ANGULAR

4.4 Components :

4.4.1 How to Create and Use Components in Angular?

In Angular, components are essential building blocks for developing an application. In simple words, an Angular component is a piece of your application's UI that is controlled by the Angular framework. Components might include things like the header, footer, or the entire page, depending on how reusable your code needs to be.

Every Angular component is made up of three parts:

1. **Template -** Every Angular component has an associated HTML template that defines a view to be presented in a target environment.
2. **Class -** Just like in any other programming languages, in Angular, a class acts as a template for creating an object. Each Angular component contains a TypeScript class that holds application data and logic. A class is created with a @Component() decorator. A Decorator is a function that allows to add metadata to a class, method, accessor, property, or argument. A decorator is used with the form @expression, where expression is the decorator's name.
3. **Metadata -** This is the additional information about a class. The metadata helps to determine if a class is a component class or a regular class.

Root Component in Angular

In every Angular application, there is at least one root component that connects a component hierarchy to the page document object model (DOM). The following files make up the root component of your Angular application, which can be found in the `src/app/` directory:

- **app-routing.module.ts** - This file contains routing configuration. The purpose of this file is to load and configure the router in a separate file for readability. The module class name of this file is AppRoutingModule by convention and it is imported by the root `AppModule` in `app.module.ts` file.
- **app.component.css** - This is the component view's stylesheet file. Every component has one or more associated style-sheet files. This file can also be in other formats: CSS, Sass, Stylus, or Less.
- **app.component.html** - This is the component view's HTML template file.
- **app.component.spec.ts** - This is the component's test file which is also known as a spec file. The file extension `.spec.ts` helps tools to recognize it as a file containing the tests.
- **app.component.ts** - This file contains all of the component's code for controlling its functionality.
- **app.module.ts** - Every Angular application contains at least one NgModule class, the root module, which is commonly referred to as AppModule and is in the `src/app/app.module.ts` file. This root NgModule is to define Angular modules as well as to startup your application.

How to Create a new Component in Angular?

Use the following command to create a new component in your Angular project:

```
ng generate component user-detail
```

The above command creates a directory src/app/user-detail and inside that director the following four files are generated:

- A TypeScript class file with a component class named UserDetailComponent.
- An HTML template file for the component.
- A CSS file for styling the component.
- A test file for UserDetailComponent.

Creating and Using Components in Angular Example

In this example, we will learn how to create components in an Angular application from scratch. Follow the steps below to complete this example:

1. Lets start by creating a new Angular project using the Angular CLI ng new project-name command as shown in the example below:

```
ng new my-sample-app
```

2. When you are prompted with **Would you like to add Angular routing?** Choose Yes by pressing Y.
3. When you see **Which stylesheet format would you like to use?** Choose CSS and press Enter.
4. After the project gets created, navigate to the project folder:

```
cd my-sample-app
```

5. Run the project by executing the following command:

```
ng serve --open
```

6. If your installation and setup was successful, you should see a default welcome page like the following on your browser, running at http://localhost:4200/:
7. Now, lets create a navbar component that we can re-use in many views. Open a new terminal and navigate to your project root directory and execute the following command to create a navbar component:

```
ng generate component navbar
```

8. The above command will create the following component files in src/app/navbar/ directory and will also register **NavbarComponent** to the src/app/app.module.ts file:
   - navbar.component.css
   - navbar.component.html
   - navbar.component.spec.ts
   - navbar.component.ts

   If you go to src/app/app.module.ts file of your Angular project, you will see entry of **NavbarComponent** as shown in the example below:

   src/app/app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
//Enry of NavbarComponent
import { NavbarComponent } from './navbar/navbar.component';

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

9. Now, go to src/app/navbar/navbar.components.ts file and find the value of selector. Here, the value of selector is **app-navbar** as shown in the example below:src/app/navbar/navbar.components.ts

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

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

  constructor() { }

  ngOnInit(): void {
  }

      }
```

10. Go to the root src/app/app.component.html file and remove everything except this line <router-outlet></router-outlet>. Copy the value of selector from the newly created src/app/navbar/navbar.components.ts file to src/app/app.component.html. Your root src/app/app.component.html file should look like this:

```
<app-navbar></app-navbar>
```

```
        <router-outlet></router-outlet>
```

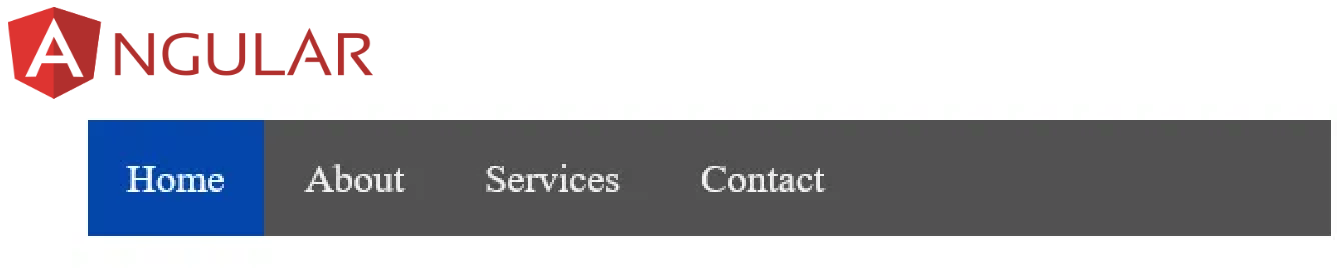
11. Now, go to navbar component HTML view file at src/app/navbar/navbar.component.html and add the HTML code to display navbar to it as shown in the example below:

```
<div class="navbar">
    <a class="active" href="#">Home</a>
    <a href="#">About</a>
    <a href="#">Services</a>
    <a href="#">Contact</a>
      </div>
```

12. Next, add the following CSS to the style-sheet file of navbar component at src/app/navbar/navbar.component.css:

```
.navbar {
  background-color: rgb(82, 81, 81);
  overflow: hidden;
}

.navbar a {
  color: #f2f2f2;
  text-align: center;
  float: left;
  text-decoration: none;
  font-size: 16px;
  padding: 14px 16px;
}

.navbar a:hover {
  background-color: #f2f2f2;
  color: rgb(46, 45, 45);
}

.navbar a.active {
  background-color: #0446aa;
  color: white;
      }
```

13. Go to your browser and check your application page at http://localhost:4200/. The resulting page should be something like this:



Congratulations! you have learned how to create and use a component in an Angular application.

4.4.2 Components

Components are building block of Angular application. The main job of Angular Component is to generate a section of web page called **view**. Every component will have an associated template and it will be used to generate views.

Let us learn the basic concept of component and template in this chapter.

Add a component

Let us create a new component in our **ExpenseManager** application.

Open command prompt and go to **ExpenseManager** application.

```
cd /go/to/expense-manager
```

Create a new component using **ng generate component** command as specified below −

```
ng generate component expense-entry
```

Output

The output is mentioned below −

```
CREATE src/app/expense-entry/expense-entry.component.html (28 bytes)
CREATE src/app/expense-entry/expense-entry.component.spec.ts (671 bytes)
CREATE src/app/expense-entry/expense-entry.component.ts (296 bytes)
CREATE src/app/expense-entry/expense-entry.component.css (0 bytes)
UPDATE src/app/app.module.ts (431 bytes)
```

Here,

- **ExpenseEntryComponent** is created under src/app/expense-entry folder.
- Component class, Template and stylesheet are created.
- AppModule is updated with new component.

Add title property to **ExpenseEntryComponent** (src/app/expense-entry/expense-entry.component.ts) component.

```
import { Component, OnInit } from '@angular/core'; @Component({
  selector: 'app-expense-entry',
  templateUrl: './expense-entry.component.html', styleUrls: ['./expense-entry.component.css']
})
export class ExpenseEntryComponent implements OnInit {
  title: string;
  constructor() { }
  ngOnInit() {
    this.title = "Expense Entry"
  }
}
```

Update template, **src/app/expense-entry/expense-entry.component.html**with below content.

```
<p>{{ title }}</p>
```

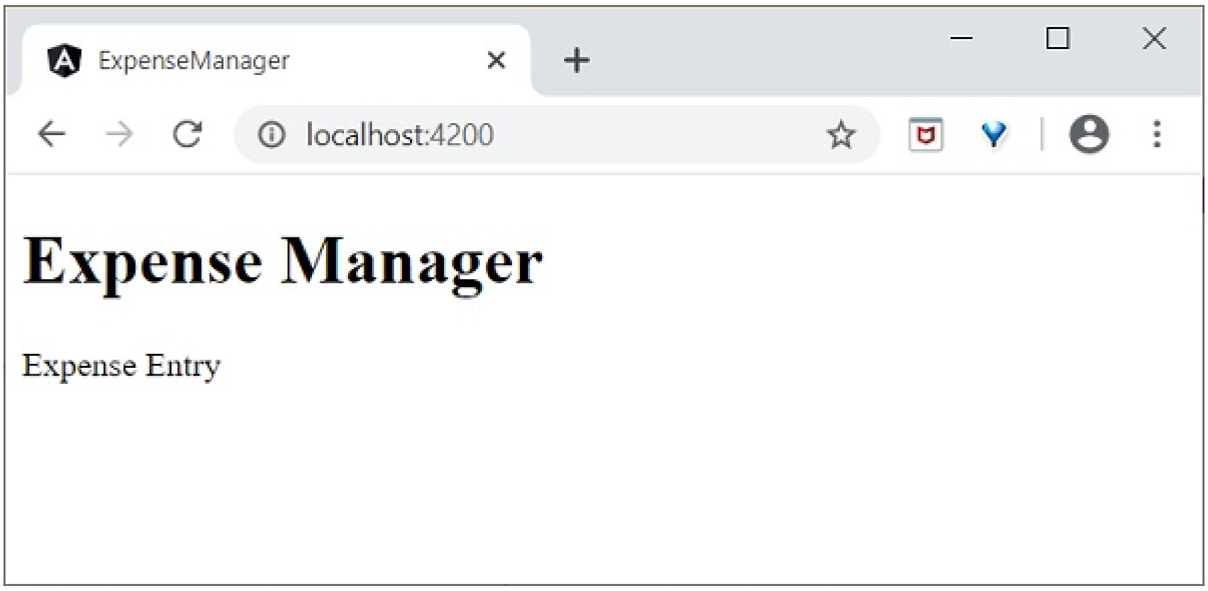
Open **src/app/app.component.html** and include newly created component.

```
<h1>{{ title }}</h1>
<app-expense-entry></app-expense-entry>
```

Here,

app-expense-entry is the selector value and it can be used as regular HTML Tag.

Finally, the output of the application is as shown below –



Expense Manager

Expense Entry

We will update the content of the component during the course of learning more about templates.

Templates

The integral part of Angular component is **Template**. It is used to generate the HTML content. **Templates** are plain HTML with additional functionality.

Attach a template

Template can be attached to Angular component using **@component** decorator's meta data. Angular provides two meta data to attach template to components.

templateUrl

We already know how to use templateUrl. It expects the relative path of the template file. For example, AppComponent set its template as app.component.html.

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

template

template enables to place the HTML string inside the component itself. If the template content is minimal, then it will be easy to have it **Component** class itself for easy tracking and maintenance purpose.

```
@Component({
  selector: 'app-root',
  templateUrl: `<h1>{{ title }}</h1>`,
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  title = 'Expense Manager';
  constructor(private debugService : DebugService) {} ngOnInit() {
    this.debugService.info("Angular Application starts");
  }
}
```

Attach Stylesheet

Angular Templates can use CSS styles similar to HTML. Template gets its style information from two sources, a) from its component b) from application configuration.

Component configuration

Component decorator provides two option, **styles** and **styleUrls** to provide CSS style information to its template.

- Styles − **styles** option is used to place the CSS inside the component itself.

styles: ['h1 { color: '#ff0000'; }']

- styleUrls − **styleUrls** is used to refer external CSS stylesheet. We can use multiple stylesheet as well.

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

Application configuration

Angular provides an option in project configuration **(angular.json)** to specify the CSS stylesheets. The styles specified in **angular.json** will be applicable for all templates. Let us check our **angular.json** as shown below −

```
{
"projects": {
  "expense-manager": {
    "architect": {
      "build": {
        "builder": "@angular-devkit/build-angular:browser", "options": {
          "outputPath": "dist/expense-manager",
          "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": []
        },
      },
    }
  }},
  "defaultProject": "expense-manager"
}
```

Here,

styles option sets**src/styles.css** as global CSS stylesheet. We can include any number of CSS stylesheets as it supports multiple values.

Include bootstrap

Let us include bootstrap into our **ExpenseManager** application using **styles** option and change the default template to use bootstrap components.

Open command prompt and go to ExpenseManager application.

```
cd /go/to/expense-manager
```

Install **bootstrap** and **JQuery** library using below commands

```
npm install --save bootstrap@4.5.0 jquery@3.5.1
```

Here,

We have installed JQuery, because, bootstrap uses jquery extensively for advanced components.

Option **angular.json** and set bootstrap and jquery library path.

```
{
  "projects": {
    "expense-manager": {
      "architect": {
        "build": {
          "builder":"@angular-devkit/build-angular:browser", "options": {
            "outputPath": "dist/expense-manager",
            "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": [
              "./node_modules/bootstrap/dist/css/bootstrap.css", "src/styles.css"
            ],
            "scripts": [
              "./node_modules/jquery/dist/jquery.js", "./node_modules/bootstrap/dist/js/bootstrap.js"
            ]
          },
        },
      }
  }},
  "defaultProject": "expense-manager"
}
```

Here,

scripts option is used to include JavaScript library. **JavaScript** registered through **scripts** will be available to all Angular components in the application.

Open **app.component.html** and change the content as specified below

```
<!-- Navigation -->
<nav class="navbar navbar-expand-lg navbar-dark bg-dark static-top">
  <div class="container">
       <a class="navbar-brand" href="#">{{ title }}</a> <button class="navbar-toggler" type="button"
data-toggle="collapse" data-target="#navbarResponsive" aria-controls="navbarResponsive" aria-
expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon">
      </span>
    </button>
    <div class="collapse navbar-collapse" id="navbarResponsive">
      <ul class="navbar-nav ml-auto">
        <li class="nav-item active">
        <a class="nav-link" href="#">Home
          <span class="sr-only">(current)
          </span>
        </a>
        </li>
        <li class="nav-item">
        <a class="nav-link" href="#">Report</a>
        </li>
        <li class="nav-item">
        <a class="nav-link" href="#">Add Expense</a>
        </li>
        <li class="nav-item">
        <a class="nav-link" href="#">About</a>
        </li>
      </ul>
    </div>
  </div>
</nav>
<app-expense-entry></app-expense-entry>
```

Here,

Used bootstrap navigation and containers.

Open **src/app/expense-entry/expense-entry.component.html** and place below content.

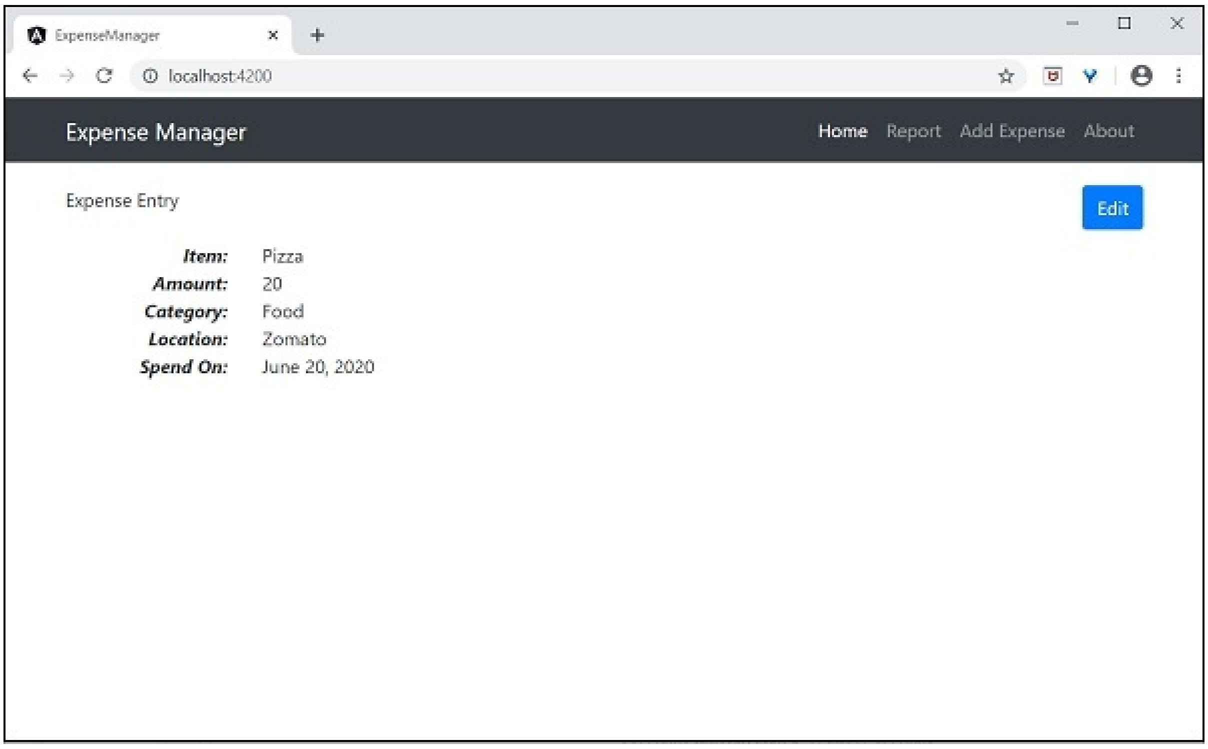
```
<!-- Page Content -->
<div class="container">
  <div class="row">
```

```
    <div class="col-lg-12 text-center" style="padding-top: 20px;">
      <div class="container" style="padding-left: 0px; padding-right: 0px;">
        <div class="row">
        <div class="col-sm" style="text-align: left;"> {{ title }}
        </div>
        <div class="col-sm" style="text-align: right;">
          <button type="button" class="btn btn-primary">Edit</button>
        </div>
        </div>
      </div>
      <div class="container box" style="margin-top: 10px;">
      <div class="row">
      <div class="col-2" style="text-align: right;">
        <strong><em>Item:</em></strong>
      </div>
      <div class="col" style="text-align: left;">
        Pizza
      </div>
      </div>
      <div class="row">
      <div class="col-2" style="text-align: right;">
        <strong><em>Amount:</em></strong>
      </div>
      <div class="col" style="text-align: left;">
        20
      </div>
      </div>
      <div class="row">
      <div class="col-2" style="text-align: right;">
        <strong><em>Category:</em></strong>
      </div>
      <div class="col" style="text-align: left;">
        Food
      </div>
      </div>
      <div class="row">
      <div class="col-2" style="text-align: right;">
        <strong><em>Location:</em></strong>
      </div>
      <div class="col" style="text-align: left;">
        Zomato
      </div>
      </div>
      <div class="row">
      <div class="col-2" style="text-align: right;">
        <strong><em>Spend On:</em></strong>
      </div>
      <div class="col" style="text-align: left;">
        June 20, 2020
      </div>
      </div>
    </div>
  </div>
```

```
</div>
</div>
```

Restart the application.

The output of the application is as follows –



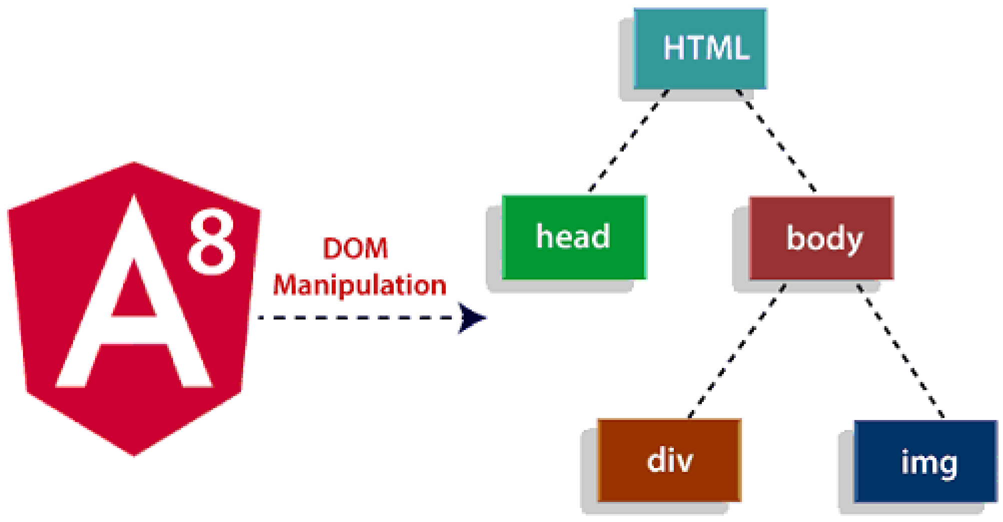
Expense Entry

Item:	Pizza
Amount:	20
Category:	Food
Location:	Zomato
Spend On:	June 20, 2020

4.5. Directives, Expression, Filters

Angular 8 Directives

The Angular 8 directives are used to manipulate the DOM. By using Angular directives, you can change the appearance, behavior or a layout of a DOM element. It also helps you to extend HTML.



Angular 8 Directive

Angular 8 directives can be classified in 3 categories based on how they behave:

- Component Directives
- Structural Directives
- Attribute Directives

Component Directives: Component directives are used in main class. They contain the detail of how the component should be processed, instantiated and used at runtime.

Structural Directives: Structural directives start with a * sign. These directives are used to manipulate and change the structure of the DOM elements. For example, *ngIf directive, *ngSwitch directive, and *ngFor directive.

- ***ngIf Directive:** The ngIf allows us to Add/Remove DOM Element.
- ***ngSwitch Directive:** The *ngSwitch allows us to Add/Remove DOM Element. It is similar to switch statement of C#.
- ***ngFor Directive:** The *ngFor directive is used to repeat a portion of HTML template once per each item from an iterable list (Collection).

Attribute Directives: Attribute directives are used to change the look and behavior of the DOM elements. For example: ngClass directive, and ngStyle directive etc.

- **ngClass Directive:** The ngClass directive is used to add or remove CSS classes to an HTML element.
- **ngStyle Directive:** The ngStyle directive facilitates you to modify the style of an HTML element using the expression. You can also use ngStyle directive to dynamically change the style of your HTML element.

Angular 8 ngIf Directive

The ngIf Directives is used to add or remove HTML Elements according to the expression. The expression must return a Boolean value. If the expression is false then the element is removed, otherwise element is inserted. It is similar to the ng-if directive of AngularJS.

ngIf Syntax

```
<p *ngIf="condition">
  condition is true and ngIf is true.
</p>
<p *ngIf="!condition">
  condition is false and ngIf is false.
</p>
```

The *ngIf directive form with an "else" block

```
<div *ngIf="condition; else elseBlock">
Content to render when condition is true.
</div>
<ng-template #elseBlock>
Content to render when condition is false.
</ng-template>
```

The ngIf directive 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.

The *ngIf directive is most commonly used to conditionally show an inline template. See the following example:

```
@Component({
 selector: 'ng-if-simple',
 template: `
  <button (click)="show = !show">{{show ? 'hide' : 'show'}}</button>
  show = {{show}}
  <br>
  <div *ngIf="show">Text to show</div>
`
})
export class NgIfSimple {
 show: boolean = true;
}
```

Same template example with else block

```
@Component({
  selector: 'ng-if-else',
  template: `
    <button (click)="show = !show">{{show ? 'hide' : 'show'}}</button>
    show = {{show}}
    <br>
    <div *ngIf="show; else elseBlock">Text to show</div>
    <ng-template #elseBlock>Alternate text while primary text is hidden</ng-template>
`
})
export class NgIfElse {
  show: boolean = true;
}
```

Angular 8 *ngFor Directive

The *ngFor directive is used to repeat a portion of HTML template once per each item from an iterable list (Collection). The ngFor is an Angular structural directive and is similar to ngRepeat in AngularJS. Some local variables like Index, First, Last, odd and even are exported by *ngFor directive.

Syntax of ngFor

See the simplified syntax for the ngFor directive:

```
<li *ngFor="let item of items;"> .... </li>
```

How to use ngFor Directive?

To Use ngFor directive, you have to create a block of HTML elements, which can display a single item of the items collection. After that you can use the ngFor directive to tell angular to repeat that block of HTML elements for each item in the list.

Example for *ngFor Directive

First, you have to create an angular Application. After that open the app.component.ts and add the following code.

The following Code contains a list of Top 3 movies in a movies array. Let's build a template to display these movies in a tabular form.

```
import { Component } from '@angular/core';
@Component({
   selector: 'movie-app',
   templateUrl:'./app/app.component.html',
   styleUrls:['./app/app.component.css']
})
export class AppComponent
{
    title: string ="Top 10 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 America: 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'},
    ]
}
class Movie {
    title : string;
    director : string;
    cast : string;
    releaseDate : string;
}
```

Now, open the app. component.html and add the following code:

```
<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>
```

When you run the application, It will show the movies in tabular form.

Angular 8 ngSwitch Directive

In Angular 8, ngSwitch is a structural directive which is used to Add/Remove DOM Element. It is similar to switch statement of C#. The ngSwitch directive is applied to the container element with a switch expression.

Syntax of ngSwitch

```
<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>
```

ngSwitchCase

In Angular ngSwitchCase directive, the inner elements are placed inside the container element. The ngSwitchCase directive is applied to the inner elements with a match expression. Whenever the value of the match expression matches the value of the switch expression, the corresponding inner element is added to the DOM. All other inner elements are removed from the DOM

If there is more than one match, then all the matching elements are added to the DOM.

ngSwitchDefault

You can also apply the ngSwitchDefault directive in Angular 8. The element with ngSwitchDefault is displayed only if no match is found. The inner element with ngSwitchDefault can be placed anywhere inside the container element and not necessarily at the bottom. If you add more than one ngSwitchDefault directive, all of them are displayed.

Any elements placed inside the container element, but outside the ngSwitchCase or ngSwitchDefault elements are displayed as it is.

ngSwitch Directive Example

Use the following code in app.component.ts file of your application:

```
class item {
   name: string;
   val: number;
}
export class AppComponent
{
   items: item[] = [{name: 'One', val: 1}, {name: 'Two', val: 2}, {name: 'Three', val: 3}];
   selectedValue: string= 'One';
}
```

Use the following code in the app.component.html file of your application:

```
<select [(ngModel)]="selectedValue">
   <option *ngFor="let item of items;" [value]="item.name">{{item.name}}</option>
</select>
<div class='row' [ngSwitch]="selectedValue">
   <div *ngSwitchCase="'One'">One is Pressed</div>
   <div *ngSwitchCase="'Two'">Two is Selected</div>
   <div *ngSwitchDefault>Default Option</div>
</div>
```

Dependency injection in Angular

Dependencies are services or objects that a class needs to perform its function. Dependency injection, or DI, is a design pattern in which a class requests dependencies from external sources rather than creating them.
Angular's DI framework provides dependencies to a class upon instantiation. You can use Angular DI to increase flexibility and modularity in your applications.
See the live example / download example for a working example containing the code snippets in this guide.

Creating an injectable service

To generate a new HeroService class in the src/app/heroes folder use the following Angular CLI command.

```
content_copyng generate service heroes/hero
```

This command creates the following default HeroService.

src/app/heroes/hero.service.ts (CLI-generated)

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

@Injectable({
  providedIn: 'root',
})
export class HeroService {
  constructor() { }
}
```

The @Injectable() decorator specifies that Angular can use this class in the DI system. The metadata, providedIn: 'root', means that the HeroService is visible throughout the application.

Next, to get the hero mock data, add a getHeroes() method that returns the heroes from mock.heroes.ts.

src/app/heroes/hero.service.ts

```
content_copyimport { Injectable } from '@angular/core';
import { HEROES } from './mock-heroes';

@Injectable({
  // declares that this service should be created
  // by the root application injector.
  providedIn: 'root',
})
export class HeroService {
  getHeroes() { return HEROES; }
}
```

For clarity and maintainability, it is recommended that you define components and services in separate files.

If you do combine a component and service in the same file, it is important to define the service first, and then the component. If you define the component before the service, Angular returns a run-time null reference error.

Injecting services

Injecting services results in making them visible to a component.

To inject a dependency in a component's constructor(), supply a constructor argument with the dependency type. The following example specifies the HeroService in the HeroListComponent constructor. The type of heroService is HeroService.

src/app/heroes/hero-list.component (constructor signature)

```
content_copyconstructor(heroService: HeroService)
```

For more information, see Providing dependencies in modules and Hierarchical injectors.

Using services in other services

When a service depends on another service, follow the same pattern as injecting into a component. In the following example HeroService depends on a Logger service to report its activities.

First, import the Logger service. Next, inject the Logger service in the HeroService constructor() by specifying private logger: Logger within the parentheses.

When you create a class whose constructor() has parameters, specify the type and metadata about those parameters so that Angular can inject the correct service.

Here, the constructor() specifies a type of Logger and stores the instance of Logger in a private field called logger.

The following code tabs feature the Logger service and two versions of HeroService. The first version of HeroService does not depend on the Logger service. The revised second version does depend on Logger service.

src/app/heroes/hero.service (v2)
src/app/heroes/hero.service (v1)
src/app/logger.service

```
content_copyimport { Injectable } from '@angular/core';
import { HEROES } from './mock-heroes';
import { Logger } from '../logger.service';

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

  constructor(private logger: Logger) {  }

  getHeroes() {
    this.logger.log('Getting heroes ...');
    return HEROES;
  }
}
```

In this example, the getHeroes() method uses the Logger service by logging a message when fetching heroes.

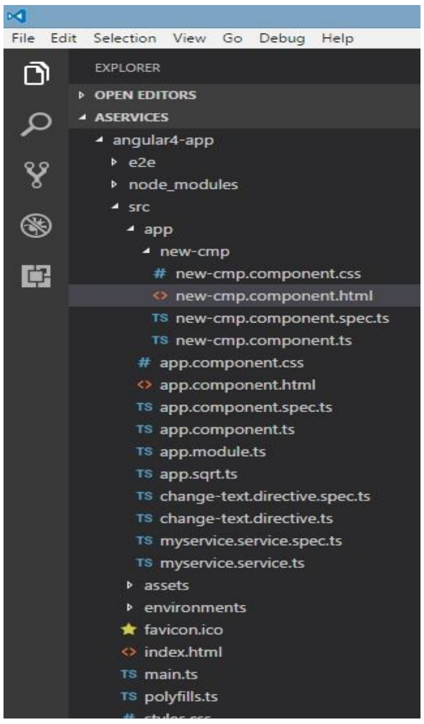
Angular Service

We might come across a situation where we need some code to be used everywhere on the page. It can be for data connection that needs to be shared across components, etc. Services help us achieve that. With services, we can access methods and properties across other components in the entire project.

To create a service, we need to make use of the command line. The command for the same is −

```
C:\projectA4\Angular 4-app>ng g service myservice
installing service
   create src\app\myservice.service.spec.ts
   create src\app\myservice.service.ts
   WARNING Service is generated but not provided, it must be provided to be used

   C:\projectA4\Angular 4-app>
```

The files are created in the app folder as follows −



```
angular4-app
  e2e
  node_modules
  src
    app
      new-cmp
        new-cmp.component.css
        new-cmp.component.html
        new-cmp.component.spec.ts
        new-cmp.component.ts
      app.component.css
      app.component.html
      app.component.spec.ts
      app.component.ts
      app.module.ts
      app.sqrt.ts
      change-text.directive.spec.ts
      change-text.directive.ts
      myservice.service.spec.ts
      myservice.service.ts
    assets
    environments
    favicon.ico
    index.html
    main.ts
    polyfills.ts
```

Following are the files created at the bottom - **myservice.service.specs.ts** and **myservice.service.ts**.

myservice.service.ts

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

@Injectable()
export class MyserviceService {
  constructor() { }
}
```

Here, the Injectable module is imported from the **@angular/core**. It contains the **@Injectable** method and a class called **MyserviceService**. We will create our service function in this class.

Before creating a new service, we need to include the service created in the main parent **app.module.ts**.

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

import { MyserviceService } from './myservice.service';
import { NewCmpComponent } from './new-cmp/new-cmp.component';
import { ChangeTextDirective } from './change-text.directive';
import { SqrtPipe } from './app.sqrt';

@NgModule({
  declarations: [
    SqrtPipe,
    AppComponent,
    NewCmpComponent,
    ChangeTextDirective
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot([
      {
        path: 'new-cmp',
        component: NewCmpComponent
      }
    ])
  ],
  providers: [MyserviceService],
  bootstrap: [AppComponent]
})

export class AppModule { }
```

We have imported the Service with the class name and the same class is used in the providers. Let us now switch back to the service class and create a service function.

In the service class, we will create a function which will display today's date. We can use the same function in the main parent component **app.component.ts** and also in the new component **new-cmp.component.ts** that we created in the previous chapter.

Let us now see how the function looks in the service and how to use it in components.

```
import { Injectable } from '@angular/core';
@Injectable()
export class MyserviceService {
  constructor() { }
  showTodayDate() {
    let ndate = new Date();
    return ndate;
  }
}
```

In the above service file, we have created a function **showTodayDate**. Now we will return the new Date () created. Let us see how we can access this function in the component class.

app.component.ts

```
import { Component } from '@angular/core';
import { MyserviceService } from './myservice.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title = 'Angular 4 Project!';
  todaydate;
  constructor(private myservice: MyserviceService) {}
  ngOnInit() {
    this.todaydate = this.myservice.showTodayDate();
  }
}
```

The **ngOnInit** function gets called by default in any component created. The date is fetched from the service as shown above. To fetch more details of the service, we need to first include the service in the component **ts** file.

We will display the date in the **.html** file as shown below –

```
{{todaydate}}
<app-new-cmp></app-new-cmp>
// data to be displayed to user from the new component class.
```

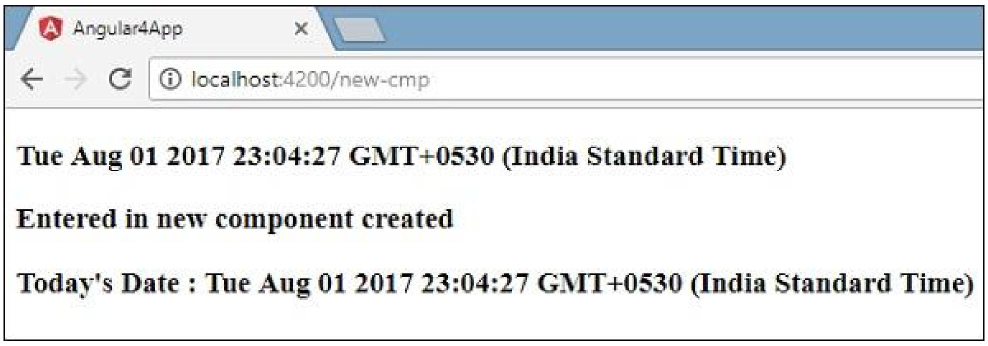
Let us now see how to use the service in the new component created.

```
import { Component, OnInit } from '@angular/core';
import { MyserviceService } from './../myservice.service';

@Component({
  selector: 'app-new-cmp',
  templateUrl: './new-cmp.component.html',
  styleUrls: ['./new-cmp.component.css']
})

export class NewCmpComponent implements OnInit {
  todaydate;
  newcomponent = "Entered in new component created";
  constructor(private myservice: MyserviceService) {}

  ngOnInit() {
    this.todaydate = this.myservice.showTodayDate();
  }
}
```

In the new component that we have created, we need to first import the service that we want and access the methods and properties of the same. Please see the code highlighted. The todaydate is displayed in the component html as follows –

```
<p>
  {{newcomponent}}
</p>
<p>
  Today's Date : {{todaydate}}
</p>
```

The selector of the new component is used in the **app.component.html** file. The contents from the above html file will be displayed in the browser as shown below –



```
Tue Aug 01 2017 23:04:27 GMT+0530 (India Standard Time)

Entered in new component created

Today's Date : Tue Aug 01 2017 23:04:27 GMT+0530 (India Standard Time)
```

If you change the property of the service in any component, the same is changed in other components too. Let us now see how this works.

We will define one variable in the service and use it in the parent and the new component. We will again change the property in the parent component and will see if the same is changed in the new component or not.

In **myservice.service.ts**, we have created a property and used the same in other parent and new component.

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

@Injectable()
export class MyserviceService {
  serviceproperty = "Service Created";
  constructor() { }
  showTodayDate() {
    let ndate = new Date();
    return ndate;
  }
}
```

Let us now use the **serviceproperty** variable in other components. In **app.component.ts**, we are accessing the variable as follows −

```
import { Component } from '@angular/core';
import { MyserviceService } from './myservice.service';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title = 'Angular 4 Project!';
  todaydate;
  componentproperty;
  constructor(private myservice: MyserviceService) {}
  ngOnInit() {
    this.todaydate = this.myservice.showTodayDate();
    console.log(this.myservice.serviceproperty);
    this.myservice.serviceproperty = "component created"; // value is changed.
    this.componentproperty = this.myservice.serviceproperty;
  }
}
```

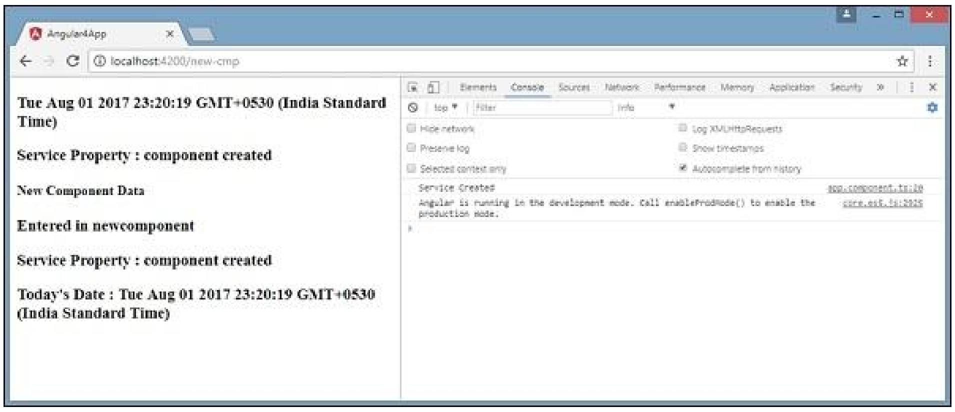
We will now fetch the variable and work on the console.log. In the next line, we will change the value of the variable to “**component created**”. We will do the same in **new-cmp.component.ts**.

```
import { Component, OnInit } from '@angular/core';
import { MyserviceService } from './../myservice.service';

@Component({
  selector: 'app-new-cmp',
  templateUrl: './new-cmp.component.html',
  styleUrls: ['./new-cmp.component.css']
})

export class NewCmpComponent implements OnInit {
  todaydate;
  newcomponentproperty;
  newcomponent = "Entered in newcomponent";
  constructor(private myservice: MyserviceService) {}
  ngOnInit() {
    this.todaydate = this.myservice.showTodayDate();
    this.newcomponentproperty = this.myservice.serviceproperty;
  }
}
```

In the above component, we are not changing anything but directly assigning the property to the component property.

Now when you execute it in the browser, the service property will be changed since the value of it is changed in **app.component.ts** and the same will be displayed for the **new-cmp.component.ts**.

Also check the value in the console before it is changed.



Tue Aug 01 2017 23:20:19 GMT+0530 (India Standard Time)

Service Property : component created

New Component Data

Entered in newcomponent

Service Property : component created

Today's Date : Tue Aug 01 2017 23:20:19 GMT+0530 (India Standard Time)

```
Service Created                                                         app.component.ts:10
Angular is running in the development mode. Call enableProdMode() to enable the    core.es5.js:3025
production mode.
```

Angular Http Service

Http Service will help us fetch external data, post to it, etc. We need to import the http module to make use of the http service. Let us consider an example to understand how to make use of the http service.

To start using the http service, we need to import the module in **app.module.ts** as shown below −

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { HttpModule } from '@angular/http';
import { AppComponent } from './app.component';

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

If you see the highlighted code, we have imported the HttpModule from @angular/http and the same is also added in the imports array.

Let us now use the http service in the **app.component.ts**.

```
import { Component } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  constructor(private http: Http) { }
  ngOnInit() {
    this.http.get("http://jsonplaceholder.typicode.com/users").
    map((response) ⇒ response.json()).
    subscribe((data) ⇒ console.log(data))
  }
}
```

Let us understand the code highlighted above. We need to import http to make use of the service, which is done as follows –

```
import { Http } from '@angular/http';
```

In the class **AppComponent**, a constructor is created and the private variable http of type Http. To fetch the data, we need to use the **get API** available with http as follows

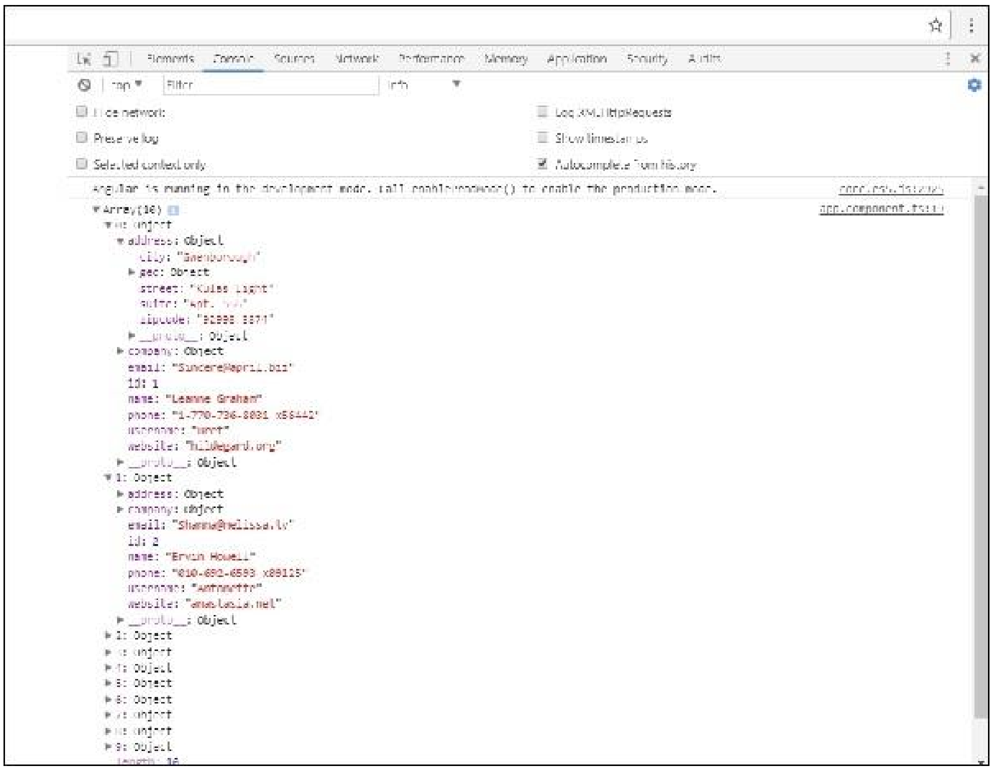
```
this.http.get();
```

It takes the url to be fetched as the parameter as shown in the code.

We will use the test url - https://jsonplaceholder.typicode.com/users to fetch the json data. Two operations are performed on the fetched url data map and subscribe. The Map method helps to convert the data to json format. To use the map, we need to import the same as shown below –

```
import 'rxjs/add/operator/map';
```

Once the map is done, the subscribe will log the output in the console as shown in the browser –



```
☐ Hide network                 ☐ Log XMLHttpRequests
☐ Preserve log                 ☐ Show timestamps
☐ Selected context only        ☑ Autocomplete from history
```

```
Angular is running in the development mode. Call enableProdMode() to enable the production mode.   core.es5.js:3025
▼Array(10)                                                                                           app.component.ts:11
  ▼0: Object
    ▼address: Object
       city: "Gwenborough"
      ▶geo: Object
       street: "Kulas Light"
       suite: "Apt. 556"
       zipcode: "92998-3874"
      ▶__proto__: Object
    ▶company: Object
     email: "Sincere@april.biz"
     id: 1
     name: "Leanne Graham"
     phone: "1-770-736-8031 x56442"
     username: "Bret"
     website: "hildegard.org"
    ▶__proto__: Object
  ▼1: Object
    ▶address: Object
    ▶company: Object
     email: "Shanna@melissa.tv"
     id: 2
     name: "Ervin Howell"
     phone: "010-692-6593 x09125"
     username: "Antonette"
     website: "anastasia.net"
    ▶__proto__: Object
  ▶2: Object
  ▶3: Object
  ▶4: Object
  ▶5: Object
  ▶6: Object
  ▶7: Object
  ▶8: Object
  ▶9: Object
   length: 10
```

If you see, the json objects are displayed in the console. The objects can be displayed in the browser too.

For the objects to be displayed in the browser, update the codes in **app.component.html** and **app.component.ts** as follows −

```
import { Component } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(private http: Http) { }
  httpdata;
  ngOnInit() {
    this.http.get("http://jsonplaceholder.typicode.com/users").
    map(
      (response) ⇒ response.json()
    ).
    subscribe(
      (data) ⇒ {this.displaydata(data);}
    )
  }
  displaydata(data) {this.httpdata = data;}
}
```

In **app.component.ts**, using the subscribe method we will call the display data method and pass the data fetched as the parameter to it.

In the display data method, we will store the data in a variable httpdata. The data is displayed in the browser using **for** over this httpdata variable, which is done in the **app.component.html** file.

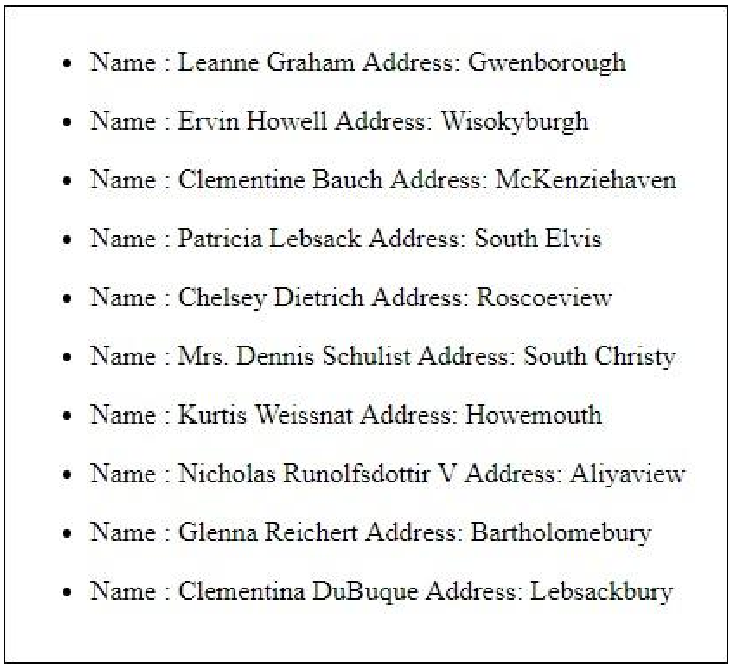
```
<ul *ngFor = "let data of httpdata">
  <li>Name : {{data.name}} Address: {{data.address.city}}</li>
</ul>
```

The json object is as follows −

```
{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "Sincere@april.biz",

  "address": {
    "street": "Kulas Light",
    "suite": "Apt. 556",
    "city": "Gwenborough",
    "zipcode": "92998-3874",
    "geo": {
      "lat": "-37.3159",
      "lng": "81.1496"
    }
  },

  "phone": "1-770-736-8031 x56442",
  "website": "hildegard.org",
  "company": {
    "name": "Romaguera-Crona",
    "catchPhrase": "Multi-layered client-server neural-net",
    "bs": "harness real-time e-markets"
  }
}
```

The object has properties such as id, name, username, email, and address that internally has street, city, etc. and other details related to phone, website, and company. Using the **for** loop, we will display the name and the city details in the browser as shown in the **app.component.html** file.

This is how the display is shown in the browser −



```
• Name : Leanne Graham Address: Gwenborough
• Name : Ervin Howell Address: Wisokyburgh
• Name : Clementine Bauch Address: McKenziehaven
• Name : Patricia Lebsack Address: South Elvis
• Name : Chelsey Dietrich Address: Roscoeview
• Name : Mrs. Dennis Schulist Address: South Christy
• Name : Kurtis Weissnat Address: Howemouth
• Name : Nicholas Runolfsdottir V Address: Aliyaview
• Name : Glenna Reichert Address: Bartholomebury
• Name : Clementina DuBuque Address: Lebsackbury
```

Let us now add the search parameter, which will filter based on specific data. We need to fetch the data based on the search param passed.

Following are the changes done in **app.component.html** and **app.component.ts** files –

app.component.ts

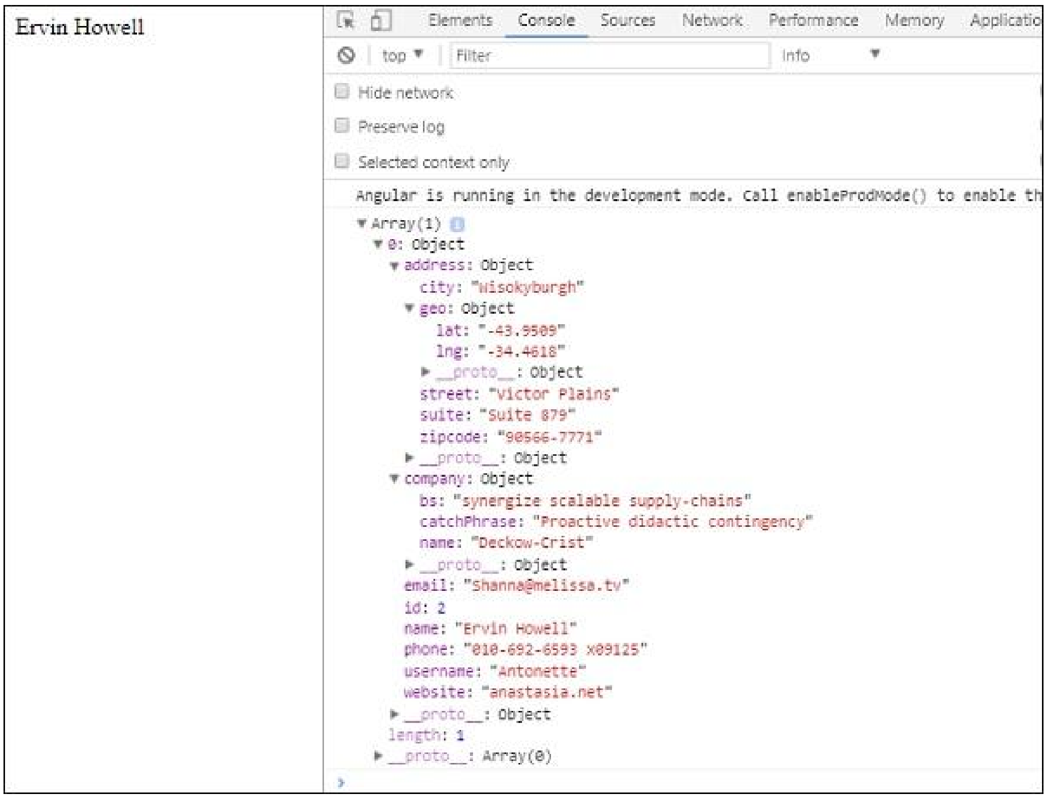
```
import { Component } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app';
  searchparam = 2;
  jsondata;
  name;
  constructor(private http: Http) { }
  ngOnInit() {
    this.http.get("http://jsonplaceholder.typicode.com/users?id="+this.searchparam).
    map(
      (response) ⇒ response.json()
    ).
    subscribe((data) ⇒ this.converttoarray(data))
  }
  converttoarray(data) {
    console.log(data);
    this.name = data[0].name;
  }
}
```

For the **get api**, we will add the search param id = this.searchparam. The searchparam is equal to 2. We need the details of **id=2** from the json file.

app.component.html

```
{{name}}
```

This is how the browser is displayed –



Ervin Howell

```
Angular is running in the development mode. Call enableProdMode() to enable th
▼ Array(1)
  ▼ 0: Object
    ▼ address: Object
        city: "Wisokyburgh"
      ▼ geo: Object
          lat: "-43.9509"
          lng: "-34.4618"
        ▶ __proto__: Object
        street: "Victor Plains"
        suite: "Suite 879"
        zipcode: "90566-7771"
      ▶ __proto__: Object
    ▼ company: Object
        bs: "synergize scalable supply-chains"
        catchPhrase: "Proactive didactic contingency"
        name: "Deckow-Crist"
      ▶ __proto__: Object
      email: "Shanna@melissa.tv"
      id: 2
      name: "Ervin Howell"
      phone: "010-692-6593 x09125"
      username: "Antonette"
      website: "anastasia.net"
    ▶ __proto__: Object
    length: 1
  ▶ __proto__: Array(0)
>
```

We have consoled the data in the browser, which is received from the http. The same is displayed in the browser console. The name from the json with **id=2** is displayed in the browser.

Angular Routing

In a single-page app, you change what the user sees by showing or hiding portions of the display that correspond to particular components, rather than going out to the server to get a new page. As users perform application tasks, they need to move between the different views that you have defined.
To handle the navigation from one view to the next, you use the Angular Router. The Router enables navigation by interpreting a browser URL as an instruction to change the view.

Common Routing Tasks

This topic describes how to implement many of the common tasks associated with adding the Angular router to your application.

Generate an application with routing enabled

The following command uses the Angular CLI to generate a basic Angular application with an application routing module, called AppRoutingModule, which is an NgModule where you can configure your routes. The application name in the following example is routing-app.

```
content_copyng new routing-app --routing
```

When generating a new application, the CLI prompts you to select CSS or a CSS preprocessor. For this example, accept the default of CSS.

Adding components for routing

To use the Angular router, an application needs to have at least two components so that it can navigate from one to the other. To create a component using the CLI, enter the following at the command line where first is the name of your component:

```
content_copyng generate component first
```

Repeat this step for a second component but give it a different name. Here, the new name is second.

```
content_copyng generate component second
```

The CLI automatically appends Component, so if you were to write first-component, your component would be FirstComponentComponent.

> ***<base href>***
> This guide works with a CLI-generated Angular application. If you are working manually, make sure that you have <base href="/"> in the <head> of your index.html file. This assumes that the app folder is the application root, and uses "/".

Importing your new components

To use your new components, import them into AppRoutingModule at the top of the file, as follows:

AppRoutingModule (excerpt)

```
content_copyimport { FirstComponent } from './first/first.component';
import { SecondComponent } from './second/second.component';
```

Defining a basic route

There are three fundamental building blocks to creating a route.
Import the AppRoutingModule into AppModule and add it to the imports array.
The Angular CLI performs this step for you. However, if you are creating an application manually or working with an existing, non-CLI application, verify that the imports and configuration are correct. The following is the default AppModule using the CLI with the --routing flag.

Default CLI AppModule with routing

```
content_copyimport { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module'; // CLI imports AppRoutingModule
import { AppComponent } from './app.component';
```

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule // CLI adds AppRoutingModule to the AppModule's imports array
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

1. Import RouterModule and Routes into your routing module.
   The Angular CLI performs this step automatically. The CLI also sets up a Routes array for your routes and configures the imports and exports arrays for @NgModule().
   CLI application routing module

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

   const routes: Routes = []; // sets up routes constant where you define your routes

   // configures NgModule imports and exports
   @NgModule({
     imports: [RouterModule.forRoot(routes)],
     exports: [RouterModule]
   })
   export class AppRoutingModule { }
   ```
2. Define your routes in your Routes array.
   Each route in this array is a JavaScript object that contains two properties. The first property, path, defines the URL path for the route. The second property, component, defines the component Angular should use for the corresponding path.
   AppRoutingModule (excerpt)

   ```
   content_copyconst routes: Routes = [
     { path: 'first-component', component: FirstComponent },
     { path: 'second-component', component: SecondComponent },
   ];
   ```
3. Add your routes to your application.
   Now that you have defined your routes, you can add them to your application. First, add links to the two components. Assign the anchor tag that you want to add the route to the routerLink attribute. Set the value of the attribute to the component to show when a user clicks on each link. Next, update your component template to include <router-outlet>. This element informs Angular to update the application view with the component for the selected route.
   Template with routerLink and router-outlet

   ```
   content_copy<h1>Angular Router App</h1>
   <!-- This nav gives you links to click, which tells the router which route to use (defined in
   the routes constant in  AppRoutingModule) -->
   <nav>
     <ul>
                   <li><a    routerLink="/first-component"    routerLinkActive="active">First
   Component</a></li>
               <li><a   routerLink="/second-component"   routerLinkActive="active">Second
   Component</a></li>
     </ul>
   ```

```
</nav>
<!-- The routed views render in the <router-outlet>-->
<router-outlet></router-outlet>
```

Route order

The order of routes is important because the Router uses a first-match wins strategy when matching routes, so more specific routes should be placed above less specific routes. List routes with a static path first, followed by an empty path route, which matches the default route. The wildcard route comes last because it matches every URL and the Router selects it only if no other routes match first.

Getting route information

Often, as a user navigates your application, you want to pass information from one component to another. For example, consider an application that displays a shopping list of grocery items. Each item in the list has a unique id. To edit an item, users click an Edit button, which opens an EditGroceryItem component. You want that component to retrieve the id for the grocery item so it can display the right information to the user.

You can use a route to pass this type of information to your application components. To do so, you use the ActivatedRoute interface.

To get information from a route:

1. Import ActivatedRoute and ParamMap to your component.
   In the component class (excerpt)
   ```
   content_copyimport { Router, ActivatedRoute, ParamMap } from '@angular/router';
   ```
   These import statements add several important elements that your component needs. To learn more about each, see the following API pages:
   - Router
   - ActivatedRoute
   - ParamMap
2. Inject an instance of ActivatedRoute by adding it to your application's constructor:
   In the component class (excerpt)
   ```
   content_copyconstructor(
     private route: ActivatedRoute,
   ) {}
   ```
3. Update the ngOnInit() method to access the ActivatedRoute and track the id parameter:
   In the component (excerpt)
   ```
   content_copyngOnInit() {
     this.route.queryParams.subscribe(params => {
       this.name = params['name'];
     });
   }
   ```
   Note: The preceding example uses a variable, name, and assigns it the value based on the name parameter.

Setting up wildcard routes

A well-functioning application should gracefully handle when users attempt to navigate to a part of your application that does not exist. To add this functionality to your application, you set up a wildcard route. The Angular router selects this route any time the requested URL doesn't match any router paths.

To set up a wildcard route, add the following code to your routes definition.

AppRoutingModule (excerpt)

```
content_copy{ path: '**', component:  }
```

The two asterisks, **, indicate to Angular that this routes definition is a wildcard route. For the component property, you can define any component in your application. Common choices include an application-specific PageNotFoundComponent, which you can define to display a 404 page to your users;

or a redirect to your application's main component. A wildcard route is the last route because it matches any URL. For more detail on why order matters for routes, see Route order.

Displaying a 404 page

To display a 404 page, set up a wildcard route with the component property set to the component you'd like to use for your 404 page as follows:

AppRoutingModule (excerpt)

```
content_copyconst routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  { path: '**', component: PageNotFoundComponent },  // Wildcard route for a 404 page
];
```

The last route with the path of ** is a wildcard route. The router selects this route if the requested URL doesn't match any of the paths earlier in the list and sends the user to the PageNotFoundComponent.

Setting up redirects

To set up a redirect, configure a route with the path you want to redirect from, the component you want to redirect to, and a pathMatch value that tells the router how to match the URL.

AppRoutingModule (excerpt)

```
content_copyconst routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  { path: '',   redirectTo: '/first-component', pathMatch: 'full' }, // redirect to `first-component`
  { path: '**', component: PageNotFoundComponent },  // Wildcard route for a 404 page
];
```

In this example, the third route is a redirect so that the router defaults to the first-component route. Notice that this redirect precedes the wildcard route. Here, path: '' means to use the initial relative URL ('').
For more details on pathMatch see Spotlight on pathMatch.

Nesting routes

As your application grows more complex, you may want to create routes that are relative to a component other than your root component. These types of nested routes are called child routes. This means you're adding a second <router-outlet> to your app, because it is in addition to the <router-outlet> in AppComponent.
In this example, there are two additional child components, child-a, and child-b. Here, FirstComponent has its own <nav> and a second <router-outlet> in addition to the one in AppComponent.

In the template

```
content_copy<h2>First Component</h2>

<nav>
  <ul>
    <li><a routerLink="child-a">Child A</a></li>
    <li><a routerLink="child-b">Child B</a></li>
  </ul>
</nav>

<router-outlet></router-outlet>
```

A child route is like any other route, in that it needs both a path and a component. The one difference is that you place child routes in a children array within the parent route.

AppRoutingModule (excerpt)

```
content_copyconst routes: Routes = [
  {
    path: 'first-component',
    component: FirstComponent, // this is the component with the <router-outlet> in the template
    children: [
      {
        path: 'child-a', // child route path
        component: ChildAComponent, // child route component that the router renders
      },
      {
        path: 'child-b',
        component: ChildBComponent, // another child route component that the router renders
      },
    ],
  },
];
```

Using relative paths

Relative paths allow you to define paths that are relative to the current URL segment. The following example shows a relative route to another component, second-component. FirstComponent and SecondComponent are at the same level in the tree, however, the link to SecondComponent is situated within the FirstComponent, meaning that the router has to go up a level and then into the second directory to find the SecondComponent. Rather than writing out the whole path to get to SecondComponent, you can use the ../ notation to go up a level.

In the template

```
content_copy<h2>First Component</h2>

<nav>
  <ul>
    <li><a routerLink="../second-component">Relative Route to second component</a></li>
  </ul>
</nav>
<router-outlet></router-outlet>
```

In addition to ../, you can use ./ or no leading slash to specify the current level.

Specifying a relative route

To specify a relative route, use the NavigationExtras relativeTo property. In the component class, import NavigationExtras from the @angular/router.

Then use relativeTo in your navigation method. After the link parameters array, which here contains items, add an object with the relativeTo property set to the ActivatedRoute, which is this.route.

RelativeTo

```
content_copygoToItems() {
  this.router.navigate(['items'], { relativeTo: this.route });
}
```

The goToItems() method interprets the destination URI as relative to the activated route and navigates to the items route.

Accessing query parameters and fragments

Sometimes, a feature of your application requires accessing a part of a route, such as a query parameter or a fragment. The Tour of Heroes application at this stage in the tutorial uses a list view in which you can click on a hero to see details. The router uses an id to show the correct hero's details.

First, import the following members in the component you want to navigate from.

Component import statements (excerpt)

```
content_copyimport { ActivatedRoute } from '@angular/router';
import { Observable } from 'rxjs';
import { switchMap } from 'rxjs/operators';
```

Next inject the activated route service:

Component (excerpt)

```
content_copyconstructor(private route: ActivatedRoute) {}
```

Configure the class so that you have an observable, heroes$, a selectedId to hold the id number of the hero, and the heroes in the ngOnInit(), add the following code to get the id of the selected hero. This code snippet assumes that you have a heroes list, a hero service, a function to get your heroes, and the HTML to render your list and details, just as in the Tour of Heroes example.

Component 1 (excerpt)

```
content_copyheroes$: Observable;
selectedId: number;
heroes = HEROES;

ngOnInit() {
  this.heroes$ = this.route.paramMap.pipe(
    switchMap(params => {
      this.selectedId = Number(params.get('id'));
      return this.service.getHeroes();
    })
  );
}
```

Next, in the component that you want to navigate to, import the following members.

Component 2 (excerpt)

```
content_copyimport { Router, ActivatedRoute, ParamMap } from '@angular/router';
import { Observable } from 'rxjs';
```

Inject ActivatedRoute and Router in the constructor of the component class so they are available to this component:

Component 2 (excerpt)

```
content_copyhero$: Observable;

  constructor(
    private route: ActivatedRoute,
    private router: Router  ) {}

  ngOnInit() {
    const heroId = this.route.snapshot.paramMap.get('id');
    this.hero$ = this.service.getHero(heroId);
  }

  gotoItems(hero: Hero) {
    const heroId = hero ? hero.id : null;
    // Pass along the hero id if available
    // so that the HeroList component can select that item.
    this.router.navigate(['/heroes', { id: heroId }]);
  }
```

Lazy loading

You can configure your routes to lazy load modules, which means that Angular only loads modules as needed, rather than loading all modules when the application launches. Additionally, you can preload parts of your application in the background to improve the user experience.
For more information on lazy loading and preloading see the dedicated guide Lazy loading NgModules.

Preventing unauthorized access

Use route guards to prevent users from navigating to parts of an application without authorization. The following route guards are available in Angular:

- CanActivate
- CanActivateChild
- CanDeactivate
- Resolve
- CanLoad

To use route guards, consider using component-less routes as this facilitates guarding child routes.
Create a service for your guard:

```
content_copyng generate guard your-guard
```

In your guard class, implement the guard you want to use. The following example uses CanActivate to guard the route.
Component (excerpt)

```
content_copyexport class YourGuard implements CanActivate {
  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): boolean {
      // your  logic goes here
  }
}
```

In your routing module, use the appropriate property in your routes configuration. Here, canActivate tells the router to mediate navigation to this particular route.
Routing module (excerpt)

```
content_copy{
  path: '/your-path',
  component: YourComponent,
  canActivate: [YourGuard],
}
```

For more information with a working example, see the routing tutorial section on route guards.

Link parameters array

A link parameters array holds the following ingredients for router navigation:

- The path of the route to the destination component.
- Required and optional route parameters that go into the route URL.

You can bind the RouterLink directive to such an array like this:
src/app/app.component.ts (h-anchor)

```
content_copy<a [routerLink]="['/heroes']">Heroes</a>
```

The following is a two-element array when specifying a route parameter:
src/app/heroes/hero-list/hero-list.component.html (nav-to-detail)

```
content_copy<a [routerLink]="['/hero', hero.id]">
  <span class="badge">{{ hero.id }}</span>{{ hero.name }}
</a>
```

You can provide optional route parameters in an object, as in { foo: 'foo' }:
src/app/app.component.ts (cc-query-params)

```
content_copy<a [routerLink]="['/crisis-center', { foo: 'foo' }]">Crisis Center</a>
```

These three examples cover the needs of an application with one level of routing. However, with a child router, such as in the crisis center, you create new link array possibilities.
The following minimal RouterLink example builds upon a specified default child route for the crisis center.
src/app/app.component.ts (cc-anchor-w-default)

```
content_copy<a [routerLink]="['/crisis-center']">Crisis Center</a>
```

Note the following:

- The first item in the array identifies the parent route (/crisis-center).
- There are no parameters for this parent route.
- There is no default for the child route so you need to pick one.
- You're navigating to the CrisisListComponent, whose route path is /, but you don't need to explicitly add the slash.

Consider the following router link that navigates from the root of the application down to the Dragon Crisis:
src/app/app.component.ts (Dragon-anchor)

```
content_copy<a [routerLink]="['/crisis-center', 1]">Dragon Crisis</a>
```

- The first item in the array identifies the parent route (/crisis-center).
- There are no parameters for this parent route.
- The second item identifies the child route details about a particular crisis (/:id).
- The details child route requires an id route parameter.
- You added the id of the Dragon Crisis as the second item in the array (1).
- The resulting path is /crisis-center/1.

You could also redefine the AppComponent template with Crisis Center routes exclusively:
src/app/app.component.ts (template)

```
content_copytemplate: `
  <h1 class="title">Angular Router</h1>
  <nav>
    <a [routerLink]="['/crisis-center']">Crisis Center</a>
    <a [routerLink]="['/crisis-center/1', { foo: 'foo' }]">Dragon Crisis</a>
    <a [routerLink]="['/crisis-center/2']">Shark Crisis</a>
  </nav>
  <router-outlet></router-outlet>
`
```

In summary, you can write applications with one, two or more levels of routing. The link parameters array affords the flexibility to represent any routing depth and any legal sequence of route paths, (required) router parameters, and (optional) route parameter objects.

LocationStrategy and browser URL styles

When the router navigates to a new component view, it updates the browser's location and history with a URL for that view.
Modern HTML5 browsers support history.pushState, a technique that changes a browser's location and history without triggering a server page request. The router can compose a "natural" URL that is indistinguishable from one that would otherwise require a page load.
Here's the Crisis Center URL in this "HTML5 pushState" style:

```
content_copylocalhost:3002/crisis-center/
```

Older browsers send page requests to the server when the location URL changes unless the change occurs after a "#" (called the "hash"). Routers can take advantage of this exception by composing in-application route URLs with hashes. Here's a "hash URL" that routes to the Crisis Center.

```
content_copylocalhost:3002/src/#/crisis-center/
```

The router supports both styles with two LocationStrategy providers:

1. PathLocationStrategy—the default "HTML5 pushState" style.
2. HashLocationStrategy—the "hash URL" style.

The RouterModule.forRoot() function sets the LocationStrategy to the PathLocationStrategy, which makes it the default strategy. You also have the option of switching to the HashLocationStrategy with an override during the bootstrapping process.

For more information on providers and the bootstrap process, see Dependency Injection.

Choosing a routing strategy

You must choose a routing strategy early in the development of your project because once the application is in production, visitors to your site use and depend on application URL references.
Almost all Angular projects should use the default HTML5 style. It produces URLs that are easier for users to understand and it preserves the option to do server-side rendering.
Rendering critical pages on the server is a technique that can greatly improve perceived responsiveness when the application first loads. An application that would otherwise take ten or more seconds to start could be rendered on the server and delivered to the user's device in less than a second.
This option is only available if application URLs look like normal web URLs without hashes (#) in the middle.

<base href>

The router uses the browser's history.pushState for navigation. pushState allows you to customize in-application URL paths; for example, localhost:4200/crisis-center. The in-application URLs can be indistinguishable from server URLs.
Modern HTML5 browsers were the first to support pushState which is why many people refer to these URLs as "HTML5 style" URLs.

HTML5 style navigation is the router default. In the LocationStrategy and browser URL styles section, learn why HTML5 style is preferable, how to adjust its behavior, and how to switch to the older hash (#) style, if necessary.

You must add a <base href> element to the application's index.html for pushState routing to work. The browser uses the <base href> value to prefix relative URLs when referencing CSS files, scripts, and images.
Add the <base> element just after the <head> tag. If the app folder is the application root, as it is for this application, set the href value in index.html as shown here.
src/index.html (base-href)

```
content_copy<base href="/">
```

HTML5 URLs and the <base href>

The guidelines that follow will refer to different parts of a URL. This diagram outlines what those parts refer to:

```
content_copyfoo://example.com:8042/over/there?name=ferret#nose
\_/   \______________/\_________/ \_________/ \__/
 |           |            |            |        |
scheme   authority      path       query   fragment
```

While the router uses the HTML5 pushState style by default, you must configure that strategy with a <base href>.
The preferred way to configure the strategy is to add a <base href> element tag in the <head> of the index.html.
src/index.html (base-href)

```
content_copy<base href="/">
```

Without that tag, the browser may not be able to load resources (images, CSS, scripts) when "deep linking" into the application.
Some developers may not be able to add the <base> element, perhaps because they don't have access to <head> or the index.html.
Those developers may still use HTML5 URLs by taking the following two steps:

1. Provide the router with an appropriate APP_BASE_HREF value.

2. Use root URLs (URLs with an authority) for all web resources: CSS, images, scripts, and template HTML files.

- The <base href> path should end with a "/", as browsers ignore characters in the path that follow the right-most "/".
- If the <base href> includes a query part, the query is only used if the path of a link in the page is empty and has no query. This means that a query in the <base href> is only included when using HashLocationStrategy.
- If a link in the page is a root URL (has an authority), the <base href> is not used. In this way, an APP_BASE_HREF with an authority will cause all links created by Angular to ignore the <base href> value.
- A fragment in the <base href> is *never* persisted.

For more complete information on how <base href> is used to construct target URIs, see the RFC section on transforming references.

HashLocationStrategy

You can use HashLocationStrategy by providing the useHash: true in an object as the second argument of the RouterModule.forRoot() in the AppModule.

src/app/app.module.ts (hash URL strategy)

```
content_copyimport { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { Routes, RouterModule } from '@angular/router';

import { AppComponent } from './app.component';
import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

const routes: Routes = [

];

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(routes, { useHash: true })  // .../#/crisis-center/
  ],
  declarations: [
    AppComponent,
    PageNotFoundComponent
  ],
  providers: [

  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Using Angular routes in a single-page application

This tutorial describes how you can build a single-page application, SPA that uses multiple Angular routes.

In a Single Page Application (SPA), all of your application's functions exist in a single HTML page. As users access your application's features, the browser needs to render only the parts that matter to the user, instead of loading a new page. This pattern can significantly improve your application's user experience.

To define how users navigate through your application, you use routes. You can add routes to define how users navigate from one part of your application to another. You can also configure routes to guard against unexpected or unauthorized behavior.

To explore a sample application featuring the contents of this tutorial, see the live example / download example.

Objectives

- Organize a sample application's features into modules.
- Define how to navigate to a component.
- Pass information to a component using a parameter.
- Structure routes by nesting several routes.
- Check whether users can access a route.
- Control whether the application can discard unsaved changes.
- Improve performance by pre-fetching route data and lazy loading feature modules.
- Require specific criteria to load components.

Create a sample application

Using the Angular CLI, create a new application, *angular-router-sample*. This application will have two components: *crisis-list* and *heroes-list*.

1. Create a new Angular project, *angular-router-sample*.
   ```
   content_copyng new angular-router-sample
   ```
   When prompted with Would you like to add Angular routing?, select N.
   When prompted with Which stylesheet format would you like to use?, select CSS.
   After a few moments, a new project, angular-router-sample, is ready.
2. From your terminal, navigate to the angular-router-sample directory.
3. Create a component, *crisis-list*.
   ```
   content_copyng generate component crisis-list
   ```
4. In your code editor, locate the file, crisis-list.component.html and replace the placeholder content with the following HTML.
   src/app/crisis-list/crisis-list.component.html
   ```
   content_copy<h3>CRISIS CENTER</h3>
   <p>Get your crisis here</p>
   ```
5. Create a second component, *heroes-list*.
   ```
   content_copyng generate component heroes-list
   ```
6. In your code editor, locate the file, heroes-list.component.html and replace the placeholder content with the following HTML.
   src/app/heroes-list/heroes-list.component.html
   ```
   content_copy<h3>HEROES</h3>
   <p>Get your heroes here</p>
   ```
7. In your code editor, open the file, app.component.html and replace its contents with the following HTML.
   src/app/app.component.html
   ```
   content_copy<h1>Angular Router Sample</h1>
   <app-crisis-list></app-crisis-list>
   <app-heroes-list></app-heroes-list>
   ```
8. Verify that your new application runs as expected by running the ng serve command.
   ```
   content_copyng serve
   ```
9. Open a browser to http://localhost:4200.
   You should see a single web page, consisting of a title and the HTML of your two components.

Import RouterModule from @angular/router

Routing allows you to display specific views of your application depending on the URL path. To add this functionality to your sample application, you need to update the app.module.ts file to use the module, RouterModule. You import this module from @angular/router.

1. From your code editor, open the app.module.ts file.
2. Add the following import statement.
   src/app/app.module.ts
   ```
   content_copyimport { RouterModule } from '@angular/router';
   ```

Define your routes

In this section, you'll define two routes:

- The route /crisis-center opens the crisis-center component.
- The route /heroes-list opens the heroes-list component.

A route definition is a JavaScript object. Each route typically has two properties. The first property, path, is a string that specifies the URL path for the route. The second property, component, is a string that specifies what component your application should display for that path.

1. From your code editor, open the app.module.ts file.
2. Locate the @NgModule() section.
3. Replace the imports array in that section with the following.
   src/app/app.module.ts

```
content_copyimports: [
  BrowserModule,
  RouterModule.forRoot([
    {path: 'crisis-list', component: CrisisListComponent},
    {path: 'heroes-list', component: HeroesListComponent},
  ]),
],
```

This code adds the RouterModule to the imports array. Next, the code uses the forRoot() method of the RouterModule to define your two routes. This method takes an array of JavaScript objects, with each object defining the properties of a route. The forRoot() method ensures that your application only instantiates one RouterModule. For more information, see Singleton Services.

Update your component with router-outlet

At this point, you have defined two routes for your application. However, your application still has both the crisis-list and heroes-list components hard-coded in your app.component.html template. For your routes to work, you need to update your template to dynamically load a component based on the URL path.

To implement this functionality, you add the router-outlet directive to your template file.

1. From your code editor, open the app.component.html file.
2. Delete the following lines.
   src/app/app.component.html

```
content_copy<app-crisis-list></app-crisis-list>
<app-heroes-list></app-heroes-list>
```

3. Add the router-outlet directive.
   src/app/app.component.html

```
content_copy<router-outlet></router-outlet>
```

View your updated application in your browser. You should see only the application title. To view the crisis-list component, add crisis-list to the end of the path in your browser's address bar. For example:

```
content_copyhttp://localhost:4200/crisis-list
```

Notice that the crisis-list component displays. Angular is using the route you defined to dynamically load the component. You can load the heroes-list component the same way:

```
content_copyhttp://localhost:4200/heroes-list
```

Control navigation with UI elements

Currently, your application supports two routes. However, the only way to use those routes is for the user to manually type the path in the browser's address bar. In this section, you'll add two links that users can click to navigate between the heroes-list and crisis-list components. You'll also add some CSS styles. While these styles are not required, they make it easier to identify the link for the currently-displayed component. You'll add that functionality in the next section.

1. Open the app.component.html file and add the following HTML below the title.
   src/app/app.component.html

```
content_copy<nav>
  <a class="button" routerLink="/crisis-list">Crisis Center</a> |
  <a class="button" routerLink="/heroes-list">Heroes</a>
</nav>
```

This HTML uses an Angular directive, routerLink. This directive connects the routes you defined to your template files.

2. Open the app.component.css file and add the following styles.

src/app/app.component.css

```
content_copy.button {
   box-shadow: inset 0 1px 0 0 #ffffff;
   background: #ffffff linear-gradient(to bottom, #ffffff 5%, #f6f6f6 100%);
   border-radius: 6px;
   border: 1px solid #dcdcdc;
   display: inline-block;
   cursor: pointer;
   color: #666666;
   font-family: Arial, sans-serif;
   font-size: 15px;
   font-weight: bold;
   padding: 6px 24px;
   text-decoration: none;
   text-shadow: 0 1px 0 #ffffff;
   outline: 0;
}
.activebutton {
   box-shadow: inset 0 1px 0 0 #dcecfb;
   background: #bddbfa linear-gradient(to bottom, #bddbfa 5%, #80b5ea 100%);
   border-radius: 6px;
   border: 1px solid #84bbf3;
   display: inline-block;
   cursor: pointer;
   color: #ffffff;
   font-family: Arial, sans-serif;
   font-size: 15px;
   font-weight: bold;
   padding: 6px 24px;
   text-decoration: none;
   text-shadow: 0 1px 0 #528ecc;
   outline: 0;
}
```

If you view your application in the browser, you should see these two links. When you click on a link, the corresponding component appears.

Identify the active route

While users can navigate your application using the links you added in the previous section, they don't have an easy way to identify what the active route is. You can add this functionality using Angular's routerLinkActive directive.

1. From your code editor, open the app.component.html file.
2. Update the anchor tags to include the routerLinkActive directive.

src/app/app.component.html

```
content_copy<nav>
   <a class="button" routerLink="/crisis-list" routerLinkActive="activebutton">Crisis
Center</a> |
   <a class="button" routerLink="/heroes-list"
routerLinkActive="activebutton">Heroes</a>
</nav>
```

View your application again. As you click one of the buttons, the style for that button updates automatically, identifying the active component to the user. By adding the routerLinkActive directive, you inform your application to apply a specific CSS class to the active route. In this tutorial, that CSS class is activebutton, but you could use any class that you want.

Adding a redirect

In this step of the tutorial, you add a route that redirects the user to display the /heroes-list component.

1. From your code editor, open the app.module.ts file.
2. In the imports array, update the RouterModule section as follows.

   src/app/app.module.ts

   ```
   content_copyimports: [
     BrowserModule,
     RouterModule.forRoot([
       {path: 'crisis-list', component: CrisisListComponent},
       {path: 'heroes-list', component: HeroesListComponent},
       {path: '', redirectTo: '/heroes-list', pathMatch: 'full'},
     ]),
   ],
   ```

   Notice that this new route uses an empty string as its path. In addition, it replaces the component property with two new ones:

   - redirectTo. This property instructs Angular to redirect from an empty path to the heroes-list path.
   - pathMatch. This property instructs Angular on how much of the URL to match. For this tutorial, you should set this property to full. This strategy is recommended when you have an empty string for a path. For more information about this property, see the Route API documentation.

Now when you open your application, it displays the heroes-list component by default.

Adding a 404 page

It is possible for a user to try to access a route that you have not defined. To account for this behavior, the best practice is to display a 404 page. In this section, you'll create a 404 page and update your route configuration to show that page for any unspecified routes.

1. From the terminal, create a new component, PageNotFound.

   ```
   content_copyng generate component page-not-found
   ```

2. From your code editor, open the page-not-found.component.html file and replace its contents with the following HTML.

   src/app/page-not-found/page-not-found.component.html

   ```
   content_copy<h2>Page Not Found</h2>
   <p>We couldn't find that page! Not even with x-ray vision.</p>
   ```

3. Open the app.module.ts file. In the imports array, update the RouterModule section as follows.

   src/app/app.module.ts

   ```
   content_copyimports: [
     BrowserModule,
     RouterModule.forRoot([
       {path: 'crisis-list', component: CrisisListComponent},
       {path: 'heroes-list', component: HeroesListComponent},
       {path: '', redirectTo: '/heroes-list', pathMatch: 'full'},
       {path: '**', component: PageNotFoundComponent}
     ]),
   ],
   ```

   The new route uses a path, **. This path is how Angular identifies a wildcard route. Any route that does not match an existing route in your configuration will use this route.

   Notice that the wildcard route is placed at the end of the array. The order of your routes is important, as Angular applies routes in order and uses the first match it finds.

Try navigating to a non-existing route on your application, such as http://localhost:4200/powers. This route doesn't match anything defined in your app.module.ts file. However, because you defined a wildcard route, the application automatically displays your PageNotFound component.

Tutorial: Creating custom route matches

The Angular Router supports a powerful matching strategy that you can use to help users navigate your application. This matching strategy supports static routes, variable routes with parameters, wildcard routes, and so on. You can also build your own custom pattern matching for situations in which the URLs are more complicated.
In this tutorial, you'll build a custom route matcher using Angular's UrlMatcher. This matcher looks for a Twitter handle in the URL.
For a working example of the final version of this tutorial, see the live example / download example.

Create a sample application

Using the Angular CLI, create a new application, *angular-custom-route-match*. In addition to the default Angular application framework, you will also create a *profile* component.

1. Create a new Angular project, *angular-custom-route-match*.
   ```
   content_copyng new angular-custom-route-match
   ```
   When prompted with Would you like to add Angular routing?, select Y.
   When prompted with Which stylesheet format would you like to use?, select CSS.
   After a few moments, a new project, angular-custom-route-match, is ready.
2. From your terminal, navigate to the angular-custom-route-match directory.
3. Create a component, *profile*.
   ```
   content_copyng generate component profile
   ```
4. In your code editor, locate the file, profile.component.html and replace the placeholder content with the following HTML.
   src/app/profile/profile.component.html
   ```
   content_copy<p>
   Hello {{ username$ | async }}!
   </p>
   ```
5. In your code editor, locate the file, app.component.html and replace the placeholder content with the following HTML.
   src/app/app.component.html
   ```
   content_copy<h2>Routing with Custom Matching</h2>

   Navigate to <a routerLink="/@Angular">my profile</a>

   <router-outlet></router-outlet>
   ```

Configure your routes for your application

With your application framework in place, you next need to add routing capabilities to the app.module.ts file. As a part of this process, you will create a custom URL matcher that looks for a Twitter handle in the URL. This handle is identified by a preceding @ symbol.

1. In your code editor, open your app.module.ts file.
2. Add an import statement for Angular's RouterModule and UrlMatcher.
   src/app/app.module.ts
   ```
   content_copyimport { RouterModule, UrlSegment } from '@angular/router';
   ```
3. In the imports array, add a RouterModule.forRoot([]) statement.
   src/app/app.module.ts
   ```
   content_copy@NgModule({
     imports:      [
       BrowserModule,
       FormsModule,
       RouterModule.forRoot([
         {
   /* . . . */
       ])],
     declarations: [ AppComponent, ProfileComponent ],
     bootstrap:    [ AppComponent ]
   })
   ```

4. Define the custom route matcher by adding the following code to the RouterModule.forRoot() statement.
   src/app/app.module.ts
```
content_copymatcher: (url) => {
  if (url.length === 1 && url[0].path.match(/^@[\w]+$/gm)) {
    return {
      consumed: url,
      posParams: {
        username: new UrlSegment(url[0].path.substr(1), {})
      }
    };
  }

  return null;
},
component: ProfileComponent
}
```
This custom matcher is a function that performs the following tasks:

- The matcher verifies that the array contains only one segment.
- The matcher employs a regular expression to ensure that the format of the username is a match.
- If there is a match, the function returns the entire URL, defining a username route parameter as a substring of the path.
- If there isn't a match, the function returns null and the router continues to look for other routes that match the URL.

A custom URL matcher behaves like any other route definition. You can define child routes or lazy loaded routes as you would with any other route.

Subscribe to the route parameters

With the custom matcher in place, you now need to subscribe to the route parameters in the profile component.

1. In your code editor, open your profile.component.ts file.
2. Add an import statement for Angular's ActivatedRoute and ParamMap.
   src/app/profile/profile.component.ts
```
content_copyimport { ActivatedRoute, ParamMap } from '@angular/router';
```
3. Add an import statement for RxJS map.
   src/app/profile/profile.component.ts
```
content_copyimport { map } from 'rxjs/operators';
```
4. Subscribe to the username route parameter.
   src/app/profile/profile.component.ts
```
content_copyusername$ = this.route.paramMap
  .pipe(
    map((params: ParamMap) => params.get('username'))
  );
```
5. Inject the ActivatedRoute into the component's constructor.
   src/app/profile/profile.component.ts
```
content_copyconstructor(private route: ActivatedRoute) { }
```

Test your custom URL matcher

With your code in place, you can now test your custom URL matcher.

1. From a terminal window, run the ng serve command.
```
content_copyng serve
```
2. Open a browser to http://localhost:4200.
   You should see a single web page, consisting of a sentence that reads Navigate to my profile.
3. Click the my profile hyperlink.
   A new sentence, reading Hello, Angular! appears on the page.

Router Reference

The following sections highlight some core router concepts.

Router imports

The Angular Router is an optional service that presents a particular component view for a given URL. It is not part of the Angular core and thus is in its own library package, @angular/router.

Import what you need from it as you would from any other Angular package.

src/app/app.module.ts (import)

```
content_copyimport { RouterModule, Routes } from '@angular/router';
```

For more on browser URL styles, see LocationStrategy and browser URL styles.

Configuration

A routed Angular application has one singleton instance of the Router service. When the browser's URL changes, that router looks for a corresponding Route from which it can determine the component to display.

A router has no routes until you configure it. The following example creates five route definitions, configures the router via the RouterModule.forRoot() method, and adds the result to the AppModule's imports array.

src/app/app.module.ts (excerpt)

```
content_copyconst appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'hero/:id',      component: HeroDetailComponent },
  {
    path: 'heroes',
    component: HeroListComponent,
    data: { title: 'Heroes List' }
  },
  { path: '',
    redirectTo: '/heroes',
    pathMatch: 'full'
  },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
    // other imports here
  ],
  ...
})
export class AppModule { }
```

The appRoutes array of routes describes how to navigate. Pass it to the RouterModule.forRoot() method in the module imports to configure the router.

Each Route maps a URL path to a component. There are no leading slashes in the path. The router parses and builds the final URL for you, which allows you to use both relative and absolute paths when navigating between application views.

The :id in the second route is a token for a route parameter. In a URL such as /hero/42, "42" is the value of the id parameter. The corresponding HeroDetailComponent uses that value to find and present the hero whose id is 42.

The data property in the third route is a place to store arbitrary data associated with this specific route. The data property is accessible within each activated route. Use it to store items such as page titles, breadcrumb text, and other read-only, static data. You can use the resolve guard to retrieve dynamic data.
The empty path in the fourth route represents the default path for the application—the place to go when the path in the URL is empty, as it typically is at the start. This default route redirects to the route for the /heroes URL and, therefore, displays the HeroesListComponent.
If you need to see what events are happening during the navigation lifecycle, there is the enableTracing option as part of the router's default configuration. This outputs each router event that took place during each navigation lifecycle to the browser console. Use enableTracing only for debugging purposes. You set the enableTracing: true option in the object passed as the second argument to the RouterModule.forRoot() method.

Router outlet

The RouterOutlet is a directive from the router library that is used like a component. It acts as a placeholder that marks the spot in the template where the router should display the components for that outlet.

```
content_copy<router-outlet></router-outlet>
<!-- Routed components go here -->
```

Given the configuration above, when the browser URL for this application becomes /heroes, the router matches that URL to the route path /heroes and displays the HeroListComponent as a sibling element to the RouterOutlet that you've placed in the host component's template.

Router links

To navigate as a result of some user action such as the click of an anchor tag, use RouterLink.
Consider the following template:
src/app/app.component.html

```
content_copy<h1>Angular Router</h1>
<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
  <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
</nav>
<router-outlet></router-outlet>
```

The RouterLink directives on the anchor tags give the router control over those elements. The navigation paths are fixed, so you can assign a string to the routerLink (a "one-time" binding).
Had the navigation path been more dynamic, you could have bound to a template expression that returned an array of route link parameters; that is, the link parameters array. The router resolves that array into a complete URL.

Active router links

The RouterLinkActive directive toggles CSS classes for active RouterLink bindings based on the current RouterState.
On each anchor tag, you see a property binding to the RouterLinkActive directive that looks like routerLinkActive="...".
The template expression to the right of the equal sign, =, contains a space-delimited string of CSS classes that the Router adds when this link is active (and removes when the link is inactive). You set the RouterLinkActive directive to a string of classes such as [routerLinkActive]="'active fluffy'" or bind it to a component property that returns such a string.
Active route links cascade down through each level of the route tree, so parent and child router links can be active at the same time. To override this behavior, you can bind to the [routerLinkActiveOptions] input binding with the { exact: true } expression. By using { exact: true }, a given RouterLink will only be active if its URL is an exact match to the current URL.

Router state

After the end of each successful navigation lifecycle, the router builds a tree of ActivatedRoute objects that make up the current state of the router. You can access the current RouterState from anywhere in the application using the Router service and the routerState property.
Each ActivatedRoute in the RouterState provides methods to traverse up and down the route tree to get information from parent, child, and sibling routes.

Activated route

The route path and parameters are available through an injected router service called the ActivatedRoute. It has a great deal of useful information including:

Property	Description
url	An Observable of the route path(s), represented as an array of strings for each part of the route path.
data	An Observable that contains the data object provided for the route. Also contains any resolved values from the resolve guard.
paramMap	An Observable that contains a map of the required and optional parameters specific to the route. The map supports retrieving single and multiple values from the same parameter.
queryParamMap	An Observable that contains a map of the query parameters available to all routes. The map supports retrieving single and multiple values from the query parameter.
fragment	An Observable of the URL fragment available to all routes.
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	The route's parent ActivatedRoute when this route is a child route.
firstChild	Contains the first ActivatedRoute in the list of this route's child routes.
children	Contains all the child routes activated under the current route.

Two older properties are still available; however, their replacements are preferable as they may be deprecated in a future Angular version.

- params: An Observable that contains the required and optional parameters specific to the route. Use paramMap instead.

- queryParams: An Observable that contains the query parameters available to all routes. Use queryParamMap instead.

Router events

During each navigation, the Router emits navigation events through the Router.events property. These events range from when the navigation starts and ends to many points in between. The full list of navigation events is displayed in the table below.

Router Event	**Description**
NavigationStart	An event triggered when navigation starts.
RouteConfigLoadStart	An event triggered before the Router lazy loads a route configuration.
RouteConfigLoadEnd	An event triggered after a route has been lazy loaded.
RoutesRecognized	An event triggered when the Router parses the URL and the routes are recognized.
GuardsCheckStart	An event triggered when the Router begins the Guards phase of routing.
ChildActivationStart	An event triggered when the Router begins activating a route's children.
ActivationStart	An event triggered when the Router begins activating a route.
GuardsCheckEnd	An event triggered when the Router finishes the Guards phase of routing successfully.
ResolveStart	An event triggered when the Router begins the Resolve phase of routing.
ResolveEnd	An event triggered when the Router finishes the Resolve phase of routing successfuly.
ChildActivationEnd	An event triggered when the Router finishes activating a route's children.
ActivationEnd	An event triggered when the Router finishes activating a route.

NavigationEnd	An event triggered when navigation ends successfully.
NavigationCancel	An event triggered when navigation is canceled. This can happen when a Route Guard returns false during navigation, or redirects by returning a UrlTree.
NavigationError	An event triggered when navigation fails due to an unexpected error.
Scroll	An event that represents a scrolling event.

When you enable the enableTracing option, Angular logs these events to the console. For an example of filtering router navigation events, see the router section of the Observables in Angular guide.

Router terminology

Here are the key Router terms and their meanings:

Router Part	**Meaning**
Router	Displays the application component for the active URL. Manages navigation from one component to the next.
RouterModule	A separate NgModule that provides the necessary service providers and directives for navigating through application views.
Routes	Defines an array of Routes, each mapping a URL path to a component.
Route	Defines how the router should navigate to a component based on a URL pattern. Most routes consist of a path and a component type.
RouterOutlet	The directive (<router-outlet>) that marks where the router displays a view.
RouterLink	The directive for binding a clickable HTML element to a route. Clicking an element with a routerLink directive that is bound to a *string* or a *link parameters array* triggers a navigation.
RouterLinkActive	The directive for adding/removing classes from an HTML element when an

	associated routerLink contained on or inside the element becomes active/inactive.
ActivatedRoute	A service that is provided to each route component that contains route specific information such as route parameters, static data, resolve data, global query params, and the global fragment.
RouterState	The current state of the router including a tree of the currently activated routes together with convenience methods for traversing the route tree.
Link parameters array	An array that the router interprets as a routing instruction. You can bind that array to a RouterLink or pass the array as an argument to the Router.navigate method.
Routing component	An Angular component with a RouterOutlet that displays views based on router navigations.

What are Modules in Angular?

In Angular, modules are a way of grouping and organizing code related to a specific functionality or feature separate from other code. In every Angular application, there is atleast one root module called the NgModules, conventionally named as AppModule in the app.module.ts file. NgModules are containers for a logical unit of code dedicated to a specific application domain, workflow, or set of capabilities. NgModules can contain components, service providers and other code files. The NgModules can import functionality from other NgModules as well as export functionality for usage by other NgModules. It is important to note that, an Angular application starts by bootstrapping the root NgModules.

The definition of the root NgModule looks like the following:

src\app\app.module.ts

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

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

As you can see in the above root NgModule, the class is decorated with @NgModule. The @NgModule decorator is a function that takes a single metadata object. The properties of the metadata object describe the module. The following are important properties:

- **declarations** - The declarations property is used to declare an array of components, directives, and pipes that belong to the NgModule. Their scope is limited to within the module until you export them in the exports array of NgModule.
- **exports** - The exports property is used to make the subsets of declarations visible and usable in the component templates of other NgModules.
- **imports** - The imports property is used to import an array of other modules whose classes are required by component templates that are declared in this MgModule.
- **providers** - The providers property is used to declare an array of services to make them available for dependency injection throughout the application. The providers can also be specified at the component level.
- **bootstrap** - The bootstrap property is used to tell Angular which component view to load as the top-level component when the application starts. The bootstrap property should only be set by the root NgModule.

A root NgModule always has one root component that is generated during project creation. However, any NgModule can have any number of extra components, which can be loaded through the router or created through the template.

Creating Modules in Angular

A small Angular application can have one as well as many feature NgModules. A feature module is a custom module. The root NgModule can include child NgModules in a hierarchy of any depth.

A module in Angular can be created using the Angular CLI command as follows:

```
ng generate module module-name
```

Why do we need Feature Modules?

Feature modules is needed to make the application code easier to read by breaking it down into functions that each deal with a single part of the overall functionality. It can greatly reduce the size and complexity of project files. For example, if we have all the components, directives, pipes, and services of the application in the root app.module.ts module then it will become unmanageable. Splitting the code into separate feature modules help to make the code manageable.

Example:

If we require a feature for user authentication and another for products management in our application, we can create two modules with the following steps:

1. Create user-auth module:

```
ng generate module user-auth
```

The above command will generate src/app/user-auth/user-auth.module.ts file as follows:

src/app/user-auth/user-auth.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  declarations: [],
  imports: [
    CommonModule
  ]
})
export class UserAuthModule { }
```

2. Create product module:

```
ng generate module product
```

The above command will generate src/app/user-auth/product.module.ts file as follows:

src/app/product/product.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  declarations: [],
  imports: [
    CommonModule
  ]
})
export class ProductModule { }
```

How to Create and Use Services in Angular?

In Angular, a service is typically a class that contains business logic, models, data, and functions that can be invoked from any component of an Angular application.

Angular separates components from services to increase modularity and reusability. By separating business processing logic from view-related functions of a component, the component classes becomes simple and efficient.

Why Services?

Components should never directly fetch or save data. They should concentrate on data presentation and delegate data access to a service.

Services make sharing of information among classes that do not know each other easier.

By removing data access from components, we can change the implementation at any time without having to touch any components. They have no idea how the service operates.

Creating a Service

To create a service in Angular, you can use the Angular CLI command as shown in the example below:

```
ng generate service User
```

The above command will generate a skeleton **UserService** class in src/app/user.service.ts as follows:

src\app\user.service.ts

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


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

  constructor() { }
}
```