can do that by either typing the URL in the address bar, clicking on the Menu link/ button. Every such action results in browser sending a new request to the Web server

But, the Angular Applications are single-page applications or SPA.

All the components are displayed on a single page

In a Typical Single Page Application, when the Web application is loaded it loads the single HTML page. Whenever the user interacts with the page, only a part of the page is dynamically updated.

If you open the index.html in any of the angular application, you would see the following HTML markup

|  |  |
| --- | --- |
| 1  2  3  4  5 | <body>      <app-root>Loading...</app-root>  </body> |

The "app-root" is a placeholder (selector), which is defined in the root component.

Angular generates and loads the view associated with the root component inside the "app-root". Any subsequent components are also loaded dynamically inside the "app-root" selector

Angular does all this behind the scenes.

In such a scenario, we are not required to change the URL. But that brings a few cons

You won’t be able to refresh the page

You won’t be able to go to a particular view by typing the URL

Sharing the URL with someone is not possible

The Back button will not work as you cannot go back to the previous page

SEO is not possible

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

That is where the client-side routing comes into the picture

The Client-side routing simply mimics server-side routing by running the process in the browser. It changes the URL in the browser address bar and updates the browser history, without actually sending the request to the server

How Client-Side Routing works

The Client-side routing is handled in two ways

Hash style Routing

HTML 5 Routing

Hash Style Routing

The Hash style routing using the anchor tags technique to achieve client-side routing.

The anchor tags, when used along with the # allows us to jump to a place, within the web page.

For Example

**Index.html**

|  |  |
| --- | --- |
| 1  2  3 | <a name="contact">Contact Us</a> |

And we visited the URL http://mysite.com/index.html#contact, the browser would scroll to the location of the Contact us label

When the requested anchor tag is on the current page, then the browser does not send the request to the Web server.

The Hashstyle Routing uses this technique to create the URL

The URL would look like something like

|  |  |
| --- | --- |
| 1  2  3  4  5 | http://www.example.com  http://www.example.com/#/about  http://www.example.com/#/contact |

In all the above examples, only the URL sent to the server is http://www.example.com the URL’s "#/about" and #/contact is never sent to the server

HTML 5 routing

The introduction of HTML5, now allows browsers to programmatically alter the browser’s history through the history object.

Using [history.pushState()](https://developer.mozilla.org/en-US/docs/Web/API/History_API) method, we can now programmatically add the browser history entries and change the location without triggering a server page request.

The [history.pushState](https://www.sitepoint.com/javascript-history-pushstate/) method accepts the following three parameters.

**State object:** A state object is a JavaScript object which is associated with the new history entry created by pushState()

**Title:** This is an optional title for the state

**URL:** The new history entry’s URL. The browser won’t jump to that page.

For example

|  |  |
| --- | --- |
| 1  2  3  4 | var stateObj= { message: "some message" };  history.pushState(stateObj, "title", newUrl); |

Using history.pushState the method, The browser creates new history entries that change the displayed URL without the need for a new request.

Example

When you request for http://www.example.com the server sends the index.html

Now, When you click on ProductList link, Angular use’s the history.pushState method to push the state and change the URL to http://www.example.com/ProductList

Now, when you click on the specific Product, we again the use history method to push the state and change the URL to http://www.example.com/product/1

Here, when you click the back button, the browser will retrieve the http://www.example.com/ProductList from history and displays it.

But there are cons to this approach

Not all browsers support HTML 5

The older browser does not support HTML5. So if you want to support older browser, you have to stick to the hash style routing

The server support is needed for HTML5 based routing.

Why Server Support Needed for HTML 5 routing

Now, consider the above example

What would happen, when you type the URL http://www.example.com/ProductList and hit the refresh button.

The browser will send the request to the webserver. Since the page ProductList does not exist, it will return the 404 (page not found) error.

This problem could be solved, if we are able to redirect all the request to the index.html

It means that when you ask from http://www.example.com/ProductList, the Web server must redirect it to index.html and return the request. Then in the Front-end Angular will read the URL and dynamically load the ProductListComponent.

To make HTML5 routing work you need to send the instruction to the webserver to serve /index.html for any incoming request, no matter what the path is.

Location Strategy

As mentioned earlier, Angular implements both Hashstyle & HTML 5 Routing. HashLocationstrategy implements the Hashstyle routing & Pathlocationstrategy implements the HTML5 style routing

PathLocationStrategy Vs HashLocationStrategy

PathLocationStrategy

**Pros:**

Produces a clear URL like http://example.com/foo

Supports Server-Side Rendering

Server-side Rendering is a technique that renders critical pages on the server that can greatly improve perceived responsiveness when the app first loads

Cons:

Older browser does not support

Server Support needed for this to work

HashLocationStrategy

**Pros:**

Supported by all browsers

**Cons:**

Produces a URL like http://example.com/#foo

Will not Support Server-Side Rendering

PathLocationStrategy

The PathLocationStrategy is the default strategy in Angular application.

To Configure the strategy, we need to add <base href> in the <head> section of root page (index.html) of our application

|  |  |
| --- | --- |
| 1  2  3 | <base href="/"> |

The Browser uses this element to construct the relative URLs for static resources (images, CSS, scripts) contained in the document.

If you do not have access to <head> Section of the index.html, then you can follow either of the two steps

Add the APP\_BASE\_HREF value as shown in the[provider’s](https://www.tektutorialshub.com/angular/angular-providers/) section of the root module

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | import {Component, NgModule} from '@angular/core';  import {APP\_BASE\_HREF} from '@angular/common';    @NgModule({  providers: [{provide: APP\_BASE\_HREF, useValue: '/my/app'}]  })  class AppModule {} |

or use the absolute path for all the static resources like CSS, images, scripts, and HTML files.

HashLocationStrategy

You can use the HashLocationStrategy by providing the useHash: true in an object as the second argument of the RouterModule.forRoot in the AppModule.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @NgModule({  declarations: [      AppComponent,HomeComponent,ContactComponent,ProductComponent,ErrorComponent  ],  imports: [      BrowserModule,      FormsModule,      HttpModule,      Hashlocationstrategy RouterModule.forRoot(appRoutes, { useHash: true }  ],  providers: [ProductService],  bootstrap: [AppComponent]  }) |

Which Location Strategy to Use

We recommend you to use the HTML 5 style (PathLocationStrategy ) as your location strategy.

Because

It produces clean and SEO Friendly URLs that are easier for users to understand and remember.

You can take advantage of the server-side rendering, which will make our application load faster, by rendering the pages in the server first before delivering it the client

Use hash location strategy only if you have to support the older browsers.

Summary

Angular supports two different location strategies or Routing strategy in Angular. One is PathlocationStrategy and the other one is HashLocationStrategy. The HashLocationStrategy use the Hash style routing, while PathlocationStrategy uses the HTML 5 Routing.

What are Route Parameters

The Route parameters are a dynamic part of the Route and essential in determining the route.

For example, consider the following route

|  |  |
| --- | --- |
| 1  2  3 | { path: 'product', component: ProductComponent } |

The above route match only if the URL is /product

To retrieve the product details, our URL should look something like

/product/1  
/product/2

Where the second URL segment ( 1 and 2 ) being the id of the product. The **id** is dynamic and changes as per the selected Product. To handle such scenario[angular router](https://www.tektutorialshub.com/angular/angular-routing-navigation/) allows us to include **route parameters**, where we can send any dynamic value for a URL segment

How to Pass parameters to Angular Route

Defining the Route

We can define parameter by adding forward slash followed colon and a placeholder (id) as shown below

|  |  |
| --- | --- |
| 1  2  3 | { path: 'product/:id', component: ProductDetailComponent } |

Now above path matches the URLs /product/1 , /product/2, etc.

If you have more than one parameter, then you can extend it by adding one more forward slash followed colon and a placeholder

|  |  |
| --- | --- |
| 1  2  3 | { path: 'product/:id/:id1/:id2', component: ProductDetailComponent } |

The name id, id1 & id2 are placeholders for parameters. We will use them while retrieving the values of the parameters.

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Defining the Navigation

We, now need to provide both **path** and the**route parameter** routerLink directive.

This is done by adding the productID as the second element to the routerLink parameters array as shown below

|  |  |
| --- | --- |
| 1  2  3 | <a [routerLink]="['/Product', ‘2’]">{{product.name}} </a> |

Which translates to the URL /product/2

OR

|  |  |
| --- | --- |
| 1  2  3 | <a [routerLink]="['/Product', product.productID]">{{product.name}} </a> |

Which, dynamically takes the value of id from the product object.

Retrieve the parameter in the component

Finally, our component needs to extract the route parameter from the URL

This is done via the **ActivatedRoute** service from angular/router module to get the parameter value

ActviatedRoute

The [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) is a service, which keeps track of the currently activated route associated with the loaded Component.

To use ActivatedRoute, we need to import it in our component

|  |  |
| --- | --- |
| 1  2  3 | import { ActivatedRoute } from '@angular/router'; |

Then inject it into the component using dependency injection

|  |  |
| --- | --- |
| 1  2  3 | constructor(private \_Activatedroute:ActivatedRoute) |

ParamMap

The Angular adds the map all the route parameters in the ParamMap object, which can be accessed from the ActivatedRoute service

The [ParamMap](https://angular.io/api/router/ParamMap) makes it easier to work with parameters. We can use get or getAll methods to retrieve the value of the parameters in the component. Use the has method to check if a certain parameter exists.

The Older version of ActivatedRoute class has a Params array which is an array of the parameter values, indexed by name. You can still use it but It is now deprecated and is replaced by the ParamMap.

There are two ways in which you can use the ActivatedRoute to get the parameter value from the ParamMap object.

Using Snapshot

Using observable

Using Snapshot

|  |  |
| --- | --- |
| 1  2  3 | this.id=this.\_Activatedroute.snapshot.paramMap.get("id"); |

The snapshot property returns the initial value of the route. You can then access the paramsMap array, to access the value of the id, as shown above,

Use the Snapshot option, if you only need the initial value.

Using Observable

|  |  |
| --- | --- |
| 1  2  3  4  5 | this.\_Activatedroute.paramMap.subscribe(params => {      this.id = params.get('id');  }); |

You can retrieve the value of id by subscribing to the paramMap observable property of the activateRoute as shown above

Use this option if you expect the value of the parameter to change over time.

Why use observable

We usually retrieve the value of the parameter in the [**ngOninit**](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/) life cycle hook, when the component initialised.

When the user navigates to the component again, the Angular does not create the new component but reuses the existing instance. In such circumstances, the ngOnInit method of the component is not called again. Hence you need a way to get the value of the parameter.

By subscribing to the observable paramMap property, you will retrieve the latest value of the parameter and update the component accordingly.

The above difference is explained in our next tutorial [Angular child routes tutorial](https://www.tektutorialshub.com/angular/angular-child-routes-nested-routes/).

Passing Parameters to Route: Example

Here is the complete list of code.

**product.component.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | import { Component, OnInit } from '@angular/core';    import { ProductService } from './product.service';  import { Product } from './product';    @Component({    templateUrl: './product.component.html',  })    export class ProductComponent  {       products:Product[];       constructor(private productService:ProductService){     }       ngOnInit() {       this.products=this.productService.getProducts();     }    } |

**product.component.html**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | <h1>Product List</h1>  <div class='table-responsive'>      <table class='table'>          <thead>              <tr>                  <th>ID</th>                  <th>Name</th>                  <th>Price</th>              </tr>          </thead>          <tbody>              <tr \*ngFor="let product of products;">                  <td>{{product.productID}}</td>                  <td><a [routerLink]="['/product',product.productID]">{{product.name}} </a> </td>                  <td>{{product.price}}</td>              </tr>          </tbody>      </table>  </div> |

**product.service.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | import { Observable } from 'rxjs';  import {Product} from './Product'      export class ProductService{        public getProducts() {            let products:Product[];            products=[              new Product(1,'Memory Card',500),              new Product(2,'Pen Drive',750),              new Product(3,'Power Bank',100)          ]            return products;      }          public getProduct(id) {          let products:Product[]=this.getProducts();          return products.find(p => p.productID==id);      }      } |

**product.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | export class Product {        constructor(productID:number,    name: string ,   price:number) {          this.productID=productID;          this.name=name;          this.price=price;      }        productID:number ;      name: string ;      price:number;    } |

In the product.component.html, we have added product.productID as the second argument to the routerLink parameters array.

|  |  |
| --- | --- |
| 1  2  3 | <a [routerLink]="['/product',product.productID]">{{product.name}} </a> |

Product Details Component

**product-detail.component.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70 | import { Component, OnInit, OnDestroy } from '@angular/core';  import { Router,ActivatedRoute } from '@angular/router';    import { ProductService } from './product.service';  import { Product } from './product';      @Component({    templateUrl: './product-detail.component.html',  })    export class ProductDetailComponent  {       product:Product;     id;       constructor(private \_Activatedroute:ActivatedRoute,                 private \_router:Router,                 private \_productService:ProductService){     }         /\* Using snapshot \*/     // ngOnInit() {       //    //This still works but is deprecated     //    //this.id=this.\_Activatedroute.snapshot.params['id'];       //    this.id=this.\_Activatedroute.snapshot.paramMap.get("id");         //    let products=this.\_productService.getProducts();     //    this.product=products.find(p => p.productID==this.id);     // }         /\* Using Subscribe \*/       sub;       ngOnInit() {          this.sub=this.\_Activatedroute.paramMap.subscribe(params => {           console.log(params);            this.id = params.get('id');            let products=this.\_productService.getProducts();            this.product=products.find(p => p.productID==this.id);        });          // This params is deprecated          //this.sub=this.\_Activatedroute.params.subscribe(params => {        //    this.id = params['id'];        //    let products=this.\_productService.getProducts();        //    this.product=products.find(p => p.productID==this.id);        //        //});     }       ngOnDestroy() {       this.sub.unsubscribe();     }       onBack(): void {        this.\_router.navigate(['product']);     }  } |

**product-detail.component.html**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <h3>Product Details Page</h3>      product : {{product.name}}  price : {{ product.price}}  <p>      <a class='btn btn-default' (click)="onBack()">Back </a>  </p> |

In the ProductDetailComponent, we have imported router and ActivatedRoute from the angular router module

|  |  |
| --- | --- |
| 1  2  3  4 | import { Component, OnInit, OnDestroy } from '@angular/core';  import { Router,ActivatedRoute } from '@angular/router'; |

In the constructor, we inject the ActivatedRoute, Router service along with ProductService

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | constructor(private \_Activatedroute:ActivatedRoute,                 private \_router:Router,                 private \_productService:ProductService){     } |

Finally, we use ngOninit life cycle hook to retrieve the value of the id parameter and use that value to retrieve the details of the product.

Note that, there are two ways, by which you can retrieve the data.

**Using snapshot**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | ngOnInit() {         //This still works but is deprecated       //this.id=this.\_Activatedroute.snapshot.params['id'];         this.id=this.\_Activatedroute.snapshot.paramMap.get("id");           let products=this.\_productService.getProducts();       this.product=products.find(p => p.productID==this.id);  } |

**Using Subscribe**

We used snapshot method to retrieve the parameter in the ProductDetailcomponet.ts. To Subscribe to params remove the ngOnInit and replace it with the following code

We recommend you to use the subscribe method as it offers the benefit of responding to the parameter changes dynamically.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | ngOnInit() {          this.sub=this.\_Activatedroute.paramMap.subscribe(params => {           console.log(params);            this.id = params.get('id');            let products=this.\_productService.getProducts();            this.product=products.find(p => p.productID==this.id);        });          // This params is deprecated          //this.sub=this.\_Activatedroute.params.subscribe(params => {        //    this.id = params['id'];        //    let products=this.\_productService.getProducts();        //    this.product=products.find(p => p.productID==this.id);        //        //});     } |

The Routes

**app.routing.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import { Routes } from '@angular/router';    import { HomeComponent} from './home.component'  import { ContactComponent} from './contact.component'  import { ProductComponent} from './product.component'  import { ErrorComponent} from './error.component'    import { ProductDetailComponent} from './product-detail.component'    export const appRoutes: Routes = [    { path: 'home', component: HomeComponent },    { path: 'contact', component: ContactComponent },    { path: 'product', component: ProductComponent },    { path: 'product/:id', component: ProductDetailComponent },    { path: '', redirectTo: 'home', pathMatch: 'full' },    { path: '\*\*', component: ErrorComponent }  ]; |

We have added the following route to our routes array

|  |  |
| --- | --- |
| 1  2  3 | { path: 'product/:id', component: ProductDetailComponent }, |

Other components

**app.component.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'Routing Module - Parameters Demo';  } |

**app.component.html**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | <div class="container">    <nav class="navbar navbar-default">    <div class="container-fluid">      <div class="navbar-header">        <a class="navbar-brand" [routerLink]="['/']"><strong> {{title}} </strong></a>      </div>      <ul class="nav navbar-nav">          <li><a [routerLink]="['home']">Home</a></li>          <li><a [routerLink]="['product']">Product</a></li>          <li><a [routerLink]="['contact']">Contact us</a></li>      </ul>    </div>  </nav>    <router-outlet></router-outlet>    </div> |

**contact.component.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import {Component} from '@angular/core';    @Component({       template: `<h1>Contact Us</h1>                  <p>TekTutorialsHub </p>                  `  })  export class ContactComponent {  } |

**home.component.ts**

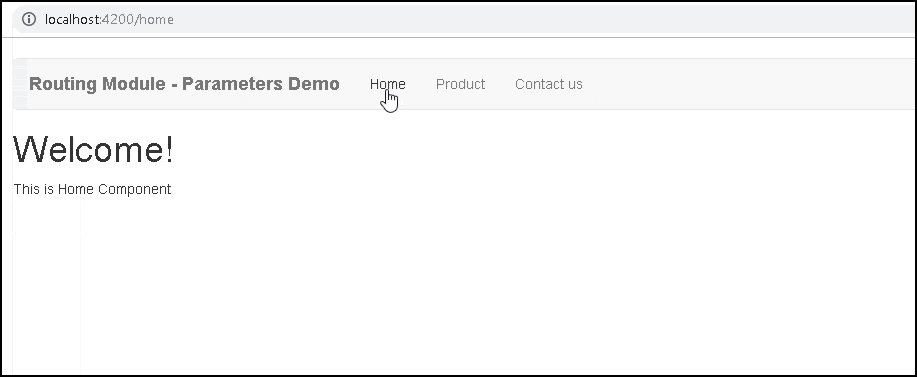
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import {Component} from '@angular/core';    @Component({      template: `<h1>Welcome!</h1>                <p>This is Home Component </p>               `  })    export class HomeComponent {  } |

**error.component.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import {Component} from '@angular/core';    @Component({      template: `<h1>Page not found</h1>                 <p>This is a Error Page</p>                `  })    export class ErrorComponent {  } |

**app.module.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule } from '@angular/forms';  import { HttpModule } from '@angular/http';    import { RouterModule } from '@angular/router';    import { AppComponent } from './app.component';  import { HomeComponent} from './home.component'  import { ContactComponent} from './contact.component'  import { ProductComponent} from './product.component'  import { ErrorComponent} from './error.component'  import { ProductDetailComponent} from './product-detail.component'    import { ProductService } from './product.service';    import { appRoutes } from './app.routes';    @NgModule({    declarations: [      AppComponent,HomeComponent,ContactComponent,ProductComponent,ErrorComponent,ProductDetailComponent    ],    imports: [      BrowserModule,      FormsModule,      HttpModule,      RouterModule.forRoot(appRoutes)    ],    providers: [ProductService],    bootstrap: [AppComponent]  })  export class AppModule { } |



ActivatedRoute

The ActivatedRoute service has a great deal of useful information including:

**url:** This property returns an array of Url Segment objects, each of which describes a single segment in the URL that matched the current route.

**params:** This property returns a Params object, which describes the URL parameters, indexed by name.

**queryParams:** This property returns a Params object, which describes the URL query parameters, indexed by name.

**fragment:** This property returns a string containing the URL fragment.

**Snapshot:** The initial snapshot of this route

**data:** An Observable that contains the data object provided for the route

**Component:** The component of the route. It’s a constant

**outlet:** The name of the RouterOutlet used to render the route. For an unnamed outlet, the outlet name is primary.

**routeConfig:** The route configuration used for the route that contains the origin path.

**parent:** an ActivatedRoute that contains the information from the parent route when using child routes.

**firstChild:** contains the first ActivatedRoute in the list of child routes.

**children:** contains all the child routes activated under the current route

**pathFromRoot:** The path from the root of the router state tree to this route

Conclusion

We looked at how to pass parameters or data to the Route. The parameters are passed to the route by using routerLink parameters in the routerLink directive. We retrieve the parameters from ActivatedRoute Service by reading params collection of the snapshot object or by subscribing to the params observable.

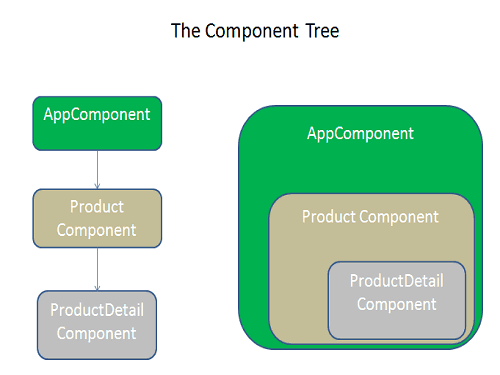
Child Routes / Nested Routes

The Angular 2 applications are based on the idea of Components. The Components follows a Tree structure, where we have a root component at the top. We can then add child components forming loosely coupled components resembling a Tree

The Routes in Angular also follows the component tree structure and allows us to define the nested or child routes.

Example

Consider the following Component Tree



In the above example, the ProductComponent displays the list of Products. The ProductDetailsComponent is defined as the child of the ProductComponent displays the details of the selected Product.

So Our routes would be /Product and /Product/Details/:Id

How to Create Child Routes / Nested Routes

This tutorial builds on the app we have built in the [Passing Parameters to Route Tutorial](https://www.tektutorialshub.com/angular/angular-passing-parameters-to-route/). You can download the code from [gitHub](https://github.com/tekTutorialsHub/Angular-Router).

We already have created ProductDetailsComponent, but it is not designed as the child route of the ProductComponent. Let us update the code make it child route of the Product route.

Define the Routes

Open the app.routing.ts file

You will see the following routes defined in our application.

Here the ProductDetailComponent is defined as the sibling of the ProductComponent and not as the child

|  |  |
| --- | --- |
| 1  2  3  4 | { path: 'product', component: ProductComponent },  { path: 'product/:id', component: ProductDetailComponent }, |

To make ProductDetailComponent as the child of the ProductComponent, we need to add the children key to the product route, which is an array of all child routes as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | { path: 'product', component: ProductComponent,    children: [       { path: 'detail/:id', component: ProductDetailComponent }    ], |

The child route definition is similar to the parent route definition. It has a path and component that gets invoked when the user navigates to the child route.

In the above example, the parent route path is ‘product’ and the child route is ‘detail/:id’

This is will match the URL path “/product/detail/id”.

When the user navigates to the “/product/detail/id”, the router will start to look for a match in the routes array

It starts off the first URL segment that is ‘product’ and finds the match in the path ‘product’ and instantiates the ProductComponent and displays it in the <router-outlet> directive of its parent component ( which is AppComponent)

The router then takes the remainder of the URL segment ‘detail/id’ and continues to search for the child routes of Product route. It will match it with the path ‘detail/:id’ and instantiates the ProductDetailComponent and renders it in the <router-outlet> directive present in the ProductComponent

Final app.routing.ts looks like this

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import { Routes } from '@angular/router';    import { HomeComponent} from './home.component'  import { ContactComponent} from './contact.component'  import { ProductComponent} from './product.component'  import { ErrorComponent} from './error.component'    import { ProductDetailComponent} from './product-detail.component'    export const appRoutes: Routes = [    { path: 'home', component: HomeComponent },    { path: 'contact', component: ContactComponent },    { path: 'product', component: ProductComponent,      children: [        { path: 'detail/:id', component: ProductDetailComponent }      ]     },    { path: '', redirectTo: 'home', pathMatch: 'full' },    { path: '\*\*', component: ErrorComponent }  ]; |

Display the component using Router-outlet

The components are always rendered in the <RouterOutlet> of the parent component.

For ProductDetailComponent the parent component is ProductComponent and not the AppComponent

Hence, we need to add <router-outlet></router-outlet> in the product.component.html as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | <h1>Product List</h1>  <div class='table-responsive'>      <table class='table'>          <thead>              <tr>                  <th>ID</th>                  <th>Name</th>                  <th>Price</th>              </tr>          </thead>          <tbody>              <tr \*ngFor="let product of products;">                  <td>{{product.productID}}</td>                  <td><a [routerLink]="['detail',product.productID]">{{product.name}} </a> </td>                  <td>{{product.price}}</td>              </tr>          </tbody>      </table>  </div>    <router-outlet></router-outlet> |

There is no change in the Product Detail Component.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | import { Component, OnInit } from '@angular/core';  import { Router,ActivatedRoute } from '@angular/router';    import { ProductService } from './product.service';  import { Product } from './product';      @Component({    templateUrl: './product-detail.component.html',  })    export class ProductDetailComponent  {     product:Product;     id;     sub;       constructor(private \_Activatedroute:ActivatedRoute,                 private \_router:Router,                 private \_productService:ProductService){     }       ngOnInit() {         this.id=this.\_Activatedroute.snapshot.params['id'];         let products=this.\_productService.getProducts();         this.product=products.find(p => p.productID==this.id);     }    } |

Note that we are using the **snapshot method** to retrieve the route parameter id.

Testing the Nested/Child Route

Run the app and click on the Product link. You will see that the Product Page is displayed. Click on any of the Product, and you will see the Product details page is displayed

Why subscribe to route params

Now, click on another product, and you will notice that the Product details page does not get updated with the new product.

Why?

Because the angular does not create the component if it is already present in the DOM. It reuses**the component** instance

This implies that the **ngOnInit life cycle** hook is not invoked when the user navigates to the component again. We are retrieving the parameter value in the ngOnInit using the snapshot method. Hence our component does not update itself.

This issue can be rectified by subscribing to the observable params property. Our component will be notified, whenever the value of the parameter changes. So that we can update the component accordingly.

Using the Subscribe method to retrieve the parameters in child routes

Now, open the product-detail.component.ts and change the ngOnInit method to subscribe to the params property as shown below

sub;

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | ngOnInit() {      this.sub=this.\_Activatedroute.params.subscribe(params => {          this.id = params['id'];          let products=this.\_productService.getProducts();          this.product=products.find(p => p.productID==this.id);        });     } |

We need to unsubscribe when the component is destroyed so as to stop the memory leakage

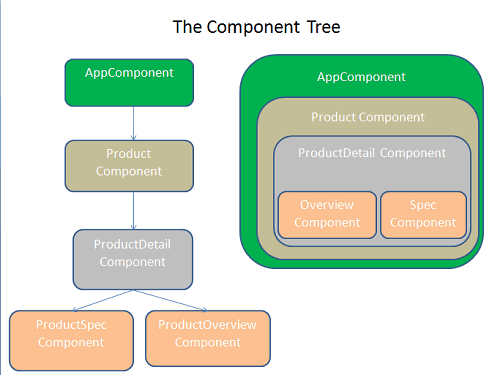
|  |  |
| --- | --- |
| 1  2  3  4  5 | ngOnDestroy() {      this.sub.unsubscribe();  } |

Now, you will see that as you click on another product, the ProductDetailComponents updates itself.

Nesting Children’s under a child

We can add child routes to a child route.

For Example, What if we want to show Product Overview & Specification under the Product Details Page. Our Component Tree is as shown below



Defining the child Route

First, we need to add three child routes under the Product/Details route as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | export const appRoutes: Routes = [    { path: 'home', component: HomeComponent },    { path: 'contact', component: ContactComponent },    { path: 'product', component: ProductComponent,        children: [          { path: 'detail/:id', component: ProductDetailComponent,              children : [                  { path: 'overview', component: ProductOverviewComponent },                  { path: 'spec', component: ProductSpecComponent },                  { path: '', redirectTo:'overview', pathMatch:"full" }              ]          }        ]      },    { path: '', redirectTo: 'home', pathMatch: 'full' },    { path: '\*\*', component: ErrorComponent }  ]; |

The first two child routes are simple. ‘Overview’ path is associated with ProductOverviewComponent & ‘spec’ URL path is associated with the ProductSpecComponent

The Url would become ‘/product/detail/:id/overview’ and ‘/product/detail/:id/spec’

The last route is an empty path which is redirected to ‘Overview’ route. Note that **pathMatch** is set to ‘full’

Mapping the action to View

The updated code for Product Details page is shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | <h1>Product Details Page</h1>    product : {{product.name}}  price : {{ product.price}}    <ul class="nav navbar-nav">      <li><a [routerLink]="['overview']">OverView </a></li>      <li><a [routerLink]="['spec']">Specification </a></li>  </ul>    <router-outlet></router-outlet>    <p>      <a class='btn btn-default' (click)="onBack()">Back to Product List </a>  </p> |

We are using a relative path to while binding path to the routerlink directive. Absolute paths will begin with a forward slash /. When using the relative path, the router will append the path to the parent route path to construct the final URL.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <ul class="nav navbar-nav">      <li><a [routerLink]="['overview']">OverView </a></li>      <li><a [routerLink]="['spec']">Specification </a></li>  </ul> |

Both ProductOverviewComponent & ProductSpecComponent are rendered inside the ProductDetailComponent. Hence we need to add <router-outlet></router-outlet> in the Template of ProductDetailComponent

|  |  |
| --- | --- |
| 1  2  3 | <router-outlet></router-outlet> |

The Child Components

The ProductOverviewComponent just displays the text “Overview of <Name of the Product>” Message.

The most important point is how we retrieve the product id from the route. We are subscribing to the params array of the parent component.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import { Component, OnInit } from '@angular/core';  import { Router,ActivatedRoute } from '@angular/router';    import { ProductService } from './product.service';  import { Product } from './product';    @Component({    template: `<h3> Overview of {{product.name}} <h3>`  })    export class ProductOverviewComponent  {     product:Product;     id;     sub;       constructor(private \_Activatedroute:ActivatedRoute,                 private \_router:Router,                 private \_productService:ProductService){     }     ngOnInit() {          this.sub=this.\_Activatedroute.parent.params.subscribe(params => {            this.id = params['id'];            let products=this.\_productService.getProducts();            this.product=products.find(p => p.productID==this.id);        });     }       ngOnDestroy() {       this.sub.unsubscribe();     }    } |

ProductSpecComponent is similar to the above

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | import { Component, OnInit } from '@angular/core';    import { Router,ActivatedRoute } from '@angular/router';      import { ProductService } from './product.service';    import { Product } from './product';      @Component({      template: `<h3> Specification of {{product.name}} <h3>`    })      export class ProductSpecComponent    {        product:Product;      id;      sub;        constructor(private \_Activatedroute:ActivatedRoute,                  private \_router:Router,                  private \_productService:ProductService){      }        ngOnInit() {            this.sub=this.\_Activatedroute.parent.params.subscribe(params => {              this.id = params['id'];              let products=this.\_productService.getProducts();              this.product=products.find(p => p.productID==this.id);            });      }        ngOnDestroy() {        this.sub.unsubscribe();      }      } |

Finally, do not forget to import both the components in app.routes.ts & app.module.ts

What are query parameters

Query parameters are optional parameters that you pass to a route. The query parameters are added to the end of the URL Separated by Question Mark

For Example

|  |  |
| --- | --- |
| 1  2  3 | /product?page=2 |

where page=2 is the query parameter

The above URL is an example of paginated product list, where URL indicates that second page of the Product list is to be loaded.

Difference between Query parameter and Route parameter

The [route parameters](https://www.tektutorialshub.com/angular/angular-passing-parameters-to-route/) are required and is used by [Angular Router](https://www.tektutorialshub.com/angular/angular-routing-navigation/) to determine the route. They are part of the route definition.

For Example, when we define the route as shown below, the id is the route parameter.

|  |  |
| --- | --- |
| 1  2  3  4 | { path: 'product', component: ProductComponent }  { path: 'product/:id', component: ProductDetailComponent } |

The above route matches the following URL The angular maps the values 1 & 2 to the id field

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | URL  /product         matches => path: 'product'  /product/1       matches => path: 'product/:id'  /product/2       matches => path: 'product/:id' |

The Router will not navigate to the ProductDetailComponent route, if the id is not provided. It will navigate to ProductComponent instead. If the product route is not defined, then it will result in a error.

However, the query parameters are optional. The missing parameter does not stop angular from navigating to the route. The query parameters are added to the end of the URL Separated by Question Mark

**Route Parameters or Query Parameters?**

Use route parameter when the value is required

Use query parameter, when the value is optional.

How to use Query Parameters

The Query parameters are not part of the route. Hence you do not define them in the routes array like route parameters. You can add them using the routerlink directive or via router.navigate method.

Passing Query Parameters

Use the queryParams directive to add the query parameter. Use this directive along with the routerlink directive as shown below

|  |  |
| --- | --- |
| 1  2  3 | <a [routerLink]="['product']" [queryParams]="{ page:2 }">Page 2</a> |

The router will construct the URL as /product?pageNum=2

You can pass more than one Query Parameter as shown below

|  |  |
| --- | --- |
| 1  2  3 | <a [routerLink]="['product']" [queryParams]="{ val1:2 , val2:10}">Whatever</a> |

The router will construct the URL as /product?val1=2&val2=10

You can also navigate programmatically using the navigate method of the Router service as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5 | goToPage(pageNum) {      this.router.navigate(['/product'], { queryParams: { page: pageNum } });  } |

Reading Query Parameters

Reading the Query parameters is similar to reading the Router Parameter. There are two ways by which you can retrieve the query parameters.

Note that queryParams is deprecated. It is replaced by the queryParamMap.

Using queryParamsMap observable

The queryParamsMap is a Observable that contains a [map](https://angular.io/api/router/ParamMap) of the query parameters available to the current route. We can use this to retrieve values from the query parameter. The  
queryParamsMap is accessible via ActivatedRoute

Hence, we need to inject the ActivatedRoute in the constructor of the component/service,where we want to read the query parameter as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5 | constructor(private Activatedroute:ActivatedRoute,                 private router:Router){     } |

You can subscribe to the queryParamMap of the ActivatedRoute, which returns the  
observable of type [ParamMap](https://angular.io/api/router/ParamMap). We can then use the get method to read the query parameter as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | this.sub = this.Activatedroute.queryParamMap         .subscribe(params => {       this.pageNum = +params.get('pageNum')||0;  }); |

Using snapshot.queryParamMap property

You can also read the value of the query parameter from queryParamMap using the snapshot property of the ActivatedRoute as shown below

|  |  |
| --- | --- |
| 1  2  3 | this.Activatedroute.snapshot.queryParamMap.get('pageNum')||0;; |

Remember, the router populates the snapshot, when the component loads for the first time. Hence you will read only the initial value of the query parameter with the snapshot property. You will not be able to retrieve any subsequent changes to the query parameter.

queryParamsHandling

The query parameter is lost when the user navigates to another route.

For Example, if user navigates to the product page with route /product?pageNum=2 then he navigates to the product detail page, the angular removes the query parameter from the url. This is the default behaviour

You can change this behavior by configuring the queryParamsHandling strategy. This Configuration strategy determines how the angular router handles query parameters, when user navigates away from the current route. It has three options

queryParamsHandling : null

This is default option. The angular removes the query parameter from the URL, when navigating to the next..

|  |  |
| --- | --- |
| 1  2  3 | this.router.navigate(['product'], { queryParams: { pageNum: this.pageNo + 1 }, queryParamsHandling :null}   ); |

|  |  |
| --- | --- |
| 1  2  3 | <a [routerLink]="['product']" [queryParams]="{ pageNum:2 }">Page 2</a> |

queryParamsHandling : preserve

The Angular preserves or carry forwards the query parameter of the current route to next navigation. Any query parameters of the next route are discarded

|  |  |
| --- | --- |
| 1  2  3 | this.router.navigate(['product'], { queryParams: { pageNum: this.pageNo + 1 }, queryParamsHandling :"preserve"}   ); |

|  |  |
| --- | --- |
| 1  2  3 | <a [routerLink]="['product']" [queryParams]="{ pageNum:2 }" queryParamsHandling="preserve">Page 2</a> |

queryParamsHandling : merge

The Angular merges the query parameters from the current route with that of next route before navigating to the next route.

|  |  |
| --- | --- |
| 1  2  3 | this.router.navigate(['product'], { queryParams: { pageNum: this.pageNo + 1 }, queryParamsHandling :"merge"}   ); |

|  |  |
| --- | --- |
| 1  2  3 | <a [routerLink]="['product']" [queryParams]="{ pageNum:2 }" queryParamsHandling="merge">Page 2</a> |

Note that [preserveQueryParams](https://angular.io/api/router/NavigationExtras#preserveQueryParams)is DEPRECATED

Query Params Example

Let us build a small application to demonstrate the use of Query parameters

Let us update the app that we have built in the [Passing Parameters to Route tutorial](https://www.tektutorialshub.com/angular/angular-passing-parameters-to-route/). You can download the code from [gitHub](https://github.com/tekTutorialsHub/Angular-Router). From the folder “Parameters” and the Final code at “QueryParameters”

AppComponent

app.component.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'Routing Module - Query Parameters';    pageNum=0;    } |

app.component.html﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | <div class="container">    <nav class="navbar navbar-default">    <div class="container-fluid">      <div class="navbar-header">        <a class="navbar-brand" href="#"><strong> {{title}} </strong></a>      </div>        <ul class="nav navbar-nav">          <li><a [routerLink]="['home']">Home</a></li>          <div class="navbar-form navbar-left">          <div class="form-group">            <input type="text" class="form-control" placeholder="Page No"  [(ngModel)]="pageNum">          </div>        </div>        <li class="nav"><a [routerLink]="['product']" [queryParams]="{ pageNum: pageNum }">Product</a></li>        <li><a [routerLink]="['contact']">Contact us</a></li>      </ul>    </div>  </nav>    <router-outlet></router-outlet>    </div> |

We have defined the pageNum variable in the component. An input box for the pageNum is provided, so that the user can change the page no.

The AppComponent has the navigation Menu. It contains the link to the product page. The link is created using the routerlink directive. The routerlink directive contains the queryParams directive, where we pass the pageNum to "pageNum" variable.

|  |  |
| --- | --- |
| 1  2  3 | <li class="nav"><a [routerLink]="['product']" [queryParams]="{ pageNum: pageNum }">Product</a></li> |

The Angular will construct the URL as /product?pageNum=2

Product Component

The ProductComponent does not display any products. But it displays the query parameters received

product.component.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | import { Component, OnInit } from '@angular/core';  import { ActivatedRoute, Router} from '@angular/router';      @Component({    templateUrl: './product.component.html',  })    export class ProductComponent implements OnInit  {       pageNo=0;     snapshotPageNo=0;       constructor(private Activatedroute:ActivatedRoute,                 private router:Router){     }         ngOnInit() {        //  the following code works but is DEPRECATED      //  this.snapshotPageNo =this.Activatedroute.snapshot.queryParams['pageNum']||0;        //  this.Activatedroute.queryParams      //       .subscribe(params => {      //         this.pageNo = +params['pageNum']||0;      //         console.log('Query params ',this.pageNo)      //       });             this.snapshotPageNo =+this.Activatedroute.snapshot.queryParamMap.get('pageNum')||0;;              this.Activatedroute.queryParamMap              .subscribe(params => {                this.pageNo = +params.get('pageNum')||0;                console.log('Query params ',this.pageNo)            });     }       nextPage() {        this.router.navigate(['product'], { queryParams: { pageNum: this.pageNo + 1 }} );     }      } |

product.component.html﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <h1>Product Page</h1>    <p>Current Page No <strong> {{pageNo}} </strong></p>  <p>snapshot Page No  <strong> {{ snapshotPageNo }} </strong></p>    <button (click)="nextPage()">Next Page</button> |

First, we have injected both ActivatedRoute and Router Service in the constructor of the ProductComponent

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | constructor(private Activatedroute:ActivatedRoute,              private router:Router){  } |

Next, in the [ngOnInit lifecycle hook](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/), we use the Activatedroute.snapshot method to retrieve the pageNum and update the snapshotPageNo variable.

We also subscribe to the queryParams property. We are updating our local variable pageNo with page number obtained from the Queryparams

In our template, we display both snapshotPageNo pageNo variable.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | ngOnInit() {          this.snapshotPageNo = this.Activatedroute.snapshot.queryParams['pageNum'] || 0;        this.Activatedroute.queryParams          .subscribe(params => {              this.pageNum = +params['pageNum']||0;             console.log('Query params ',this.pageNum)      });  } |

We also have nextPage method.

|  |  |
| --- | --- |
| 1  2  3  4  5 | nextPage() {        this.\_router.navigate(['./'], { queryParams: { pageNum: this.pageNum + 1 }, relativeTo: this.\_Activatedroute }   );     } |

Here we are using the router.navigate method to navigate to the next page. This actually does not change route as we are already in the page. The angular router does not re create the component, but since we have subscribed to the Query parameters, we will be notified of the change in page Number. The pageNum variable is updated to the new value, while the snapshotPageNo does not change

Other Components

app.module.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule } from '@angular/forms';  import { HttpModule } from '@angular/http';    import { RouterModule } from '@angular/router';    import { AppComponent } from './app.component';  import { HomeComponent} from './home.component'  import { ContactComponent} from './contact.component'  import { ProductComponent} from './product.component'    import { appRoutes } from './app.routes';    @NgModule({    declarations: [      AppComponent,HomeComponent,ContactComponent,ProductComponent    ],    imports: [      BrowserModule,      FormsModule,      HttpModule,      RouterModule.forRoot(appRoutes)    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

app.routes.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import { Routes } from '@angular/router';    import { HomeComponent} from './home.component'  import { ContactComponent} from './contact.component'  import { ProductComponent} from './product.component'      export const appRoutes: Routes = [    { path: 'home', component: HomeComponent },    { path: 'contact', component: ContactComponent },    { path: 'product', component: ProductComponent },    { path: '', redirectTo: 'home', pathMatch: 'full' },  ]; |

home.component.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import {Component} from '@angular/core';    @Component({      template: `<h1>Welcome!</h1>                <p>This is Home Component </p>               `  })    export class HomeComponent {  } |

contact.component.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import {Component} from '@angular/core';    @Component({       template: `<h1>Contact Us</h1>                  <p>TekTutorialsHub </p>                  `  })    export class ContactComponent {  } |

Navigating between Angular routes

You can navigate between routes in Angular 2 in two ways

Using RouterLink Directive

Via Code

RouterLink directive

We looked at how to navigate using the RouterLink directive in the previous tutorials.

The RouterLink is a directive, which you can use to bind any clickable HTML element to a Route. When the user clicks on the HTML element the router will navigate to the associated Route.

For Example

|  |  |
| --- | --- |
| 1  2  3 | <li><a [routerLink]="['product']">Product</a></li> |

Will map to URL “/product” and renders the associated ProductComponent

Navigating Using Code

You can also navigate imperatively by using the code. This is done using the router service, which provides navigate and navigatebyUrl methods via which you can perform route changes.

router.navigate

Use this method, if you want to Navigate to a route using the link parameters array. The first argument to the navigate method is link parameters array, which is similar to what we provide while defining the routerlink directive

***Navigate Method always uses the absolute path unless you provide a starting point.***

navigate.navigateByUrl

Use this method if you want to navigate to a URL by using the absolute path. The first argument is a string containing the complete URL.

***NavigateByUrl Method always uses the absolute path***

To use both these methods, we need to inject router service into our component as shown below

|  |  |
| --- | --- |
| 1  2  3  4 | constructor(private \_router:Router){  } |

And then invoke

|  |  |
| --- | --- |
| 1  2  3 | this.\_router.navigate(['product'] |

Or

|  |  |
| --- | --- |
| 1  2  3 | this.\_router.navigateByUrl('product') |

To navigate to the desired route.

Link Parameters array

LINK Parameters array is an array of strings, which you must specify as argument to either to routerlink directive or navigate method for navigation to work

We need to specify the path of the route and route parameters that go into the route URL.

The following example resolves to the URL path ‘/product/detail/1’

|  |  |
| --- | --- |
| 1  2  3 | <li><a [routerLink]="['product/detail/1']">Product 1 Overview</a></li> |

or

|  |  |
| --- | --- |
| 1  2  3 | this.\_router.navigate(['product/detail/1']) |

Relative and Absolute Paths in Routes

The Angular routes resemble directory-like tree structure.

Hence, We can use directory like syntaxes like add / (root node) , ./ (current node) or ../ (Parent node) in the link parameters array

The First segment of the link parameters array can be prepended with “**/**“, “**./**“, or “**../**“

If the First segment of the route starts with “**/**“, then the path is considered to be Absolute path

If the First segment begins with “**./”** or it does not begin with a slash, then the path is considered to be the relative path.

And if the First segment begins with “.**./**“, then the path is relative to the parent route. (one level up)

router.navigate method and relative path

As mentioned earlier navigate method always uses the absolute path. To make Navigate method work with a relative path, we must let know the router where are we in the route tree.

This done by setting the relativeTo Property to the ActivatedRoute as shown below

|  |  |
| --- | --- |
| 1  2  3 | this.\_router.navigate(['detail'], { queryParams: { pageNum: this.pageNum + 1 }, relativeTo: this.\_Activatedroute } ); |

RouterLink directive and relative path

If you were using a RouterLink to navigate instead of the Router service, you’d use the same link parameters array, but you wouldn’t provide the object with the relativeTo property. The ActivatedRoute is implicit in a RouterLink directive.

Absolute Path Vs Relative Path Which one to Use?

It is recommended to use the Relative path. Using absolute path breaks our code if the parent URL structure changes. The relative path will not change even if the parent path changes

To go to the parent route

|  |  |
| --- | --- |
| 1  2  3 | <li><a [routerLink]="['../']">Back</a></li> |

To go to the Sibling route

|  |  |
| --- | --- |
| 1  2  3 | <li><a [routerLink]="['../<sibling>']">Goto sibling</a></li> |

To go to the child route

|  |  |
| --- | --- |
| 1  2  3 | <li><a [routerLink]="['<Child>']">Goto Child</a></li> |

NavigationExtras

We can provide the extra options to both router.navigate() or router.navigatebyURL() method.

**relativeTo:**ActivatedRoute

Enables relative navigation from the current ActivatedRoute. This is applicable only to router.navigate() method.

Example:

The following Navigates to the Detail route from child route

|  |  |
| --- | --- |
| 1  2  3 | this.router.navigate(['../Detail'], { relativeTo: this.activatedRoute }); |

**queryParams:** Params

Sets query parameters to the URL. You can refer to the tutorial [How to pass query parameters to Angular route](https://www.tektutorialshub.com/angular/angular-passing-optional-query-parameters-to-route/)

Example:

The following code constructs the URL as “/product?page=2”.

|  |  |
| --- | --- |
| 1  2  3 | this.router.navigate(['/products'], { queryParams: { page: 1 } }); |

**fragment:** string  
Sets the hash fragment for the URL.

Example:

The following code constructs the URL as “/home#top”

|  |  |
| --- | --- |
| 1  2  3 | this.router.navigate(['/home'], { fragment: 'top' }); |

**preserveQueryParams:** boolean  
Passes the query parameters of the current route to the next route

Example:

If you are on the route “Product?Page=2”, then the clicking on the following will pass the query parameters to the “view” route as “view?Page=2”

|  |  |
| --- | --- |
| 1  2  3 | this.router.navigate(['/view'], { preserveQueryParams: true }); |

**queryParamsHandling:** QueryParamsHandling

The query parameters of the current route are merged with that of the new route if you set queryParamsHandling=”merge”.

|  |  |
| --- | --- |
| 1  2  3 | this.router.navigate(['/view'], { queryParams: { page: 2 },preserveQueryParams: true, queryParamsHandling: "merge" }); |

**preserveFragment:** boolean  
Passes the fragment of the current route to the next navigation. Similar to the preserveQueryParams

|  |  |
| --- | --- |
| 1  2  3 | this.router.navigate(['/view'], { preserveFragment: true }); |

**skipLocationChange:** boolean  
You can change the route, without changing the URL in the browser.  This Navigates to a new URL without pushing a new state into history.

Example:

|  |  |
| --- | --- |
| 1  2  3 | this.router.navigate(['/view'], { skipLocationChange: true }); |

**replaceUrl:** boolean  
The current route is removed from the browser history while navigating to the new route. It replaces the current state in history with the new state.

Example:

|  |  |
| --- | --- |
| 1  2  3 | this.router.navigate(['/view'], { replaceUrl: true }); |

RouterLink

You can provide the extra options to the RouterLink directive, similar to the NavigationExtras. The following options are supported

**QueryParams: Params**

|  |  |
| --- | --- |
| 1  2  3 | <a [routerLink]="['product']" [queryParams]="{ page:2}" }>Page 2</a> |

**preserveQueryParams:**boolean

|  |  |
| --- | --- |
| 1  2  3 | <a [routerLink]="['product']" { preserveQueryParams: "true" }">Page 2</a> |

**queryParamsHandling : QueryParamsHandling**

|  |  |
| --- | --- |
| 1  2  3 | <a [routerLink]="['product']" { queryParams: { page: 2 }, queryParamsHandling: "merge" }">Page 2</a> |

**Fragment: string**

|  |  |
| --- | --- |
| 1  2  3 | <a [routerLink]="['product']" { fragment: 'top' }">Page 2</a> |

**PreserveFragment: boolean**

|  |  |
| --- | --- |
| 1  2  3 | <a [routerLink]="['product']" { preserveFragment: true }">Page 2</a> |

**SkipLocationChange: boolean**

|  |  |
| --- | --- |
| 1  2  3 | <a [routerLink]="['product']" { skipLocationChange: true">Page 2</a> |

**ReplaceUrl: boolean**

|  |  |
| --- | --- |
| 1  2  3 | <a [routerLink]="['product']" { replaceUrl: true">Page 2</a> |

Conclusion

In this tutorial, we looked at various ways you can navigate between Angular routes, using the Angular router.  We learn how to navigate using either RouterLink directive or using router.navigate or router.navigatebyUrl method provided by the router service. We learned how to set up relative and absolute path routing. Finally, we looked at the various options that are provided by these navigation methods

Angular Route Guards

We use the Angular Guards to control, whether the user can navigate to or away from the current route.

Why Guards

We looked at how to configure our routes and navigate to the different parts of our application in our [Angular Router Tutorial](https://www.tektutorialshub.com/angular/angular-routing-navigation/). Allowing the user to navigate all parts of the application is not a good idea. We need to restrict the user until the user performs specific actions like login. Angular provides the **Route Guards** for this purpose.

One of the common scenario, where we use Route guards is authentication. We want our App to stop the unauthorized user from accessing the protected route. We achieve this by using the [**CanActivate guard**](https://www.tektutorialshub.com/angular/angular-canactivate-guard-example/), which angular invokes when the user tries to navigate into the protected route. Then we hook into the [**CanActivate guard**](https://www.tektutorialshub.com/angular/angular-canactivate-guard-example/) and use the authentication service to check whether the user is authorized to use the route and if not we can redirect the user to the login page.

Uses of  Angular Route Guards

To Confirm the navigational operation

Asking whether to save before moving away from a view

Allow access to certain parts of the application to specific users

Validating the route parameters before navigating to the route

Fetching some data before you display the component.

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

Types of Route Guards

The Angular Router supports Five different guards, which you can use to protect the route

CanActivate

CanDeactivate

Resolve

CanLoad

CanActivateChild

CanActivate

This guard decides if a route can be activated (or component gets used). This guard is useful in the circumstance where the user is not authorized to navigate to the target component. Or the user might not be logged into the system

Read: [Angular CanActivate Guard Example](https://www.tektutorialshub.com/angular/angular-canactivate-guard-example/)

CanDeactivate

This Guard decides if the user can leave the component (navigate away from the current route). This route is useful in where the user might have some pending changes, which was not saved. The CanDeactivate route allows us to ask user confirmation before leaving the component.  You might ask the user if it’s OK to discard pending changes rather than save them.

Read: [Angular CanDeactivate Guard Example](https://www.tektutorialshub.com/angular/angular-candeactivate-guard/)

Resolve

This guard delays the activation of the route until some tasks are complete. You can use the guard to pre-fetch the data from the backend API, before activating the route

CanLoad

The CanLoad Guard prevents the loading of the [Lazy Loaded Module](https://www.tektutorialshub.com/angular/angular-lazy-loading/). We generally use this guard when we do not want to unauthorized user to be able to even see the source code of the module.

This guard works similar to CanActivate guard with one difference. The CanActivate guard prevents a particular route being accessed. The CanLoad prevents entire lazy loaded module from being downloaded, Hence protecting all the routes within that module.

Read: [Angular CanLoad Guard Example](https://www.tektutorialshub.com/angular/angular-canload-guard-example/)

CanActivateChild

This guard determines whether a child route can be activated. This guard is very similar to CanActivateGuard. We apply this guard to the parent route. The Angular invokes this guard whenever the user tries to navigate to any of its child route. This allows us to check some condition and decide whether to proceed with the navigation or cancel it.

Read: [Angular CanActivateChild Guard Example](https://www.tektutorialshub.com/angular/angular-canactivatechild-example/)

How to Build Angular Route Guards

Building the Guards are very easy.

Build the Guard as Service.

Implement the Guard Method in the Service

Register the Guard Service in the Root Module

Update the Routes to use the guards

1. Build the Guard as Service

Building the Guard Service is as simple as building any other [Angular Service](https://www.tektutorialshub.com/angular/angular-services/). You need to import the corresponding guard from the Angular Router Library using the Import statement. For Example to use CanActivate Guard import the CanActivate in the import  the CanActivate in the import statement

|  |  |
| --- | --- |
| 1  2  3 | import { CanActivate } from '@angular/router'; |

Next, create the Guard class which implement the selected guard Interface as shown below.

|  |  |
| --- | --- |
| 1  2  3  4 | @Injectable()  export class ProductGuardService implements CanActivate {} |

You can also inject other services into the Guards using the [Dependency Injection](https://www.tektutorialshub.com/angular/how-dependency-injection-resolution-works-in-angular/)

2. Implement the Guard Method

The next step is to create the Guard Method. The name of the Guard method is same as the Guard it implements. For Example to implement the CanActivate guard, create a method CanActivate

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | canActivate(): boolean {      // Check weather the route can be activated;      return true;      // or false if you want to cancel the navigation;  } |

Return value from the Guard

The guard method must return either a True or a False value.

If it returns true, the navigation process continues. if it returns false, the navigation process stops and the user stays put.

The above method returns a True value. The Guard can also return an Observable or a Promise which eventually returns a True or false. The Angular will keep the user waiting until the guard returns true or false.

The guard can also tell the router to navigate elsewhere, effectively canceling the current navigation.

3. Register the Guard as Service in Module

As mentioned earlier, guards are nothing but services. We need to register them with the Providers array of the [Angular Module](https://www.tektutorialshub.com/angular/angular-modules/) as shown below

|  |  |
| --- | --- |
| 1  2  3 | providers: [ProductService,ProductGuardService] |

The [Angular router](https://www.tektutorialshub.com/angular/angular-navigating-between-routes/) requires the Guards and all other services that guard depends on available during the navigation. Hence the guards must be provided at the module level. This allows the router to access the guards using the [Dependency Injection](https://www.tektutorialshub.com/angular/angular-dependency-injection/).

4. Update the Routes

Finally, we need to add the guards to the routes array as shown below

|  |  |
| --- | --- |
| 1  2  3  4 | { path: 'product', component: ProductComponent, canActivate : [ProductGuardService]  } |

The above code adds the canActivate guard (ProductGuardService) to the  Product route.

When the user navigates to the Product route the Angular calls the canActivate method from the ProductGuardService. If the method returns true then the ProductComponent is rendered.

You can add more than one guard as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | { path: 'product',    component: ProductComponent,    canActivate : [ProductGuardService, AnotherProductGuardService ]  } |

The syntax for adding other guards are also similar

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | { path: 'product', component,      canActivate : any[],      canActivateChild: any[],      canDeactivate: any[],      canLoad: any[],      resolve: any[]  } |

Order of execution of route guards

A route can have multiple guards and you can have guards at every level of a routing hierarchy.

CanDeactivate() and CanActivateChild() guards are always checked first. The checking starts from the deepest child route to the top.

CanActivate() guard is checked next and checking starts from the top to the deepest child route.

CanLoad() is invoked next,  If the feature module is to be loaded asynchronously.

Resolve() Guard is invoked last.

The Angular Router cancels the navigation If any of the guards return false.

Angular Guards Example

Let us update the App, we built in the previous tutorials on Angular Routers, and use theCanActivate Guard to prevent the user from activating the ProductComponent

The source code for this tutorial is available on [GitHub](https://github.com/tekTutorialsHub/Angular-Router)

Guard Service

We create Guard classes are service. Create a file named product-guard.service.ts in the src/app folder and add the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import { Injectable } from '@angular/core';  import { Router, CanActivate, ActivatedRouteSnapshot,RouterStateSnapshot } from '@angular/router';    @Injectable()  export class ProductGuardService implements CanActivate {      constructor(private \_router:Router ) {    }      canActivate(route: ActivatedRouteSnapshot,                state: RouterStateSnapshot): boolean {        console.log("canActivate");      //return true       //remove comments to return true       alert('You are not allowed to view this page. You are redirected to Home Page');       this.\_router.navigate(["home"]);       return false;  } } |

First, we need to import the Router, CanActivate, ActivatedRouteSnapshot & RouterStateSnapshot library from the angular/core package

|  |  |
| --- | --- |
| 1  2  3 | import { Router, CanActivate, ActivatedRouteSnapshot,RouterStateSnapshot } from '@angular/router'; |

Define the ProductGuardService which implements the CanActivate Interface

|  |  |
| --- | --- |
| 1  2  3 | export class ProductGuardService implements CanActivate { |

Finally, define the canActivate method

The canActivate method accepts two arguments. The first argument is an ActivatedRouteSnapshot object, which describes the route that is being navigated to using the properties. The second argument is a RouterStateSnapshot object, which describes the current route through a single property called URL.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | canActivate(route: ActivatedRouteSnapshot,             state: RouterStateSnapshot): boolean {        console.log("canActivate");     //return true      //remove comments to return true      alert('You are not allowed to view this page. You are redirected to Home Page');      //this.\_router.navigate(["home"]); //navigate to some other route;      return false;  } |

Import the Guard in the Root Module

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule } from '@angular/forms';  import { HttpModule } from '@angular/http';  import { RouterModule } from '@angular/router';  import { AppComponent } from './app.component';  import { HomeComponent} from './home.component';  import { ContactComponent} from './contact.component';  import { ProductComponent} from './product.component';  import { ErrorComponent} from './error.component';  import { ProductDetailComponent} from './product-detail.component';  import { ProductService } from './product.service';  import { ProductGuardService } from './product-guard.service';  import { appRoutes } from './app.routes';    @NgModule({    declarations: [ AppComponent,HomeComponent,ContactComponent,                   ProductComponent,ErrorComponent,                   ProductDetailComponent],    imports:      [ BrowserModule, FormsModule,HttpModule,                   RouterModule.forRoot(appRoutes)],    providers:    [ ProductService,ProductGuardService ],    bootstrap:    [ AppComponent]  })  export class AppModule { } |

First Import the Guard Service as shown below

|  |  |
| --- | --- |
| 1  2  3 | import { ProductGuardService } from './product-guard.service'; |

Next, Register it using the Providers metadata, so that router can use it. Remember that Guards must be provided at the angular module level

|  |  |
| --- | --- |
| 1  2  3 | providers: [ProductService,ProductGuardService], |

Update the Routes

Finally, Update the app.routes class

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import { Routes } from '@angular/router';  import { HomeComponent} from './home.component';  import { ContactComponent} from './contact.component';  import { ProductComponent} from './product.component';  import { ErrorComponent} from './error.component';  import { ProductDetailComponent} from './product-detail.component';  import { ProductGuardService } from './product-guard.service';    export const appRoutes: Routes = [    { path: 'home', component: HomeComponent },    { path: 'contact', component: ContactComponent },    { path: 'product', component: ProductComponent, canActivate :[ProductGuardService] },    { path: 'product/:id', component: ProductDetailComponent },    { path: '', redirectTo: 'home', pathMatch: 'full' },    { path: '\*\*', component: ErrorComponent }  ]; |

The only change we have done is to attach the ProductGuardService to the CanActivate guard

|  |  |
| --- | --- |
| 1  2  3  4  5 | { path: 'product',    component: ProductComponent,    canActivate : [ProductGuardService] }, |

Test the Guard Run the app and you will see the alert message “You are not allowed to view this page. You are redirected to Home Page”

Summary

The angular Guards are a great tool, which helps us to protect the route. They also help us to run some logic, get data from the back end server, etc. You can also create multiple guards against a single route or use the same guard against multiple routes.

What is CanActivate Guard

The Angular CanActivate guard decides, if a route can be activated ( or component gets rendered). We use this guard, when we want to check on some condition, before activating the component or showing it to the user. This allows us to cancel the navigation.

Use cases for the CanActivate Guard

Checking if a user has logged in

Checking if a user has permission

One of the use case of this guard is to check if the user has logged in to the system. If user has not logged in, then the guard can redirect him to login page.

How to use CanActivate Guard

First, we need to create a [Angular Service](https://www.tektutorialshub.com/angular/angular-services/).

The service must import & implement the CanActivate Interface. This Interface is defined in the @angular/router module. The Interface has one method i.e. canActivate. We need to implement it in our Service. The details of the CanActivate interface is as shown below.

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

|  |  |
| --- | --- |
| 1  2  3  4  5 | interface CanActivate {    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree  } |

The method gets the instance of the ActivatedRouteSnapshot & RouterStateSnapshot. We can use this to get access to the route parameter, query parameter etc.

The guard must return true/false or a [UrlTree](https://angular.io/api/router/UrlTree) . The return value can be in the form of observable or a promise or a simple boolean value.

A route can have more than one canActivate guard.

If **all guards** returns true, navigation to the route will continue.

If **any one of the guard** returns false, navigation will be cancelled.

If**any one of the guard** returns a [UrlTree](https://angular.io/api/router/UrlTree), current navigation will be cancelled and a new navigation will be kicked off to the [UrlTree](https://angular.io/api/router/UrlTree) returned from the guard.

The example of canActivate guard is as follows

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import { Injectable } from '@angular/core';  import { Router, CanActivate, ActivatedRouteSnapshot,RouterStateSnapshot } from '@angular/router';      @Injectable()  export class AuthGuardService implements CanActivate {        constructor(private \_router:Router ) {      }        canActivate(route: ActivatedRouteSnapshot,                  state: RouterStateSnapshot): boolean {            //check some condition          if (someCondition)  {              alert('You are not allowed to view this page');              //redirect to login/home page etc              //return false to cancel the navigation              return false;          }          return true;      }    } |

Update the route definition with the canActivate guard as shown below. You can apply more than one guard to a route and a route can have more than one guard

|  |  |
| --- | --- |
| 1  2  3 | { path: 'product', component: ProductComponent, canActivate : [AuthGuardService] }, |

CanActivate guard Example

In our example application, we will create three components. The HomeComponent & ContactComponent are not protected and can be accessed any user. The user must log in into the system to access the ProductComponent..We also need a LoginComponent to handle the user login.

LoginComponent

The following is the code for LoginComponent and associated AuthService

login.component.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | import { Component, OnInit } from '@angular/core';  import { FormControl, FormGroup } from '@angular/forms';  import { Router, ActivatedRoute } from '@angular/router';  import { AuthService } from './auth.service';    @Component({     templateUrl: './login.component.html',     styles: [``]  })  export class LoginComponent implements OnInit {        invalidCredentialMsg: string;      username:string;      password:string;      retUrl:string="home";        constructor(private authService: AuthService,                  private router: Router,                  private activatedRoute:ActivatedRoute) {      }        ngOnInit() {          this.activatedRoute.queryParamMap                  .subscribe(params => {              this.retUrl = params.get('retUrl');              console.log( 'LoginComponent/ngOnInit '+ this.retUrl);          });      }        onFormSubmit(loginForm) {         this.authService.login(loginForm.value.username, loginForm.value.password).subscribe(data => {             console.log( 'return to '+ this.retUrl);             if (this.retUrl!=null) {                  this.router.navigate( [this.retUrl]);             } else {                  this.router.navigate( ['home']);             }         });      }  } |

login.component.html﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <h3>Login Form</h3>    <div>  <form #loginForm="ngForm" (ngSubmit)="onFormSubmit(loginForm)">    <p>User Name: <input type='text'  name='username' ngModel></p>    <p>Password: <input type="password"  name="password" ngModel></p>    <p><button type="submit">Submit</button></p>  </form>  </div> |

auth.service.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | import { Injectable } from '@angular/core';  import { Observable } from 'rxjs/Observable';  import 'rxjs/add/observable/of';  import 'rxjs/add/operator/map';  import { of } from 'rxjs';    @Injectable()  export class AuthService {        private isloggedIn: boolean;      private userName:string;        constructor() {          this.isloggedIn=false;      }        login(username: string, password:string) {            //Assuming users are provided the correct credentials.          //In real app you will query the database to verify.          this.isloggedIn=true;          this.userName=username;          return of(this.isloggedIn);      }        isUserLoggedIn(): boolean {          return this.isloggedIn;      }        isAdminUser():boolean {          if (this.userName=='Admin') {              return true;          }          return false;      }        logoutUser(): void{          this.isloggedIn = false;      }    } |

The AuthService checks whether the user is allowed to login. It has the method to login & logout the users. Our implementation of the login method does not check for anything. It just marks the user as logged in.

ProductComponent

The ProductComponent is our protected component. Only the logged in users can access this. This component displays the list of Products, which it gets from the ProductService.

product.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import { Component, OnInit } from '@angular/core';  import { ProductService } from './product.service';  import { Product } from './Product';      @Component({    templateUrl: "product.component.html",  })  export class ProductComponent  {       products:Product[];     constructor(private productService: ProductService){     }       ngOnInit() {          this.productService.getProducts()          .subscribe(data => {            this.products=data;          })     }    } |

product.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | <h1>Product List</h1>     <p> This is a protected component </p>       <div class='table-responsive'>     <table class='table'>         <thead>             <tr>                 <th>Name</th>                 <th>Price</th>             </tr>         </thead>         <tbody>             <tr \*ngFor="let product of products;">                 <td><a>{{product.name}} </a> </td>                 <td>{{product.price}}</td>             </tr>         </tbody>       </table>    </div> |

product.service.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | import {Product} from './Product'  import { of, Observable, throwError} from 'rxjs';  import { delay, map } from 'rxjs/internal/operators';    export class ProductService{        products: Product[];        public constructor() {          this.products=[              new Product(1,'Memory Card',500),              new Product(2,'Pen Drive',750),              new Product(3,'Power Bank',100),              new Product(4,'Computer',100),              new Product(5,'Laptop',100),              new Product(6,'Printer',100),          ]      }        public getProducts(): Observable<Product[]> {          return of(this.products) ;      }        public getProduct(id): Observable<Product> {          var Product= this.products.find(i => i.productID==id)          return of(Product) ;      }    } |

product.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | export class Product {        constructor(productID:number,    name: string ,   price:number) {          this.productID=productID;          this.name=name;          this.price=price;      }        productID:number ;      name: string ;      price:number;    } |

Other Component

app.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import { Component } from '@angular/core';  import { AuthService } from './auth.service';  import { Router } from '@angular/router';    @Component({    selector: 'app-root',    templateUrl: './app.component.html'  })  export class AppComponent {    title = 'Routing Module - Route Guards Demo';      constructor (private authService:AuthService,                 private router:Router) {    }      logout() {      this.authService.logoutUser();      this.router.navigate(['home']);    }    } |

app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | <div class="container">    <nav class="navbar navbar-default">    <div class="container-fluid">      <div class="navbar-header">        <a class="navbar-brand" [routerLink]="['/']"><strong> {{title}} </strong></a>      </div>      <ul class="nav navbar-nav">            <li><a [routerLink]="['home']">Home</a></li>          <li><a [routerLink]="['product']">Product</a></li>          <li><a [routerLink]="['contact']">Contact us</a></li>          <li><a [routerLink]="['login']">Login</a></li>          <li><a [routerLink]="" (click)="logout()">Log out</a></li>        </ul>    </div>  </nav>    <router-outlet></router-outlet>    </div> |

home.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import {Component} from '@angular/core';    @Component({      template: `<h1>Welcome!</h1>                <p>This is Home Component </p>               `  })    export class HomeComponent {  } |

contact.component.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import {Component} from '@angular/core';    @Component({       template: `<h1>Contact Us</h1>                  <p>TekTutorialsHub </p>                  `  })  export class ContactComponent {  } |

CanActivate Guard

Finally, we build a CanActivate guard, which will check whether the users are logged in or not. If users are not logged in, then they are redirected to the login page.

auth-guard.service.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | import { Injectable } from '@angular/core';  import { Router, CanActivate, ActivatedRouteSnapshot,RouterStateSnapshot, UrlTree } from '@angular/router';  import { AuthService } from './auth.service';      @Injectable()  export class AuthGuardService implements CanActivate {        constructor(private router:Router, private authService: AuthService ) {        }        canActivate(route: ActivatedRouteSnapshot,                  state: RouterStateSnapshot): boolean|UrlTree {            if (!this.authService.isUserLoggedIn()) {              alert('You are not allowed to view this page. You are redirected to login Page');                this.router.navigate(["login"],{ queryParams: { retUrl: route.url} });              return false;                //var urlTree = this.router.createUrlTree(['login']);              //return urlTree;          }            return true;      }    } |

First, we import the CanActivate from the @angular/router module.

The AuthGuardService implements the CanActivate interface

Inject the AuthServce in the constructor of the Guard

In the CanActivate method, we will redirect the user the login page, if the user is not logged in. To cancel the navigation ,we must either return false or urlTree as shown in the example above.

Next, we will update the route definition and use the guard in all the routes, which we want to protect

app.routes﻿﻿.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import { Routes } from '@angular/router';    import { HomeComponent} from './home.component'  import { ContactComponent} from './contact.component'  import { ProductComponent} from './product.component'    import { AuthGuardService } from './auth-guard.service';  import { LoginComponent } from './login.component';      export const appRoutes: Routes = [    { path: 'home', component: HomeComponent },    { path: 'login', component:LoginComponent},    { path: 'contact', component: ContactComponent },    { path: 'product', component: ProductComponent, canActivate : [AuthGuardService] },    { path: '', redirectTo: 'home', pathMatch: 'full' },  ]; |

Finally, register the service in the app.module.

app.module.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { HttpModule } from '@angular/http';  import { FormsModule }    from '@angular/forms';    import { RouterModule } from '@angular/router';    import { AppComponent } from './app.component';  import { HomeComponent} from './home.component'  import { ContactComponent} from './contact.component'  import { ProductComponent} from './product.component'    import { AuthGuardService } from './auth-guard.service';    import { appRoutes } from './app.routes';  import { AuthService } from './auth.service';  import { LoginComponent } from './login.component';  import { ProductService } from './product.service';    @NgModule({    declarations: [      AppComponent,HomeComponent,ContactComponent,ProductComponent,LoginComponent    ],    imports: [      BrowserModule,      FormsModule,      HttpModule,      RouterModule.forRoot(appRoutes)    ],    providers: [AuthGuardService,AuthService, ProductService],    bootstrap: [AppComponent]  })  export class AppModule { } |

What is CanActivateChild Guard

The CanActivatechild guard is very similar to CanActivateGuard. We apply this guard to the parent route. The Angular invokes this guard whenever the user tris to navigate to any of its child route. This allows us check some condition and decide whether to proceed with the navigation or cancel it.

Difference between CanActivate & CanActivateChild

Consider the following routes.

The ProductComponent displays the list of product. We have attached the canActivate guard to the product route. The canActivate guard blocks the access to the route, if the user is not logged in. This guard protect both the product route and all its children.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | { path: 'product', component: ProductComponent, canActivate : [AuthGuardService],        children: [        {  path: 'view/:id', component: ProductViewComponent  },        {  path: 'edit/:id', component: ProductEditComponent  },        {  path: 'add', component: ProductAddComponent }        ]    }, |

Now, consider the case where we want all users to view the ProductComponent, but only the Admin user can view any of its child routes

We can create another guard ProductGuardService which implements the canActivate guard and attach it to each of those child routes as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | { path: 'product', component: ProductComponent, canActivate : [AuthGuardService],        children: [        {  path: 'view/:id', component: ProductViewComponent, canActivate : [ProductGuardService]  },        {  path: 'edit/:id', component: ProductEditComponent, canActivate : [ProductGuardService]  },        {  path: 'add', component: ProductAddComponent, canActivate : [ProductGuardService] }        ]    }, |

Another way is to use the CanActivateChild guard and attach it to the product route as shown below. When Angular sees a canActivateChild guard attached to the parent route, it invokes it every time the user tries to navigates to the child route. Hence instead of attaching Guard service every child, you can attach it to the parent route.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | { path: 'product', component: ProductComponent, canActivate : [AuthGuardService],  canActivateChild : [AdminGuardService],        children: [        {  path: 'view/:id', component: ProductViewComponent  },        {  path: 'edit/:id', component: ProductEditComponent  },        {  path: 'add', component: ProductAddComponent }        ]    }, |

How to Create CanActivateChild Guard

Just like all other [Angular Guards](https://www.tektutorialshub.com/angular/angular-route-guards/), we need to create an [Angular Service](https://www.tektutorialshub.com/angular/angular-services/). The service must import & implement the CanActivateChild Interface. The Interface is defined in the @angular/router module. The Interface has one method i.e. canActivateChild. The details of the CanActivateChild interface are as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5 | interface CanActivateChild {    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree  } |

The method gets the instance of the ActivatedRouteSnapshot & RouterStateSnapshot. We can use this to get access to the route parameter, query parameter, etc.

The guard must return true/false or a [UrlTree](https://angular.io/api/router/UrlTree) . It can return these value either as a observable or a promise or as a simple Boolean value.

If all guards return true, navigation will continue.

If any guard returns false, navigation will be cancelled.

If any guard returns a [UrlTree](https://angular.io/api/router/UrlTree), current navigation will be cancelled and a new navigation will be kicked off to the [UrlTree](https://angular.io/api/router/UrlTree) returned from the guard.

The example canActivateChild guard is as follows

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import { Injectable } from '@angular/core';  import { Router, CanActivate, ActivatedRouteSnapshot,RouterStateSnapshot } from '@angular/router';      @Injectable()  export class AuthGuardService implements CanActivateChild {        constructor(private \_router:Router ) {      }        canActivate(route: ActivatedRouteSnapshot,                  state: RouterStateSnapshot): boolean {            //check some condition          if (someCondition)  {              alert('You are not allowed to view this page');              //redirect to login/home page etc              //return false to cancel the navigation              return false;          }          return true;      }    } |

Angular CanActivateChild Example

In our example application, the HomeComponent & ContactComponent are not protected and can be accessed any user.

The user must log in into the system to access the ProductComponent..

The ProdudctEditComponent, ProductViewComponets are child components of the ProductComponent

We also need a LoginComponent to handle the user login.

login.component.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | import { Component, OnInit } from '@angular/core';  import { FormControl, FormGroup } from '@angular/forms';  import { Router, ActivatedRoute } from '@angular/router';  import { AuthService } from './auth.service';    @Component({     templateUrl: './login.component.html',     styles: [``]  })  export class LoginComponent implements OnInit {        invalidCredentialMsg: string;      username:string;      password:string;      retUrl:string="home";        constructor(private authService: AuthService,                  private router: Router,                  private activatedRoute:ActivatedRoute) {      }        ngOnInit() {          this.activatedRoute.queryParamMap                  .subscribe(params => {              this.retUrl = params.get('retUrl');              console.log( 'LoginComponent/ngOnInit '+ this.retUrl);          });      }        onFormSubmit(loginForm) {         this.authService.login(loginForm.value.username, loginForm.value.password).subscribe(data => {             console.log( 'return to '+ this.retUrl);             if (this.retUrl!=null) {                  this.router.navigate( [this.retUrl]);             } else {                  this.router.navigate( ['home']);             }         });      }  } |

login.component.html﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <h3>Login Form</h3>    <div>  <form #loginForm="ngForm" (ngSubmit)="onFormSubmit(loginForm)">    <p>User Name: <input type='text'  name='username' ngModel></p>    <p>Password: <input type="password"  name="password" ngModel></p>    <p><button type="submit">Submit</button></p>  </form>  </div> |

auth.service.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | import { Injectable } from '@angular/core';  import { Observable } from 'rxjs/Observable';  import 'rxjs/add/observable/of';  import 'rxjs/add/operator/map';  import { of } from 'rxjs';      @Injectable()  export class AuthService {        private isloggedIn: boolean;      private userName:string;        constructor() {          this.isloggedIn=false;      }        login(username: string, password:string) {            //Assuming users are provided the correct credentials.          //In real app you will query the database to verify.          this.isloggedIn=true;          this.userName=username;          return of(this.isloggedIn);      }        isUserLoggedIn(): boolean {          return this.isloggedIn;      }        isAdminUser():boolean {          if (this.userName=='Admin') {              return true;          }          return false;      }        logoutUser(): void{          this.isloggedIn = false;      }    } |

The AuthService checks whether the user is allowed to login. It has the method to login & logout the users. Our implementation of the login method does not check for anything. It just marks the user as logged in. isAdminUser() method returns true if the user is logged in with the user name “Admin”.

Product Component

The ProductComponent is our protected component. Only the logged in users can access this. This component displays the list of Products, which it gets from the ProductService.

product.component.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import { Component, OnInit } from '@angular/core';  import { ProductService } from './product.service';  import { Product } from './Product';      @Component({    templateUrl: "product.component.html",  })  export class ProductComponent  {       products:Product[];     constructor(private productService: ProductService){     }       ngOnInit() {          this.productService.getProducts()          .subscribe(data => {            this.products=data;          })     }    } |

product.component.html﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | <h1>Product List</h1>     <p> This is a protected component </p>       <div class='table-responsive'>     <table class='table'>         <thead>             <tr>                 <th>Name</th>                 <th>Price</th>                 <th></th>                 <th></th>             </tr>         </thead>         <tbody>             <tr \*ngFor="let product of products;">                 <td><a [routerLink]="['/product/view',product.productID]">{{product.name}}</a></td>                 <td>{{product.price}}</td>                 <td><a [routerLink]="['/product/view',product.productID]">View</a></td>                 <td><a [routerLink]="['/product/edit',product.productID]">Edit</a></td>             </tr>         </tbody>       </table>    </div>      <router-outlet></router-outlet> |

product.service.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | import {Product} from './Product'  import { of, Observable, throwError} from 'rxjs';  import { delay, map } from 'rxjs/internal/operators';    export class ProductService{        products: Product[];        public constructor() {          this.products=[              new Product(1,'Memory Card',500),              new Product(2,'Pen Drive',750),              new Product(3,'Power Bank',100),              new Product(4,'Computer',100),              new Product(5,'Laptop',100),              new Product(6,'Printer',100),          ]      }        public getProducts(): Observable<Product[]> {          return of(this.products) ;      }        public getProduct(id): Observable<Product> {          var Product= this.products.find(i => i.productID==id)          return of(Product);      }    } |

product.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | export class Product {        constructor(productID:number,    name: string ,   price:number) {          this.productID=productID;          this.name=name;          this.price=price;      }        productID:number ;      name: string ;      price:number;    } |

We have Product add, edit & view components.

product-add.component.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import { Component, OnInit } from '@angular/core';  import { Product } from './Product';  import { ProductService } from './product.service';  import { ActivatedRoute } from '@angular/router';        @Component({    template: `<h1>Add Product</h1>`,  })  export class ProductAddComponent  {      product:Product;      constructor(private productService: ProductService,                private route:ActivatedRoute ){    }        ngOnInit() {    }    } |

product-edit.component.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | import { Component, OnInit } from '@angular/core';  import { Product } from './Product';  import { ProductService } from './product.service';  import { ActivatedRoute } from '@angular/router';      @Component({    template: `<h1>Edit Product</h1>`,  })  export class ProductEditComponent  {        product:Product        constructor(private productService: ProductService,                  private route:ActivatedRoute ){      }          ngOnInit() {        let id=this.route.snapshot.params['id'];        this.productService.getProduct(id)          .subscribe(data => {              this.product=data;          })      }  } |

product-view.component.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | import { Component, OnInit } from '@angular/core';  import { Product } from './Product';  import { ProductService } from './product.service';  import { ActivatedRoute } from '@angular/router';      @Component({    template: `<h1>View Product</h1>`,  })  export class ProductViewComponent  {        product:Product        constructor(private productService: ProductService,                  private route:ActivatedRoute ){      }          ngOnInit() {              let id=this.route.snapshot.params['id'];            this.productService.getProduct(id)              .subscribe(data => {                  this.product=data;              })      }    } |

Other Components

app.component.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | import { Component } from '@angular/core';  import { AuthService } from './auth.service';  import { Router } from '@angular/router';    @Component({    selector: 'app-root',    templateUrl: './app.component.html'  })  export class AppComponent {    title = 'Routing Module - Route Guards Demo';      constructor (private authService:AuthService,                 private router:Router) {    }      logout() {      this.authService.logoutUser();      this.router.navigate(['home']);    }    } |

app.component.html﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | <div class="container">    <nav class="navbar navbar-default">    <div class="container-fluid">      <div class="navbar-header">        <a class="navbar-brand" [routerLink]="['/']"><strong> {{title}} </strong></a>      </div>      <ul class="nav navbar-nav">            <li><a [routerLink]="['home']">Home</a></li>          <li><a [routerLink]="['product']">Product</a></li>          <li><a [routerLink]="['contact']">Contact us</a></li>          <li><a [routerLink]="['login']">Login</a></li>          <li><a [routerLink]="" (click)="logout()">Log out</a></li>        </ul>    </div>  </nav>    <router-outlet></router-outlet>    </div> |

home.component.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import {Component} from '@angular/core';    @Component({      template: `<h1>Welcome!</h1>                <p>This is Home Component </p>               `  })    export class HomeComponent {  } |

contact.component.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import {Component} from '@angular/core';    @Component({       template: `<h1>Contact Us</h1>                  <p>TekTutorialsHub </p>                  `  })  export class ContactComponent {  } |

CanActivateChild Guard

Next, we will build CanActivate guard, which will check whether the users are logged in or not. The users are redirected to the login page, if they are not logged in.

Also, CanActivateChild guard, which checks whether the user is Admin User. Only the Admin Users are allowed to navigate to the ProductEditComponent & ProductViewComponent.

auth-guard.service.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | import { Injectable } from '@angular/core';  import { Router, CanActivate, ActivatedRouteSnapshot,RouterStateSnapshot, UrlTree, CanActivateChild } from '@angular/router';  import { AuthService } from './auth.service';      @Injectable()  export class AuthGuardService implements CanActivate , CanActivateChild {        constructor(private router:Router, private authService: AuthService ) {        }        canActivate(route: ActivatedRouteSnapshot,                  state: RouterStateSnapshot): boolean|UrlTree {            console.log('canActivate on '+route.url);            if (!this.authService.isUserLoggedIn()) {              alert('You are not allowed to view this page. You are redirected to login Page');              this.router.navigate(["login"],{ queryParams: { retUrl: route.url } });              return false;                //var urlTree = this.router.createUrlTree(['login']);              //return urlTree;          }            return true;      }        canActivateChild(route: ActivatedRouteSnapshot,                      state: RouterStateSnapshot): boolean|UrlTree {                if (!this.authService.isAdminUser()) {              alert('You are not allowed to view this page');              return false;          }              return true;  }    } |

First, we import the CanActivate and CanActivateChild from the @angular/router module.

The AuthGuardService implements both CanActivate & CanActivateChild interface

Inject the AuthServce in the constructor of the Guard

In the CanActivate method, we will redirect the user the login page, if the user is not logged in. To cancel the navigation ,we must either return false or urlTree as shown in the example above.

While in CanActivateChild method, we just check if the user is Admin, the we will return true else return false

Next, we will update the route definition and use the guard in all the routes, which we want to protect

app.routes﻿﻿.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | import { Routes } from '@angular/router';    import { HomeComponent} from './home.component'  import { ContactComponent} from './contact.component'  import { ProductComponent } from './product.component'    import { AuthGuardService } from './auth-guard.service';  import { LoginComponent } from './login.component';  import { ProductViewComponent } from './product-view.component';  import { ProductAddComponent } from './product-add.component';  import { ProductEditComponent } from './product-edit.component';      export const appRoutes: Routes = [    { path: 'home', component: HomeComponent },    { path: 'login', component:LoginComponent},    { path: 'contact', component: ContactComponent },    { path: 'product', component: ProductComponent, canActivate : [AuthGuardService] ,        canActivateChild : [AuthGuardService],        children: [        {  path: 'view/:id', component: ProductViewComponent  },        {  path: 'edit/:id', component: ProductEditComponent  },        {  path: 'add', component: ProductAddComponent }        ]    },      { path: '', redirectTo: 'home', pathMatch: 'full' },  ]; |

app.module.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { HttpModule } from '@angular/http';  import { FormsModule }    from '@angular/forms';    import { RouterModule } from '@angular/router';    import { AppComponent } from './app.component';  import { HomeComponent} from './home.component'  import { ContactComponent} from './contact.component'  import { ProductComponent} from './product.component'    import { AuthGuardService } from './auth-guard.service';    import { appRoutes } from './app.routes';  import { AuthService } from './auth.service';  import { LoginComponent } from './login.component';  import { ProductAddComponent } from './product-add.component';  import { ProductViewComponent } from './product-view.component';  import { ProductEditComponent } from './product-edit.component';  import { ProductService } from './product.service';    @NgModule({    declarations: [      AppComponent,HomeComponent,ContactComponent,ProductComponent,LoginComponent, ProductAddComponent, ProductViewComponent, ProductEditComponent    ],    imports: [      BrowserModule,      FormsModule,      HttpModule,      RouterModule.forRoot(appRoutes)    ],    providers: [AuthGuardService,AuthService,ProductService],    bootstrap: [AppComponent]  })  export class AppModule { } |

We apply both the guards on the product route. The canActivate guards protects the product route and canActivateChild protects all it child routes.

What is CanDeactivate Guard

The Angular CanDeactivate guard is called, whenever we navigate away from the route before the current component gets deactivated.

The best use case for CanDectivate guard is the data entry component. The user may have filled the data entry and tries to leave that component without saving his work. The CanDeactivate guard gives us a chance to warn the user that he has not saved his work and give him a chance to cancel the navigation.

How to use CanDeactivate Guard

First, we need to create an [Angular Service](https://www.tektutorialshub.com/angular/angular-services/).

The service must import & implement the CanDeactivate Interface. This Interface is defined in the @angular/router module. The Interface has one method i.e. canDeactivate. We need to implement it in our Service. The details of the CanDeactivate interface is as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | interface CanDeactivate<T> {    canDeactivate(component: T,                  currentRoute: ActivatedRouteSnapshot,                  currentState: RouterStateSnapshot,                  nextState?: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree  } |

The method gets the instance of the component being deactivated as the first argument.

The method also gets the instance of the ActivatedRouteSnapshot, the current state of the  RouterStateSnapshot, next state of the RouterStateSnapshot. We can use this to get access to the route parameter, query parameter, etc.

The guard must return **true/false** or a [UrlTree](https://angular.io/api/router/UrlTree) . The return value can be in the form of **observable** or a **promise** or a simple boolean value.

A route can have more than one canDeactivate guard.

If **all guards** return true, the component will get deactivated and you will navigate to the next route.

If **anyone of the guard returns false, navigation will be canceled** and you will stay in the same route/component.

If**any one of the guards** returns a UrlTree, current navigation will be canceled and new navigation will be kicked off to the [UrlTree](https://angular.io/api/router/UrlTree) returned from the guard. The current component will get deactivated.

The example of canDeactivate the guard is as follows

CanDeactivate Example

First create a component, with a method canExit, which returns a boolean. In the canExit method, you can check if there are any unsaved data, etc. If Yes, then ask for confirmation whether the user wants to leave or not. Return true to exit the component else return false to stay in the same component.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import { Component } from '@angular/core';    @Component({    templateUrl: "register.component.html",  })  export class RegisterComponent       //Check if there any unsaved data etc. If yes then as for confirmation    canExit() : boolean {      if (confirm("Do you wish to Please confirm")) {        return true      } else {        return false      }    }    } |

Next, create a guard service, which implements the CanDeactivate Interface.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | import { Injectable } from '@angular/core';  import { CanDeactivate } from '@angular/router/src/utils/preactivation';  import { ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';  import { Observable } from 'rxjs';  import { RegisterComponent } from './register.component';    @Injectable()  export class DeactivateGuard implements CanDeactivate  {      component: Object;      route: ActivatedRouteSnapshot;       constructor(){     }       canDeactivate(component:RegisterComponent,                  route: ActivatedRouteSnapshot,                  state: RouterStateSnapshot,                  nextState: RouterStateSnapshot) : Observable<boolean> | Promise<boolean> | boolean {            return component.canExit();      }    } |

Update the route definition with the canDeactivate guard as shown below. You can apply more than one guard to a route and a route can have more than one guard

|  |  |
| --- | --- |
| 1  2  3 | { path: 'register', component: RegisterComponent, canDeactivate:[DeactivateGuard] }, |

The guard created in the above example is specific to RegisterComponent. We can create a general-purpose DeactivateGuard Service and use it everywhere, by making use of the interface

General Purpose DeactivateGuard Service

First Define an Interface IDeactivateComponent as shown below

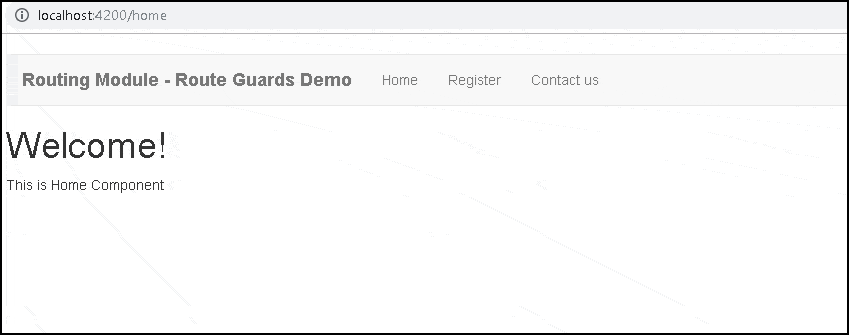
|  |  |
| --- | --- |
| 1  2  3  4  5 | export interface IDeactivateComponent {      canExit: () => Observable<boolean> | Promise<boolean> | boolean;  } |

Next, Open the component, where you wish to implement the Deactivate Guard. In our case it is RegisterComponent. Make the component implement the IDeactivateComponent as shown below. Implement the canExit method

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import { Component } from '@angular/core';  import { IDeactivateComponent } from './decativate.guard';      @Component({    templateUrl: "register.component.html",  })  export class RegisterComponent implements IDeactivateComponent  {       //Check if there any unsaved data etc. If yes then as for confirmation    canExit() : boolean {      if (confirm("Do you wish to Please confirm")) {        return true      } else {        return false      }    }    } |

Now, in the guard service, use the IDeactivateComponent interface instead of the actual component.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | @Injectable()  export class DeactivateGuard implements CanDeactivate  {      component: Object;      route: ActivatedRouteSnapshot;       constructor(){     }       canDeactivate(component:IDeactivateComponent,                  route: ActivatedRouteSnapshot,                  state: RouterStateSnapshot,                  nextState: RouterStateSnapshot) : Observable<boolean> | Promise<boolean> | boolean {            return component.canExit ? component.canExit() : true;    }    } |



Complete Source Code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | <div class="container">    <nav class="navbar navbar-default">    <div class="container-fluid">      <div class="navbar-header">        <a class="navbar-brand" [routerLink]="['/']"><strong> {{title}} </strong></a>      </div>      <ul class="nav navbar-nav">            <li><a [routerLink]="['home']">Home</a></li>          <li><a [routerLink]="['register']">Register</a></li>          <li><a [routerLink]="['contact']">Contact us</a></li>        </ul>    </div>  </nav>    <router-outlet></router-outlet>    </div> |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import { Component } from '@angular/core';  import { Router } from '@angular/router';    @Component({    selector: 'app-root',    templateUrl: './app.component.html'  })  export class AppComponent {    title = 'Routing Module - Route Guards Demo';      constructor (private router:Router) {    }      } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { HttpModule } from '@angular/http';  import { FormsModule }    from '@angular/forms';    import { RouterModule } from '@angular/router';    import { AppComponent } from './app.component';  import { HomeComponent} from './home.component'  import { ContactComponent} from './contact.component'        import { appRoutes } from './app.routes';    import { RegisterComponent } from './register.component';  import { DeactivateGuard } from './decativate.guard';    @NgModule({    declarations: [      AppComponent,HomeComponent,ContactComponent,RegisterComponent    ],    imports: [      BrowserModule,      FormsModule,      HttpModule,      RouterModule.forRoot(appRoutes)    ],    providers: [DeactivateGuard],    bootstrap: [AppComponent]  })  export class AppModule { } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import { Routes } from '@angular/router';    import { HomeComponent} from './home.component'  import { ContactComponent} from './contact.component'      import { RegisterComponent } from './register.component';  import { DeactivateGuard } from './decativate.guard';      export const appRoutes: Routes = [    { path: 'home', component: HomeComponent },    { path: 'contact', component: ContactComponent },    { path: 'register', component: RegisterComponent, canDeactivate:[DeactivateGuard] },    { path: '', redirectTo: 'home', pathMatch: 'full' },  ]; |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | import { Injectable } from '@angular/core';  import { CanDeactivate } from '@angular/router/src/utils/preactivation';  import { ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';  import { Observable } from 'rxjs';  import { RegisterComponent } from './register.component';    export interface IDeactivateComponent {      canExit: () => Observable<boolean> | Promise<boolean> | boolean;  }    @Injectable()  export class DeactivateGuard implements CanDeactivate  {      component: Object;      route: ActivatedRouteSnapshot;       constructor(){     }       canDeactivate(component:IDeactivateComponent,                  route: ActivatedRouteSnapshot,                  state: RouterStateSnapshot,                  nextState: RouterStateSnapshot) : Observable<boolean> | Promise<boolean> | boolean {            return component.canExit ? component.canExit() : true;    }      } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import {Component} from '@angular/core';    @Component({       template: `<h1>Contact Us</h1>                  <p>TekTutorialsHub </p>                  `  })  export class ContactComponent {  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import {Component} from '@angular/core';    @Component({      template: `<h1>Welcome!</h1>                <p>This is Home Component </p>               `  })    export class HomeComponent {  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | import { Component } from '@angular/core';  import { IDeactivateComponent } from './decativate.guard';      @Component({    template: `<h1>Register User</h1>            <p> </p>`,  })  export class RegisterComponent    implements IDeactivateComponent  {       //Check if there any unsaved data etc. If yes then as for confirmation    canExit() : boolean {      if (confirm("Do you wish to Please confirm")) {        return true      } else {        return false      }    }    } |

Summary

In this tutorial, we learned how to make use of CanDeactivate Guard. This guard is used to stop the user from accidentally leaving the route, before saving their work.

Angular Resolve Guard

The Angular renders the [Angular Component](https://www.tektutorialshub.com/angular/angular-component/), when we navigate to a route. The component will then sends an [HTTP request](https://www.tektutorialshub.com/angular/angular-httpclient/) to back end server to fetch data so as to display it to the user. We generally do this in [ngOnInit Life cycle hook](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/)

The Problem with the above approach is that, the use will see a empty component. The component shows the data after the arrival of the data. The one way is solve this problem is to show some loading indicator

The another way to solve this is to make use the Resolve Guard. The Resolve Guard pre fetches the data before the navigating to the route. Hence the component is rendered along with the data.

How to Use Resolve Guard

First, we need to create a [Angular Service](https://www.tektutorialshub.com/angular/angular-services/), which implements the Resolve Interface

The service must implement the resolve method. A resolve method must return either a Observable<any>, Promise<any> or just data. The interface signature of the Resolve interface is as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5 | interface Resolve<T> {    resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<T> | Promise<T> | T  } |

Inside the Resolve method, we will get the access to the ActivatedRouteSnapshot & RouterStateSnapshot, which can be used to get the values of router parameter, query parameters etc.

The following is the simple example of a Angular Resolver.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import { Injectable } from '@angular/core';  import { Resolve, ActivatedRouteSnapshot,RouterStateSnapshot } from '@angular/router';  import { ProductService } from './product.service';  import { Observable, of } from 'rxjs';    @Injectable()  export class ProductListResolveService implements Resolve<any>{        constructor(private \_router:Router , private       productService:ProductService ) {      }        resolve(route: ActivatedRouteSnapshot,              state: RouterStateSnapshot): Observable<any> {            return this.productService.getProducts();      }  } |

First, we import the Resolve from the @angular/router module.

We can make use of [Angular Dependency Injection](https://www.tektutorialshub.com/angular/angular-dependency-injection/) to inject the required services in the constructor.

The resolve method must return either an Observable<any>,  Promise<any> or just **data**. In the example above, we are invoking the getProducts method of method of the productService, which returns an observable.

The router does not create the instance of the resolver. The job falls on the [Angular dependency injection](https://www.tektutorialshub.com/angular/angular-dependency-injection/). Hence, we need it to register it in the [Providers](https://www.tektutorialshub.com/angular/angular-providers/) array of root module.

|  |  |
| --- | --- |
| 1  2  3 | providers: [ProductListResolveGuardService] |

Once the resolver is created, we need to update the route definition and add resolve property as shown in below.

|  |  |
| --- | --- |
| 1  2  3 | { path: 'product', component: ProductComponent, resolve: {products: ProductListResolveService}  }, |

The resolve property is a JavaScript object of key value pair of **resolvers**. The **key** is user defined variable. The **value** must be a resolver service.

In the example above products is the key and ProductListResolveService is the resolver. The return value of the ProductListResolveService is assigned to the key i.e products and made available to the component via **route data**

When the user navigates to the route product, the angular looks for the resolve property of the route. **For each key value pair of resolvers**, the angular calls the resolve method. If the return value of the resolver is a observable or a promise, the router will wait for that to complete. The returned value is assigned to the key products and added to the route data collection.

The component can just read the **products** from the route data from the ActivatedRoute as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | constructor(private route: ActivatedRoute){     }      ngOnInit() {        this.products=this.route.snapshot.data['products'];     } |

Remember Resolve runs after all other guards are executed

Cancelling Navigation

If you return null from the resolver, the router will cancel the navigation

If the error is generated, then the router will cancel the navigation.

Multiple Resolvers

You can define more than one resolver

|  |  |
| --- | --- |
| 1  2  3  4 | { path: 'product', component: ProductComponent,      resolve: {products: ProductListResolveService, , data:SomeOtherResolverService}  } |

Resolve Guard Example

Let us build a simple app to demonstrate the use of Resolve guard.

Product Service

The following is the simple ProductService, which retrieves the hard coded products.

product.service.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | import {Product} from './Product'  import { of, Observable, throwError} from 'rxjs';  import { delay, map } from 'rxjs/internal/operators';    export class ProductService{        products: Product[];        public constructor() {          this.products=[              new Product(1,'Memory Card',500),              new Product(2,'Pen Drive',750),              new Product(3,'Power Bank',100),              new Product(4,'Computer',100),              new Product(5,'Laptop',100),              new Product(6,'Printer',100),          ]      }        //Return Products List with a delay      public getProducts(): Observable<Product[]> {          return of(this.products).pipe(delay(1500)) ;      }        // Returning Error      // This wil stop the route from getting Activated      //public getProducts(): Observable<Product[]> {      //    return of(this.products).pipe(delay(1500), map( data => {      //        throw throwError("errors occurred") ;      //    }))      //}        public getProduct(id): Observable<Product> {          var Product= this.products.find(i => i.productID==id)          return of(Product).pipe(delay(1500)) ;      }  } |

product.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | export class Product {      constructor(productID:number,    name: string ,   price:number)     {          this.productID=productID;          this.name=name;          this.price=price;      }        productID:number ;      name: string ;      price:number;    } |

The getProducts() method returns the Observable of products using the RxJS operator of. We have included a delay of 1500 ms.

Similarly, the getProduct(id) returns the observable of Product after a delay of 1500 ms.

Product Component without resolver

Next, let us build a two components to display the list of Products. The first component Product1Component does not use resolver. The second component Product1Component makes use of the resolver.

This component, subscribes to the getProducts() method of the ProductService to get the list of Products

product1.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import { Component, OnInit } from '@angular/core';    import { ProductService } from './product.service';  import { Product } from './product';  import { ActivatedRoute } from '@angular/router';    @Component({    templateUrl: './product1.component.html',  })    export class Product1Component  {     public products:Product[];       constructor(private route: ActivatedRoute,private productService:ProductService){     }       ngOnInit() {         this.productService.getProducts().subscribe(data => {           this.products=data;        });     }    } |

Resolve Guard

The ProductListResolverService service implements the Resolve Interface. It just returns the productService.getProducts().

product.service.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import { Injectable } from '@angular/core';  import { Resolve, ActivatedRouteSnapshot,RouterStateSnapshot } from '@angular/router';  import { ProductService } from './product.service';  import { Observable, of } from 'rxjs';  import { Product } from './Product';      @Injectable()  export class ProductListResolverService implements Resolve<Product>{        constructor(private productService:ProductService ) {      }        resolve(route: ActivatedRouteSnapshot,              state: RouterStateSnapshot): Observable<any> {            console.log("ProductListResover is called");          return this.productService.getProducts();      }    } |

Product Component with resolver

The following is the Product2Component does not use the ProductService, but gets the product list from the Route data.

product-list-resolver.service.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import { Component, OnInit } from '@angular/core';    import { ProductService } from './product.service';  import { Product } from './product';  import { ActivatedRoute } from '@angular/router';    @Component({    templateUrl: './product2.component.html',  })    export class Product2Component  {       public products:Product[];       constructor(private route: ActivatedRoute,private productService:ProductService){     }       ngOnInit() {        this.products=this.route.snapshot.data['products'];     }    } |

In the app.routing.module, we need to add the resolve guard in the route definition

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | export const appRoutes: Routes = [    { path: 'home', component: HomeComponent },    { path: 'contact', component: ContactComponent },    { path: 'product1', component: Product1Component },    { path: 'product2', component: Product2Component, resolve: {products: ProductListResolverService}  }  ] |

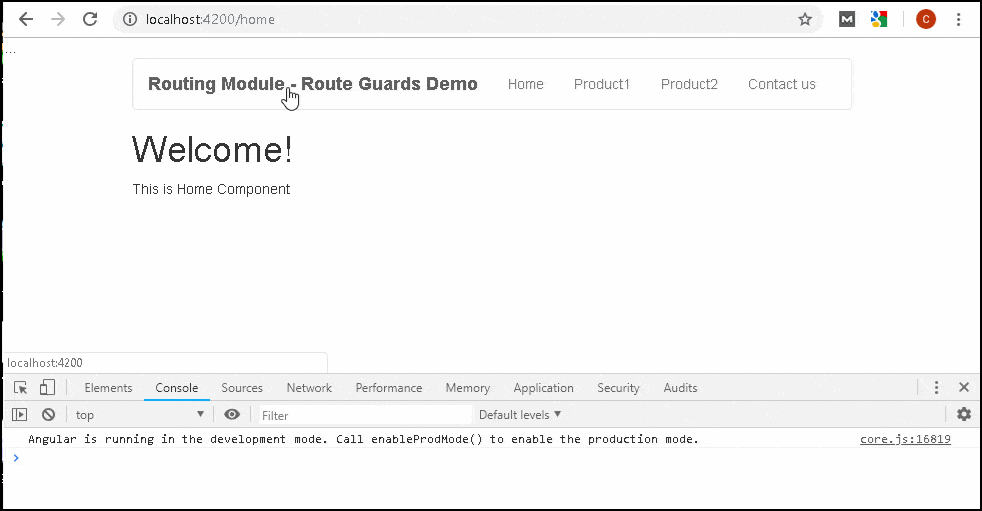
The register the ProductListResolverService in Providers array in AppModule

|  |  |
| --- | --- |
| 1  2  3 | providers: [ProductService,ProductListResolverService], |

Finally, run the app

As you click on the **Product1** link, you will see that the component gets loaded, but the data appears after a delay.

While you click on **Product2** the component itself appears after some delay. That is because it waits for the resolver to finish. The Component renders along with the data.



ProductDetail Component

We can continue and create Product2DetailComponent and create a resolver, which if product not found redirects us to the Product2Component.

product-resolver.service.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | import { Injectable } from '@angular/core';  import { Router, Resolve, ActivatedRouteSnapshot,RouterStateSnapshot } from '@angular/router';  import { ProductService } from './product.service';  import { Observable, of } from 'rxjs';  import { catchError, map } from 'rxjs/internal/operators';  import { Product } from './Product';      @Injectable()  export class ProductResolverService implements Resolve<any>{        constructor(private router:Router , private productService:ProductService ) {      }        resolve(route: ActivatedRouteSnapshot,             state: RouterStateSnapshot): any {        let id = route.paramMap.get('id');      console.log("ProductResolverService  called with "+id);      return this.productService.getProduct(id)          .pipe(map( data => {              if (data) {                  console.log(data);                  return data;              } else {                  console.log('redirecting');                  this.router.navigate(['/product2']);                  return null              }          }))      }  } |

product2-detail.component.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import { Component } from '@angular/core';  import { ActivatedRoute } from '@angular/router';    import { Product } from './product';    @Component({    templateUrl: './product2-detail.component.html',  })    export class Product2DetailComponent  {     product:Product;       constructor(private \_Activatedroute:ActivatedRoute){        this.product=this.\_Activatedroute.snapshot.data['product'];     }    } |

In the resolver service, we check to see if the product exists, if not we redirect the use the product list page and return null.

Complete Example

app.module.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule, ErrorHandler } from '@angular/core';  import { FormsModule } from '@angular/forms';  import { HttpModule } from '@angular/http';    import { RouterModule } from '@angular/router';    import { AppComponent } from './app.component';  import { HomeComponent} from './home.component'  import { ContactComponent} from './contact.component'    import { ProductService } from './product.service';  import { ProductListResolverService } from './product-list-resolver.service';  import { ProductResolverService } from './product-resolver.service';    import { AppRoutingModule } from './app-routing.module';    import { Product1Component } from './product1.component';  import { Product1DetailComponent} from './product1-detail.component'    import { Product2Component } from './product2.component';    import { Product2DetailComponent } from './product2-detail.component';    @NgModule({    declarations: [      AppComponent,HomeComponent,ContactComponent,Product1Component, Product2Component,Product1DetailComponent, Product2DetailComponent    ],    imports: [      BrowserModule,      FormsModule,      HttpModule,      RouterModule,      AppRoutingModule    ],    providers: [ProductService,ProductListResolverService,ProductResolverService,],    bootstrap: [AppComponent]  })  export class AppModule { } |

app.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'Routing Module - Route Guards Demo';  } |

app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <div class="container">  <nav class="navbar navbar-default">    <div class="container-fluid">      <div class="navbar-header">        <a class="navbar-brand" [routerLink]="['/']"><strong> {{title}} </strong></a>      </div>      <ul class="nav navbar-nav">          <li><a [routerLink]="['home']">Home</a></li>          <li><a [routerLink]="['product1']">Product1</a></li>          <li><a [routerLink]="['product2']">Product2</a></li>          <li><a [routerLink]="['contact']">Contact us</a></li>      </ul>    </div>  </nav>  <router-outlet></router-outlet>  </div> |

app.routing.module.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | import { NgModule} from '@angular/core';  import { Routes,RouterModule } from '@angular/router';    import { HomeComponent} from './home.component'  import { ContactComponent} from './contact.component'    import { Product1Component} from './product1.component'  import { Product2Component} from './product2.component'    import { Product1DetailComponent} from './product1-detail.component'  import { ProductListResolverService } from './product-list-resolver.service';  import { Product2DetailComponent } from './product2-detail.component';  import { ProductResolverService } from './product-resolver.service';      export const appRoutes: Routes = [    { path: 'home', component: HomeComponent },    { path: 'contact', component: ContactComponent },    { path: 'product1', component: Product1Component },    { path: 'product2', component: Product2Component, resolve: {products: ProductListResolverService}  },    { path: 'product1/:id', component: Product1DetailComponent },    { path: 'product2/:id', component: Product2DetailComponent, resolve:{product:ProductResolverService} },    { path: '', redirectTo: 'home', pathMatch: 'full' },  ];    @NgModule({    declarations: [    ],    imports: [      RouterModule.forRoot(appRoutes)    ],    providers: [],    bootstrap: []  })  export class AppRoutingModule { } |

contact.component.ts﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import {Component} from '@angular/core';    @Component({       template: `<h1>Contact Us</h1>                  <p>TekTutorialsHub </p>                  `  })  export class ContactComponent {  } |

home.components.ts﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import {Component} from '@angular/core';    @Component({      template: `<h1>Welcome!</h1>                <p>This is Home Component </p>               `  })    export class HomeComponent {  } |

product.ts﻿﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | export class Product {        constructor(productID:number,    name: string ,   price:number)      {          this.productID=productID;          this.name=name;          this.price=price;      }        productID:number ;      name: string ;      price:number;    } |

product.service.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | import {Product} from './Product'  import { of, Observable, throwError} from 'rxjs';  import { delay, map } from 'rxjs/internal/operators';    export class ProductService{        products: Product[];        public constructor() {          this.products=[              new Product(1,'Memory Card',500),              new Product(2,'Pen Drive',750),              new Product(3,'Power Bank',100),              new Product(4,'Computer',100),              new Product(5,'Laptop',100),              new Product(6,'Printer',100),          ]      }        //Return Products List with a delay      public getProducts(): Observable<Product[]> {          return of(this.products).pipe(delay(1500)) ;      }          // Returning Error      // This wil stop the route from getting Activated      //public getProducts(): Observable<any> {      //    return of(this.products).pipe(delay(3000), map( data => {      //        throw throwError("errors occurred") ;      //    }))      //}        public getProduct(id): Observable<Product> {          var Product= this.products.find(i => i.productID==id)          return of(Product).pipe(delay(1500)) ;      }  } |

product-list-resolver.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import { Injectable } from '@angular/core';  import { Router, Resolve, ActivatedRouteSnapshot,RouterStateSnapshot } from '@angular/router';  import { ProductService } from './product.service';  import { Observable, of } from 'rxjs';    @Injectable()  export class ProductListResolverService implements Resolve<any>{        constructor(private productService:ProductService ) {      }        resolve(route: ActivatedRouteSnapshot,              state: RouterStateSnapshot): Observable<any> {          console.log("ProductListResover is called");          return this.productService.getProducts();      }  } |

product-resolver.service.ts﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | import { Injectable } from '@angular/core';  import { Router, Resolve, ActivatedRouteSnapshot,RouterStateSnapshot } from '@angular/router';  import { ProductService } from './product.service';  import { Observable, of } from 'rxjs';  import { catchError, map } from 'rxjs/internal/operators';  import { Product } from './Product';    @Injectable()  export class ProductResolverService implements Resolve<any>{        constructor(private router:Router , private productService:ProductService ) {      }        resolve1(route: ActivatedRouteSnapshot,              state: RouterStateSnapshot): Observable<any> {            console.log("ProductResolverService  called");          let id = route.paramMap.get('id');          return this.productService.getProduct(id);      }        resolve(route: ActivatedRouteSnapshot,             state: RouterStateSnapshot): any {        let id = route.paramMap.get('id');      console.log("ProductResolverService  called with "+id);      return this.productService.getProduct(id)          .pipe(map( data => {              if (data) {                  console.log(data);                  return data;              } else {                  console.log('redirecting');                  this.router.navigate(['/product2']);                  return null              }          }))  }  } |

product1.component.ts﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | import { Component, OnInit } from '@angular/core';    import { ProductService } from './product.service';  import { Product } from './product';  import { ActivatedRoute } from '@angular/router';    @Component({    templateUrl: './product1.component.html',  })    export class Product1Component  {       public products:Product[];       constructor(private route: ActivatedRoute,private productService:ProductService){     }       ngOnInit() {        console.log('ngOnInit');          this.productService.getProducts().subscribe(data => {           this.products=data;        });     }  } |

product1.component.html﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | <h1> Without Resolve</h1>  <div class='table-responsive'>      <table class='table'>          <thead>              <tr>                  <th>Name</th>                  <th>Price</th>              </tr>          </thead>          <tbody>              <tr \*ngFor="let product of products;">                  <td><a [routerLink]="['/product1',product.productID]">{{product.name}} </a> </td>                  <td>{{product.price}}</td>              </tr>          </tbody>      </table>  </div> |

product1-detail.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | import { Component } from '@angular/core';  import { Router,ActivatedRoute } from '@angular/router';    import { ProductService } from './product.service';  import { Product } from './product';    @Component({    templateUrl: './product1-detail.component.html',  })    export class Product1DetailComponent  {     product:Product;       constructor(private \_Activatedroute:ActivatedRoute,                 private \_router:Router,                 private \_productService:ProductService){          let id=this.\_Activatedroute.snapshot.params['id'];        console.log(id);        this.\_productService.getProduct(id)           .subscribe( data => {              this.product=data              console.log(this.product);           })     }  } |

product1-detail.component.html﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <h1>Product Details Page [Without resolve]</h1>    Product : {{ product?.name}}  <br>  Price : {{product?.price}} |

product2.component.ts﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import { Component, OnInit } from '@angular/core';    import { ProductService } from './product.service';  import { Product } from './product';  import { ActivatedRoute } from '@angular/router';    @Component({    templateUrl: './product2.component.html',  })  export class Product2Component  {     public products:Product[];       constructor(private route: ActivatedRoute,private productService:ProductService){     }       ngOnInit() {        this.products=this.route.snapshot.data['products'];     }    } |

product2.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | <h1> With Resolve</h1>  <div class='table-responsive'>      <table class='table'>          <thead>              <tr>                  <th>Name</th>                  <th>Price</th>              </tr>          </thead>          <tbody>              <tr \*ngFor="let product of products;">                  <td><a [routerLink]="['/product2',product.productID]">{{product.name}} </a> </td>                  <td>{{product.price}}</td>              </tr>          </tbody>      </table>  </div> |

product2-detail.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | import { Component } from '@angular/core';  import { ActivatedRoute } from '@angular/router';    import { Product } from './product';    @Component({    templateUrl: './product2-detail.component.html',  })  export class Product2DetailComponent  {     product:Product;     constructor(private \_Activatedroute:ActivatedRoute){        this.product=this.\_Activatedroute.snapshot.data['product'];     }  } |

product2-detail.component.html﻿﻿

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <h1>Product Details Page [With resolve]</h1>    Product : {{ product?.name}}  <br>  Price : {{product?.price}} |

Passing static data to a route

We can configure the static data at the time of defining the route. This is done by using the **Angular route data property** of the [route](https://www.tektutorialshub.com/angular/angular-routing-navigation/#route). The route data property can contain an array of arbitrary string key-value pairs. You can use the static data to store items such as page titles, breadcrumb text, and other read-only, static data

For Example, consider the following route with the data property set

|  |  |
| --- | --- |
| 1  2  3 | { path: 'static', component: StaticComponent, data :{ id:'1', name:"Angular"}}, |

The [Angular Router](https://www.tektutorialshub.com/angular/angular-routing-navigation/) will pass the { id:'1', name:"Angular"} when the StaticComponent is rendered. The data value will be located in the data property of the ActivatedRoute service

We can then read the data by subscribing to the activatedroute.data property as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | ngOnInit() {        this.activatedroute.data.subscribe(data => {            this.product=data;        })  } |

Passing Dynamic data to a Route

The option to pass the dynamic data or a user-defined object was added in the **Angular Version 7.2** using the state object. The state object is stored in [History API](https://developer.mozilla.org/en-US/docs/Web/API/History)

Providing the State value

The state can be provided in two ways

Using routerLink directive

|  |  |
| --- | --- |
| 1  2  3 | <a [routerLink]="['dynamic']" [state]="{ id:1 , name:'Angular'}">Dynamic Data</a> |

Using navigateByUrl

|  |  |
| --- | --- |
| 1  2  3 | this.router.navigateByUrl('/dynamic', { state: { id:1 , name:'Angular' } }); |

The Router will add a navigationId property to the state object. Hence you cannot use a scalar value.

Accessing the state value

The state can be accessed by using the getCurrentNavigation method of the router (works only in the constructor)

|  |  |
| --- | --- |
| 1  2  3 | this.router.getCurrentNavigation().extras.state |

Or use the history.state in the [ngOnInit](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/).

|  |  |
| --- | --- |
| 1  2  3 | console.log(history.state) |

or use the getState method of the Location Service. This method is available Angular 8+

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import { Location } from '@angular/common';    export class SomeComponent  {    products:Product[];      constructor(private location:Location){    }      ngOnInit() {      console.log(this.location.getState());    }  } |

Passing Data to the Routes Example

Let us build a simple project to demonstrate how to pass data to the route

Passing static data example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import {Component, OnInit} from '@angular/core';  import { ActivatedRoute } from '@angular/router';    @Component({       template: `<h1>Passing Static Data Demo</h1>           {{product  | json}}`  })  export class StaticComponent implements OnInit {         product:any;       constructor(private activatedroute:ActivatedRoute) {       }         ngOnInit() {            this.activatedroute.data.subscribe(data => {                 this.product=data;             })       }  } |

[***Source Code***](https://stackblitz.com/edit/angular-pass-data-to-route-1)

The static component gets the static data configured in the route. It subscribes the activatedroute.data property to get the product data as shown above.

Passing dynamic data (or object) example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import {Component, OnInit, ChangeDetectorRef} from '@angular/core';  import { ActivatedRoute, Router, NavigationStart } from '@angular/router';  import { map, filter} from 'rxjs/operators';  import { Observable} from 'rxjs/observable';    @Component({       template: `<H1>Passing Dynamic Data Demo</H1>         {{ product | json }}`  })  export class DynamicComponent implements OnInit {         product;         constructor(private router:Router, private activatedRoute:ActivatedRoute) {            console.log(this.router.getCurrentNavigation().extras.state);       }         ngOnInit() {            //console.log(history.state);            this.product=history.state;       }    } |

[***Source Code***](https://stackblitz.com/edit/angular-pass-data-to-route-1)

The Dynamic Component gets dynamic data. We use the history.state to access the product data. Alternatively, we can use the this.router.getCurrentNavigation().extras.state to achieve the same. Please remember getCurrentNavigation only works in the constructor. It will return null if used elsewhere.

**home.component.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | import { Component } from '@angular/core';  import { ActivatedRoute, Router } from '@angular/router';    @Component({    template: `       <ul>         <li><a [routerLink]="['/static']">Static Data</a></li>         <li><a [routerLink]="['/dynamic']" [state]=product>Dynamic Data</a></li>      </ul>        <p>Id :   <input type="text" [(ngModel)]="product.id" > </p>      <p>name :<input type="text" [(ngModel)]="product.name" > </p>      <button (click)="gotoDynamic()" >Goto Dynamic Component</button>`  })  export class HomeComponent {      public product = { id:'1', name:"Angular"};      constructor(private router : Router) {    }      gotoDynamic() {      //this.router.navigateByUrl('/dynamic', { state: { id:1 , name:'Angular' } });      this.router.navigateByUrl('/dynamic', { state: this.product });    }  } |

[***Source Code***](https://stackblitz.com/edit/angular-pass-data-to-route-1)

In HomeComponent, we have used routerLink & navigateByUrl to pass the data to the dynamic component. You can also use the form fields to change the data, before passing it to the dynamic route.

**app.routes.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import { Routes } from '@angular/router';    import { StaticComponent} from './static.component'  import { DynamicComponent } from './dynamic.component';  import { HomeComponent } from './home.component';    export const appRoutes: Routes = [    { path: 'home', component: HomeComponent },    { path: 'static', component: StaticComponent, data :{ id:'1', name:"Angular"}},    { path: 'dynamic', component: DynamicComponent },    { path: '', redirectTo: 'home', pathMatch: 'full' }  ]; |

[***Source Code***](https://stackblitz.com/edit/angular-pass-data-to-route-1)

Here the static data is set for StaticComponent using the data property.

**app.component.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    template: `<div class="container">      <nav class="navbar navbar-default">      <div class="container-fluid">        <div class="navbar-header">          <a class="navbar-brand" [routerLink]="['/']"><strong> {{title}} </strong></a>        </div>      </div>    </nav>      <router-outlet></router-outlet>      </div>`  })  export class AppComponent {    title = 'Routing Module - Passing Dynamic / Static data route';  } |

[***Source Code***](https://stackblitz.com/edit/angular-pass-data-to-route-1)

**app.module.ts**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule } from '@angular/forms';  import { HttpModule } from '@angular/http';    import { RouterModule } from '@angular/router';    import { AppComponent } from './app.component';  import { StaticComponent} from './static.component'      import { appRoutes } from './app.routes';  import { DynamicComponent } from './dynamic.component';  import { HomeComponent } from './home.component';    @NgModule({    declarations: [      AppComponent,StaticComponent,DynamicComponent,HomeComponent    ],    imports: [      BrowserModule,      FormsModule,      HttpModule,      RouterModule.forRoot(appRoutes)    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

RouterLinkActive

The RouterLinkActive Directive is applied along with the RouterLink directive. The right-hand side of RouterLinkActive contains a **Template expression**. The template expression must contain a space-delimited string of CSS classes, which will be applied to the element when the route is active.

For Example, consider the following examples

|  |  |
| --- | --- |
| 1  2  3  4  5 | <li><a [routerLink]="['home']" routerLinkActive="active">Home</a></li>    <li><a [routerLink]="['product']" [routerLinkActive]="['active']">Product</a></li> |

With the CSS Rule defined as the global styles.

|  |  |
| --- | --- |
| 1  2  3  4  5 | .active {    background-color: yellow;  } |

When the user navigates to any of the above [routes](https://www.tektutorialshub.com/angular/angular-routing-navigation/#routes), the Angular router adds the “active” class to the activated element. And when the user navigates away the class will be removed.

The Angular does this by watching the URL. Whenever the Url matches with the URL of the routerLink directive, it applies the classes defined in the RouterLinkActive directive. When it does not match it will be removed from the element.

Using this we can apply different background or foreground color to our navigation links.

Multiple classes

You can add multiple classes to the routerLinkActive directive as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <a routerLink="/user/bob" [routerLinkActive]="['class1', 'class2']">Bob</a>    <li><a [routerLink]="['home']" routerLinkActive="active  home">Home</a></li>  <li><a [routerLink]="['product']" [routerLinkActive]="['active','home']">Product</a></li> |

Child Routes

When the child route is active, then all the parent routes are also marked as active and routerLinkActive is applied to URL tree cascading all the way to the top.

For Example

|  |  |
| --- | --- |
| 1  2  3  4  5 | <a routerLink="/product" routerLinkActive="class1 class2">Product</a>  <a routerLink="/product/PC" routerLinkActive="class1 class2">PC</a>  <a routerLink="/product/mobile" routerLinkActive="class1 class2">Mobile</a> |

In the above example. When the URL is either /product/PC or /product/mobile, the RouterLinkactive class (i.e. class1 class2) is added to the /product element also as it is the parent of these child routes.

Exact matching

You can stop that from happening by, passing the exact: true to the RouterLinkactive using the routerLinkActiveOptions as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <a routerLink="/product" routerLinkActive="class1 class1" [routerLinkActiveOptions]="{exact:  true}">Product</a>  <a routerLink="/product/PC" routerLinkActive="class1 class2">PC</a>  <a routerLink="/product/mobile" routerLinkActive="class1 class2">Mobile</a> |

This will make the RouterLinkactive to be applied only if the route URL matches exactly to the current URL

Matching

The routerActiveLink follows the following criteria before returning **true**

without exact: true

A subset of the [queryParams](https://www.tektutorialshub.com/angular/angular-passing-optional-query-parameters-to-route/) is matched.

The URL is a subtree of the URL tree.

Matrix params are ignored

with exact: true

A [queryParams](https://www.tektutorialshub.com/angular/angular-passing-optional-query-parameters-to-route/) must match exactly

The URL must match exactly

Matrix params are ignored

Adding classes to ancestors

You can apply the RouterLinkActive directive to an ancestor of a RouterLink.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <div routerLinkActive="active" [routerLinkActiveOptions]="{exact: true}">    <a routerLink="/product/pc">Jim</a>    <a routerLink="/product/mobile">Bob</a>  </div> |

This will set the active class on the div tag if the URL is either /product/pc or product/mobile

Bind to Component Property

You can also bind RouterLinkActive to a component property which returns a string of classes using template expression.

|  |  |
| --- | --- |
| 1  2  3 | <li><a [routerLink]="['home']" routerLinkActive="{{getClass()}}">Home</a></li> |

In component class

|  |  |
| --- | --- |
| 1  2  3  4  5 | getClass() {    return "active"  } |

Router Events

The [Angular Routers](https://www.tektutorialshub.com/angular/angular-routing-navigation/) triggers several events starting with when the Navigation starts ( NavigationStart ) and also when the Navigation end ( NavigationEnd ) successfully. It is triggered when the navigation is canceled either by the user ( NavigationCancel ) or due to an error in the navigation ( NavigationError). The Events trigger when the lazy loaded modules are about to load and when they finish loading. They trigger before and after the guards like [canActivate](https://www.tektutorialshub.com/angular/angular-canactivate-guard-example/), [canActivateChild](https://www.tektutorialshub.com/angular/angular-canactivatechild-example/). Events fire before and after the Angular runs the [Route Resolvers](https://www.tektutorialshub.com/angular/angular-resolve-guard/)..

The following are the list of events that are fire by the Angular Router.

| **Router Event** | **The Event triggered when** |
| --- | --- |
| NavigationStart | the Angular router stats the navigation. |
| RouteConfigLoadStart | the Router lazy loads a route configuration. |
| RouteConfigLoadEnd | after a route has been lazy-loaded. |
| RoutesRecognized | the Router parses the URL and the routes are recognized. |
| GuardsCheckStart | the Router begins the Guards phase of routing. |
| ChildActivationStart | the Router begins activating a route's children. |
| ActivationStart | the Router begins activating a route. |
| GuardsCheckEnd | the Router finishes the Guards phase of routing successfully. |
| ResolveStart | the Router begins the Resolve phase of routing. |
| ResolveEnd | the Router finishes the Resolve phase of routing successfully. |
| ChildActivationEnd | the Router finishes activating a route's children. |
| ActivationEnd | the Router finishes activating a route. |
| NavigationEnd | navigation ends successfully. |
| NavigationCancel | navigation is canceled. This is due to a Route Guard returning false during navigation. |
| NavigationError | navigation fails due to an unexpected error. |
| Scroll | An event that represents a scrolling event. |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

How to Listen to Router Events

First, we need to import the event, which we want to listen to. The following example imports NavigationStart. We also need to import the NavigationEvent & router.

|  |  |
| --- | --- |
| 1  2  3  4  5 | import { Router, NavigationStart, Event as NavigationEvent } from '@angular/router'; |

And inject the Router in the constructor

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | constructor(private router:Router) {  } |

And finally in you can listen to events by subscribing to the router.events observable.

The router.events is an Observable that gets triggered when the route changes. We receive NavigationEvent as the parameter in the callback. We check the NavigationEvent instance to check the Type of event fired.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | this.router.events    .subscribe(      (event: NavigationEvent) => {        if(event instanceof NavigationStart) {          console.log(event);        }      }); |

Or you can make use of the rxjs filter operator to filter out the required events as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import { filter } from 'rxjs/operators';  import { Router, NavigationStart, Event as NavigationEvent } from '@angular/router';    this.router.events    .pipe(      filter( event =>event instanceof NavigationStart)    )    .subscribe(      (event: NavigationEvent) => {        console.log(event);    }  ) |

Summary

The Router events allow us to watch for the router state changes and run some custom logic. One of the use case scenarios is to show the loading indicator when the user navigates from one route to another. You can listen to NavigationStart and NavigationEnd events to achieve this result

# Paste HERE