Angular 6 Features:

Separation of code (templates, components and modules)

multiple programmers can work at a time

Unit testing with karma, jasmine Libraries

Single page application developmentby routing

Dependency injection by services

Two-way data binding

Partial pages with Directives

Easy to learn and develop and better performance (This has been done by automatically adding or removing reflect-metadata from polyfills.ts file which makes application smaller in production)

Cross browser support

Mobile first approach, used for develop mobile based apps

Strongly typed or static typing approach or typescript support (compile time data type and supports object oriented features)

Components based -Use of component helps in creating loosely coupled units of application which can be developed, maintained and tested easily.

Angular Js Features:

Separation of code (MVC)

multiple programmers can work at a time

Unit testing with karma, jasmine Libraries

Single page application development by routing

Dependency injection by injecting services

Two-way data binding by ng-model

Partial pages with Directives

Easy to learn and develop

Html as templates

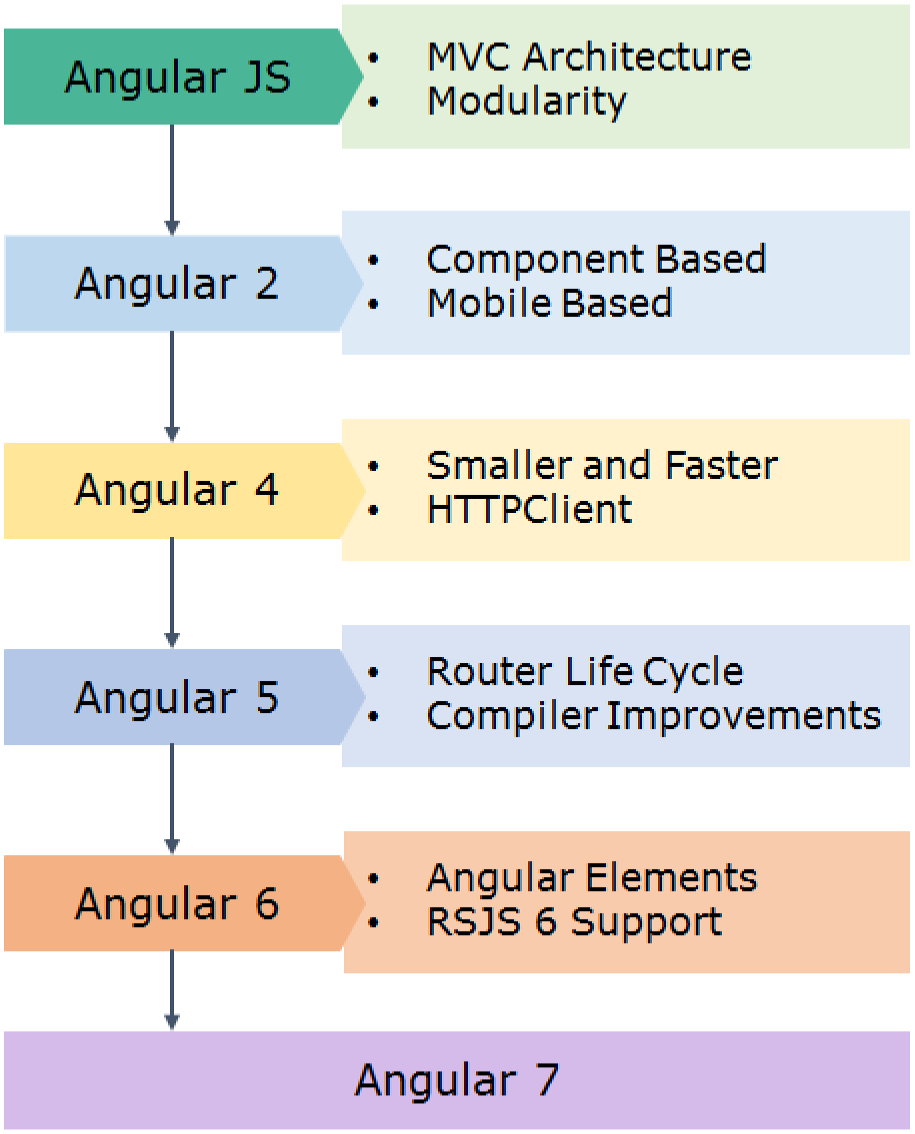
Other Points:

Angular is completely rewritten and is not an upgrade to Angular 1

Angular 1 will find it easy to migrate to Angular

Inside a component, we write both business logic and view.

Developers prefer Typescript to write Angular code. But other than Typescript, we can also write code using JavaScript (ES5 or ECMAScript 5)



Angular CLI:

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

```
ng new my-dream-app
```

```
cd my-dream-app
```

```
ng serve
```

File / Folder	**Purpose**
e2e/	This folder contains all End-to-End (e2e) tests of the application written in Jasmine and run by the Protractor
node_modules/	Node.js creates this folder and puts all npm modules installed as listed in package.json
src/	All application related files will be stored inside it
angular.json	Configuration file for Angular CLI where we set several defaults and also configure what files to be included during project build
package.json	This is node configuration file which contains all dependencies required for Angular
tsconfig.json	This is Typescript configuration file where we can configure compiler options
tslint.json	This file contains linting rules preferred by Angular style guide

Npm install	- install the npm packages listed in package.json
Npm serve - - open	- Run the application. This will open a browser with default port as 4200.
Npm serve - - open - - port 3000	-custom port options
Npm Start	

Modules and components:

Modules:

Used to **organize the application**. Angular applications are collection of modules

A module in Angular is a class with **@NgModule** decorator added to it. @NgModule metadata will contain the declarations of components, pipes, directives, services which are to be used across the application.

Every Angular application should have one root module which is loaded first to launch the application.

Root Module:

In **app.module.ts** file placed under app folder, we have the following code

```
import{ BrowserModule } from'@angular/platform-browser';
```

```
import{ NgModule } from'@angular/core';
import{ AppComponent } from'./app.component';
@NgModule({
  declarations: [
AppComponent
  ],
  imports: [
BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
exportclassAppModule{ }
```

Line 1: imports BrowserModule class which is needed to run application inside browser

Line 2: imports NgModule class to define metadata of the module

Line 4: imports AppComponent class from app.component.ts file. No need to mention .ts extension as Angular by default considers the file as .ts file

Line 7: declarations property should contain all user defined components, directives, pipes classes to be used across the application. We have added our AppComponent class here

Line 10: imports property should contain all module classes to be used across the application

Line 13: providers property should contains all service classes

Line 14: bootstrap declaration should contain the root component to load. In this example, AppComponent is the root component which will be loaded in the html page

Bootstrapping Root Module

in **main.ts** file placed under src folder, observe the following code

```
import{ enableProdMode } from'@angular/core';
import{ platformBrowserDynamic } from'@angular/platform-browser-dynamic';
import{ AppModule } from'./app/app.module';
import{ environment } from'./environments/environment';
if (environment.production) {
enableProdMode();
}
platformBrowserDynamic().bootstrapModule(AppModule);
```

Line 1 : Imports enableProdMode from the core module

Line 2: Import platformBrowserDynamic class which is used to compile the application based on the browser platform

Line 4: Import AppModule which is the root module to bootstrap

Line 5: imports environment which is used to check whether the type of environment is production or development

Line 7: Checks if we are working in production environment or not

Line 8: enableProdMode() will enable production mode which will run application faster

Line 11: bootstrapModule() method accepts root module name as parameter which will load the given module i.e., AppModule after compilation

Loading root component in HTML Page:

Open **index.html** under src folder

```
<!doctype html>
<htmllang="en">
<head>
<metacharset="utf-8">
<title>MyApp</title>
<basehref="/">
<metaname="viewport"content="width=device-width, initial-scale=1">
<linkrel="icon"type="image/x-icon"href="favicon.ico">
</head>
<body>
<app-root></app-root>
</body>
</html>
```

Line 12 loads the root component in the html page. app-root is the selector name we have given to the component. This will execute the component and renders the template inside browser.

Creating Component:

Ng generate component hello   -for create component with ng cli

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

```
exportclassHelloComponentimplementsOnInit {
courseName: string = "Angular";
constructor() { }
ngOnInit() {
  }
}
```

Templates:

Creating Templates:

The default language for templates is HTML

Templates in Angular represents a view whose role is to display data and change the data whenever an event occurs

We can define templates in 2 ways:

Inline template:

```
import{ Component } from'@angular/core';
@Component({
  selector: 'app-root',
  template: `
<h1> Welcome </h1>
<h2> Course Name: {{ courseName }}</h2>
     `,
styleUrls: ['./app.component.css']
})
exportclassAppComponent {
courseName: string = "Angular";
}
```

External template :

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

@Component({
  selector: 'app-root',
templateUrl:'./app.component.html',
styleUrls: ['./app.component.css']
})
exportclassAppComponent {
courseName: string = "Angular";
}
```

{ }   - one way data binding {demo}

( )   -call events (click)="changeName()"

Zones informs Angular about the changes in the application. It automatically detects all asynchronous actions at run time in the application

Directives:

Ng-If:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  isAuthenticated: boolean;
  submitted: boolean = false;
  userName: string;
  onSubmit(name: string, password: string) {
    this.submitted = true;
    this.userName = name;
    if (name === "admin" && password === "admin")
      this.isAuthenticated = true;
    else
      this.isAuthenticated = false;
  }
}
```

```
<div *ngIf="!submitted">
  <form>
    <label>User Name</label>
    <input type="text" #username><br/><br/>
    <label for="password">Password</label>
    <input type="password" name="password" #password><br/>
  </form>
  <button (click)="onSubmit(username.value,password.value)">Login</button>
</div>
<div *ngIf="submitted">
  <div *ngIf="isAuthenticated; else failureMsg">
    <h4> Welcome {{userName}} </h4>
  </div>
  <ng-template #failureMsg>
    <h4> Invalid Login !!! Please try again...</h4>
  </ng-template>
  <button type="button" (click)="submitted=false">Back</button>
</div>
```

Ng-For:

ngFor directive is used to iterate over collection of data i.e., arrays

```
export class AppComponent {
    courses: any[] = [
      { id: 1, name: "TypeScript" },
      { id: 2, name: "Angular" },
      { id: 3, name: "Node JS" },
      { id: 1, name: "TypeScript" }
    ];
  }


<ul>
    <li *ngFor="let course of courses;  let i = index">
         {{i}} - {{ course.name }}
    </li>
  </ul>
```

Here : { i }} displays the index of each course and course.name

Ng-switch:

```
export class AppComponent {
  value: number = 0;
  nextChoice() {
    this.value++;
  }

<div [ngSwitch]="value">
  <p *ngSwitchCase="1">First Choice</p>
  <p *ngSwitchCase="2">Second Choice</p>
  <p *ngSwitchCase="3">Third Choice</p>
  <p *ngSwitchCase="2">Second Choice Again</p>
  <p *ngSwitchDefault>Default Choice</p>
</div>
```

Custom structural directives:

We can create custom structural directive when there is no built-in directive available for required functionality.

To create a custom structural directive, we need to create a class annotated with @Directive

It also adds repeat directive to the root module i.e., app.module.ts to make it available to the entire module

```
import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';
@Directive({
  selector: '[appRepeat]'
})
export class RepeatDirective {
  constructor(private _templateRef: TemplateRef<any>, private _viewContainer:
ViewContainerRef) { }
  @Input() set appRepeat(count: number) {
    for (var i = 0; i < count; i++) {
      this._viewContainer.createEmbeddedView(this._templateRef);
    }
  }
}
```

Line 3: Annotate the class with @Directive which represents the class as a directive and specify the selector name inside brackets

Line 8: Create a constructor and inject two classes called TemplateRef which acquires <ng-template> content and another class called ViewcontainerRef which access the html container to add or remove elements from it

Line 10: Create a setter method for appRepeat directive by attaching @Input() decorator which specifies that this directive will receive value from the component. This method takes the number passed to appRepeat directive as an argument.

Line 12:As we need to render the elements based on the number passed to the appRepeat directive, run a for loop in which pass the template reference to createEmbeddedView method which renders the elements into the DOM. This structural directive creates an embedded view from the Angular generated <ng-template> and inserts that view in a view container.

```
<h3>Structural Directive</h3>
<p *appRepeat="5">I am being repeated...</p>

--o/p repeate 5 time bove stmnt
```

Line 2: appRepeat directive is applied to the paragraph element. It will render five paragraph elements into the DOM

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { RepeatDirective } from './repeat.directive';
@NgModule({
  declarations: [
    AppComponent,
    RepeatDirective
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Custom structural directives: Exportas:

exportAs property is used to define a name for a directive using which we can access directive class properties and methods in a component template

exportAs property can be applied to components and directives

```
import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';
@Directive({
  selector: '[appRepeat]',
  exportAs: 'repeat,changeText'
})
export class RepeatDirective {
  constructor(private _templateRef: TemplateRef<any>, private _viewContainer:
ViewContainerRef) {}
  repeatElement(count: number) {
    for (var i = 0; i < count; i++) {
      this._viewContainer.createEmbeddedView(this._templateRef);
    }
  }
  changeElementText(count:number) {
    for(var i=0;i<5;i++)
     document.getElementsByTagName("p").item(i).innerHTML = "Text is changed..."
  }
}
```

```
<h3>Structural Directive with exportAs property</h3>
<ng-template appRepeat #rd="repeat" #ct="changeText">
    <p>I am being repeated...</p>
</ng-template>
<button (click)="rd.repeatElement(5)">Repeat Element</button>
<button (click)="ct.changeElementText(5)">Change Text</button>
```

Attribute Directive:

Attribute directives changes the appearance / behavior of a component / element
Following are built-in attribute directives

ngStyle: directive is used to modify a component / element's style.
ngClass: It allows us to dynamically set and change the CSS classes for a given DOM element.

NgStyle:

```
export class AppComponent {
  colorName: string = 'yellow';
  color: string = 'red';
}
```

```
<div [style.background-color]="colorName" [style.color]="color">
    Uses fixed yellow background
  </div>
```

If there are more than one css styles to apply, we can use **ngStyle** attribute.

```
export class AppComponent {
  colorName: string = 'red';
  fontWeight: string = 'bold';
  borderStyle: string = '1px solid black';
}
```

```
<p [ngStyle]="{
    color:colorName,
    'font-weight':fontWeight,
    borderBottom: borderStyle
}">
Demo for attribute directive ngStyle
</p>
```

NgClass:

```
export class AppComponent {
  isBordered: boolean = true;
}

<div [class.bordered]="isBordered">
    Border {{ isBordered ? "ON" : "OFF" }}
  </div>

.bordered {
    border: 1px dashed black;
    background-color: #eee;
}
```

Custom attribute directive:

We can create custom attribute directive when there is no built-in directive available for the required functionality

It also adds message directive to the root module i.e., **app.module.ts** to make it available to the entire module

```
import { Directive, ElementRef, Renderer2, HostListener, Input } from
'@angular/core';
@Directive({
  selector: '[appMessage]'
})
export class MessageDirective {
  @Input('appMessage') message: string;
  constructor(private el: ElementRef, private renderer: Renderer2) {
    renderer.setStyle(el.nativeElement, 'cursor', 'pointer');
  }
  @HostListener('click') onClick() {
    this.el.nativeElement.innerHTML = this.message;
    this.renderer.setStyle(this.el.nativeElement, 'color', 'red');
  }
}
```

Line 3: Create a directive class with the selector name as appMessage

Line 8: @Input('appMessage') will inject the value passed to 'appMessage' directive into the 'message' property

Line10: Use constructor injection to inject ElementRef which holds the html element reference in which directive is used and Renderer2 which is used to set the CSS styles.

Line 11: Using Renderer2 reference, we are changing the cursor to pointer symbol

Line 14-16: onClick method is invoked when click event is triggered on the directive which will display the message received and changes the element color to red

```
<h3>Attribute Directive</h3>
<p [appMessage]="myMessage">Click Here</p>
```

Line 2: Apply appMessage directive by assigning 'myMessage' property which will be sent to the directive on click of the paragraph

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  myMessage: string = "Hello, I am from attribute directive";
}
```

Data Bindings:

Property binding:

Binding property of element with class property

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  imgUrl: string = 'assets/imgs/logo.jpg';
}
```

```
<img [src]='imgUrl' width=200 height=100>
```

Attribute directives:

Property binding will not work for few elements/pure attributes like ARIA, SVG and table span. In such cases, we need to go for attribute binding.

Attribute binding can be used to **bind component property to the attribute directly**

```
export class AppComponent {
  value: string ="2";
}
```

```
<td [attr.colspan] = "value">Hello</td>
```

Two way data binding:

To make two way data binding i.e., [(ngModel)] work, import FormsModule class to the root module

```
export class AppComponent {
  name: string = "Angular";
}
```

```
<input [(ngModel)] = "course.courseName">
<input bindon-ngModel="course.courseName">
<input [ngModel]="course.courseName" (ngModelChange)="course.courseName=$event">
```

Pipes(filters):

Built in pipes:

```
<h3> {{ title | titlecase}} </h3>
<table style="text-align:left">
    <tr>
        <th> Product Code </th>
        <td> {{ productCode | lowercase }} </td>
    </tr>
    <tr>
        <th> Product Name </th>
        <td> {{ productName | uppercase }} </td>
    </tr>
</table>
```

A pipe can also have optional parameters to change the output. To pass parameters, after a pipe name add a colon(:) followed by the parameter value.

```
pipename : parametervalue1:parametervalue2
```

currency:

```
{{ expression | currency:currencyCode:symbol:digitInfo:locale }}

{{ 250000 | currency:'INR':'symbol':'6.3':'fr'}} will display 250 000,000 CA$
```

Date:

```
{{ expression | date:format:timezone:locale }}
{{"1/31/2018, 11:05:04 AM" | date:'fullDate':'0':'fr'}} will display mercredi 31
janvier 2018
```

Percent:

```
{{ expression | percent:digitInfo:locale }}
{{ 0.1 | percent:'2.2-3': 'fr' }} will display 10.00 %
```

Slice: This pipe can be used to extract subset of elements or characters from an array or string respectively.

```
{{ expression | slice:start:end }}
{{ 'Laptop Charger'| slice:-4:-2}} will display rg
```

Number:

```
{{ expression | number:digitInfo }}
{{ 25000 | number:'.3-5' }} will display 25,000.000
```

```
<h3> {{ title | titlecase}} </h3>
<table style="text-align:left">
    <tr>
        <th> Product Code </th>
        <td> {{ productCode | slice:5:9 }} </td>
    </tr>
    <tr>
        <th> Product Name </th>
        <td> {{ productName | uppercase }} </td>
    </tr>
    <tr>
        <th> Product Price </th>
        <td> {{ productPrice | currency: 'INR':'symbol':'':'fr' }} </td>
    </tr>
    <tr>
        <th> Purchase Date </th>
        <td> {{ purchaseDate | date:'fullDate' | lowercase}} </td>
    </tr>
    <tr>
        <th> Product Tax </th>
        <td> {{ productTax | percent : '.2' }} </td>
    </tr>
    <tr>
        <th> Product Rating </th>
        <td>{{ productRating | number:'1.3-5'}} </td>
    </tr>
</table>
```

Json:
This pipe can be used to displays the given expression in the form of JSON string. It is mostly for debugging.

```
{{ {'productId':1234, 'productName':'Samsung Mobile'} | json  }} will display
{"productId":1234, "productName":"Samsung Mobile"}
```

I18nplural:
This pipe can be used to map numeric values against an object containing different string values to be returned. It takes a numeric value as input and compares it with the values in an object and returns a string accordingly.

```
{{ expression | i18nplural:mappingObject }}

import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  productRatings: number[] = [4, 3, 2, 4, 1]
  productMapping = { '=4': ' # - Excellent', '=3': ' # - Good', '=2': '# -
Average', 'other': ' # - Very Bad' };
}
```

I18nselect:
This pipe is similar to i18nplural but evaluates a string value instead.

```
{{ expression | i18nselect:mappingObject }}
```

Displaying greeting message based on the language selected using i18nSelect pipe.

```
import { Component } from '@angular/core';
import { I18nSelectPipe } from '@angular/common';
@Component({
  selector: 'app-root',
  styleUrls: ['./app.component.css'],
  templateUrl: './app.component.html'
})
export class AppComponent {
  message: string;
  messageMap: any = { 'en': 'Good Morning', 'fr': 'Bonjour', 'es': 'Buenos dÃƒÂ-
as', 'de': 'Guten Morgen' };
}

<h2> Wish in Different Languages </h2>
Enter your Name <input type=text #name><br/><br/> Select language to display
<select #language (change)="message = language.value">
        <option value="en">English</option>
        <option value="fr">French</option>
        <option value="es">Spanish</option>
        <option value="de">German</option>
      </select><br/><br/>
<h3 *ngIf="message"> {{ message | i18nSelect:messageMap }}, {{ name.value }}
</h3>
```

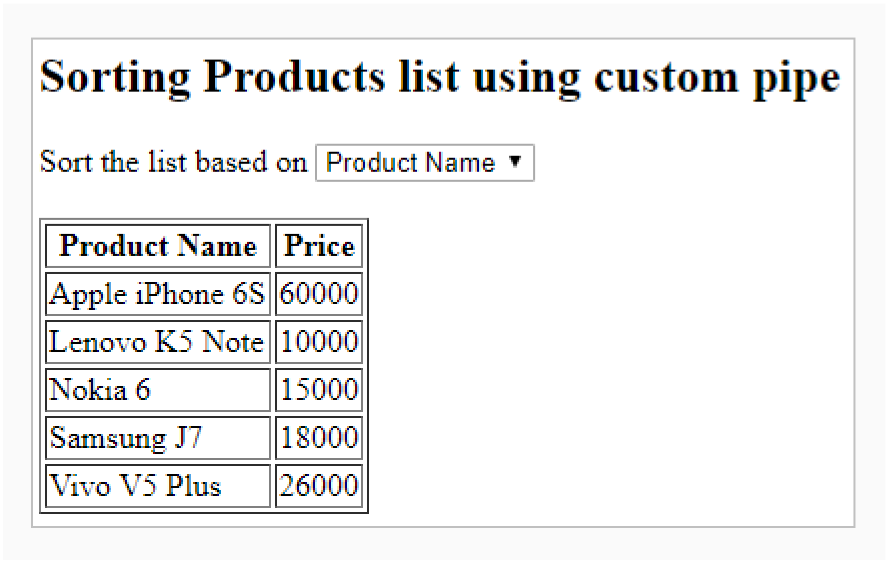
Custom pipe:
We can create our own custom pipe by inheriting PipeTransform interface

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'sort'
})
export class SortPipe implements PipeTransform {
  transform(value: string[], args?: string): string[] {
    if (args === "prodName") {
      return value.sort((a: any, b: any) => {
        if (a.productName < b.productName) {
          return -1;
        } else if (a.productName > b.productName) {
          return 1;
        } else {
          return 0;
        }
      });
    }
    else if (args === "price") {
      return value.sort((a: any, b: any) => {
        if (a.price < b.price) {
          return -1;
        } else if (a.price > b.price) {
          return 1;
        } else {
          return 0;
        }
      });
    }
    return value;
  }
}
```

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  sortoption: string = "";
  productsList = [
    { productName: "Samsung J7", price: 18000 },
    { productName: "Apple iPhone 6S", price: 60000 },
    { productName: "Lenovo K5 Note", price: 10000 },
    { productName: "Nokia 6", price: 15000 },
    { productName: "Vivo V5 Plus", price: 26000 }
  ];
}
```

```
<h2> Sorting Products list using custom pipe </h2>
Sort the list based on
<select [(ngModel)]="sortoption">
        <option value="prodName">Product Name</option>
        <option value="price">Price</option>
      </select><br/><br/>
<table border="1">
    <thead>
        <tr>
            <th>Product Name</th>
            <th>Price</th>
        </tr>
    </thead>
    <tbody>
        <tr *ngFor="let products of productsList | sort:sortoption">
            <td>{{products.productName}}</td>
            <td>{{products.price}}</td>
        </tr>
    </tbody>
</table>
```

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { SortPipe } from './sort.pipe';
@NgModule({
  declarations: [
    AppComponent,
    SortPipe
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```



Sorting Products list using custom pipe

Sort the list based on

Product Name	Price
Apple iPhone 6S	60000
Lenovo K5 Note	10000
Nokia 6	15000
Samsung J7	18000
Vivo V5 Plus	26000

Nested Components and sharing data between them:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { CoursesListComponent } from './courses-list/courses-list.component';
@NgModule({
  declarations: [
    AppComponent,
    CoursesListComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }


import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-courses-list',
  templateUrl: './courses-list.component.html',
  styleUrls: ['./courses-list.component.css']
})
export class CoursesListComponent {
  courses = [
    { courseId: 1, courseName: "Node JS" },
    { courseId: 2, courseName: "Typescript" },
    { courseId: 3, courseName: "Angular" },
    { courseId: 4, courseName: "React JS" }
  ];
}
```

```
<table border="1">
    <thead>
      <tr>
        <th>Course ID</th>
        <th>Course Name</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let course of courses">
        <td>{{course.courseId}}</td>
        <td>{{course.courseName}}</td>
      </tr>
    </tbody>
  </table>
```

```
<h2> Popular Courses </h2>
<button (click)="show=true">View Courses list</button><br/><br/>
<div *ngIf="show">
    <app-courses-list></app-courses-list>
</div>
```

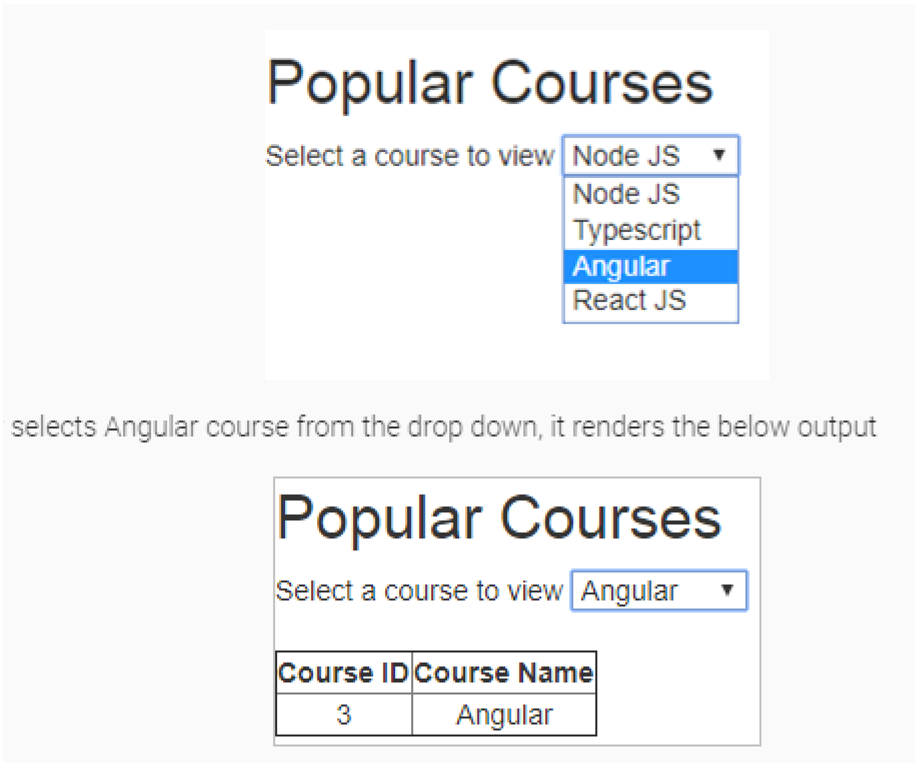
Neasted component : exportas

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-courses-list',
  templateUrl: './courses-list.component.html',
  styleUrls: ['./courses-list.component.css'],
  exportAs: 'courselist'
})
export class CoursesListComponent {
  courses = [
    { courseId: 1, courseName: "Node JS" },
    { courseId: 2, courseName: "Typescript" },
    { courseId: 3, courseName: "Angular" },
    { courseId: 4, courseName: "React JS" }
  ];
  course: any[];
  changeCourse(name: string) {
    this.course = [];
    for (var i = 0; i < this.courses.length; i++) {
      if (this.courses[i].courseName === name) {
        this.course.push(this.courses[i]);
      }
    }
  }
}
```

```
<table border="1" *ngIf="course">
    <thead>
      <tr>
        <th>Course ID</th>
        <th>Course Name</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let c of course">
        <td>{{c.courseId}}</td>
        <td>{{c.courseName}}</td>
      </tr>
    </tbody>
  </table>
```

```
<h2> Popular Courses </h2>
Select a course to view
<select #course (change)="cl.changeCourse(course.value)">
    <option value="Node JS">Node JS</option>
    <option value="Typescript">Typescript</option>
    <option value="Angular">Angular</option>
    <option value="React JS">React JS</option>
</select>
```

```
<br/>
<br/>
<app-courses-list #cl="courselist"></app-courses-list>
```



Popular Courses

Select a course to view Node JS

Node JS
Typescript
Angular
React JS

selects Angular course from the drop down, it renders the below output

Popular Courses

Select a course to view Angular

Course ID	Course Name
3	Angular

Passing data from parent to child:

We can use @Input decorator in the child component on any property type like arrays, objects etc.

```
<h2> Course Details </h2>
Select a course to view <select #course (change)="name = course.value">
    <option value="Node JS">Node JS</option>
    <option value="Typescript">Typescript</option>
    <option value="Angular">Angular</option>
    <option value="React JS">React JS</option>
    </select><br/><br/>
<app-courses-list [cName]="name"></app-courses-list>


import { Component, Input } from '@angular/core';
@Component({
  selector: 'app-courses-list',
  templateUrl: './courses-list.component.html',
  styleUrls: ['./courses-list.component.css']
})
export class CoursesListComponent {
  courses = [
    { courseId: 1, courseName: "Node JS" },
    { courseId: 2, courseName: "Typescript" },
    { courseId: 3, courseName: "Angular" },
    { courseId: 4, courseName: "React JS" }
  ];
  course: any[];
  @Input() set cName(name: string) {
    this.course = [];
    for (var i = 0; i < this.courses.length; i++) {
      if (this.courses[i].courseName === name) {
        this.course.push(this.courses[i]);
      }
    }
  }
}
```

```
<table border="1" *ngIf="course.length>0">
    <thead>
      <tr>
        <th>Course ID</th>
        <th>Course Name</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let c of course">
        <td>{{c.courseId}}</td>
        <td>{{c.courseName}}</td>
      </tr>
    </tbody>
  </table>
```

Child to parent :

If a child component wants to send data to its parent component, then it must create a property with @Output decorator.

The only method for child component to pass data to its parent component is through events. The property must be of type EventEmitter

```
import { Component, OnInit, Input, Output, EventEmitter } from '@angular/core';
@Component({
  selector: 'app-courses-list',
  templateUrl: './courses-list.component.html',
  styleUrls: ['./courses-list.component.css']
})
export class CoursesListComponent {
  @Output() onRegister = new EventEmitter<string>();
  courses = [
    { courseId: 1, courseName: "Node JS" },
    { courseId: 2, courseName: "Typescript" },
    { courseId: 3, courseName: "Angular" },
    { courseId: 4, courseName: "React JS" }
  ];
  register(courseName: string) {
    this.onRegister.emit(courseName);
  }
}
```

```
<table border="1">
    <thead>
      <tr>
        <th>Course ID</th>
        <th>Course Name</th>
        <th></th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let course of courses">
        <td>{{course.courseId}}</td>
        <td>{{course.courseName}}</td>
        <td><button (click)="register(course.courseName)">Register</button></td>
      </tr>
    </tbody>
  </table>
```

```
<h2> Courses List </h2>
<app-courses-list (onRegister)="courseReg($event)"></app-courses-list>
<br/><br/>
<div *ngIf="message">{{message}}</div>
```

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  message: string;
  courseReg(courseName: string) {
    this.message = `Your registration for ${courseName} is successful`;
  }
}
```

Angular provides a mechanism for specifying component specific styles. We will provide styles for a component that won't leak out into the rest of the page.

Component life cycle:

```
import { Component, OnInit,  DoCheck, AfterContentInit, AfterContentChecked,
    AfterViewInit, AfterViewChecked,
    OnDestroy } from '@angular/core';

export class AppComponent implements OnInit,  DoCheck,
    AfterContentInit, AfterContentChecked,
    AfterViewInit, AfterViewChecked,
    OnDestroy {

    data: string = "Angular 2";

    ngOnInit() {
        console.log("Init");
    }

    ngDoCheck() {
        console.log("Change detected");
    }

    ngAfterContentInit() {
        console.log("After content init");
    }

    ngAfterContentChecked() {
        console.log("After content checked");
    }

    ngAfterViewInit() {
        console.log("After view init");
    }

    ngAfterViewChecked() {
        console.log("After view checked");
    }

     ngOnDestroy() {
        console.log("Destroy");
    }
}
```

ngOnChanges – It will be invoked when Angular sets data bound input property i.e., property attached with @Input(). This will be invoked whenever input property changes its value

ngOnInit – It will be invoked when Angular initializes the directive or component

ngDoCheck - It will be invoked for every change detection in the application

ngAfterContentInit – It will be invoked after Angular projects content into its view

ngAfterContentChecked – It will be invoked after Angular checks the bindings of the content it projected into its view

ngAfterViewInit – It will be invoked after Angular creates component's views

ngAfterViewChecked – It will be invoked after Angular checks the bindings of the component's views

ngOnDestroy – It will be invoked before Angular destroys directive or component

Form validations :

```
export class Course {
    constructor(
      public courseId: number,
      public courseName: string,
      public duration: string
    ) { }
  }

import { Component } from '@angular/core';
import { Course } from './course';
@Component({
  selector: 'app-course-form',
  templateUrl: './course-form.component.html',
  styleUrls: ['./course-form.component.css']
})
export class CourseFormComponent {
  course = new Course(1, 'Angular', '5 days');
  submitted = false;
  onSubmit() { this.submitted = true; }
}

<div class="container">
    <div [hidden]="submitted">
      <h1>Course Form</h1>
      <form (ngSubmit)="onSubmit()" #courseForm="ngForm">
        <div class="form-group">
          <label for="id">Course Id</label>
          <input type="text" class="form-control" required
[(ngModel)]="course.courseId" name="id" #id="ngModel">
          <div [hidden]="id.valid || id.pristine" class="alert alert-danger">
            Course Id is required
          </div>
        </div>
        <div class="form-group">
          <label for="name">Course Name</label>
          <input type="text" class="form-control" required
[(ngModel)]="course.courseName" name="name" #name="ngModel">
          <div [hidden]="name.valid || name.pristine" class="alert alert-danger">
            Course Name is required
          </div>
        </div>
        <div class="form-group">
          <label for="duration">Course Duration</label>
          <input type="text" class="form-control" required
[(ngModel)]="course.duration" name="duration" #duration="ngModel">
          <div [hidden]="duration.valid || duration.pristine" class="alert alert-
danger">
            Duration is required
          </div>
        </div>
        <button type="submit" class="btn btn-default"
[disabled]="!courseForm.form.valid">Submit</button>
        <button type="button" class="btn btn-default"
(click)="courseForm.reset()">New Course</button>
      </form>
    </div>
    <div [hidden]="!submitted">
      <h2>You submitted the following:</h2>
      <div class="row">
        <div class="col-xs-3">Course ID</div>
        <div class="col-xs-9  pull-left">{{ course.courseId }}</div>
      </div>
      <div class="row">
        <div class="col-xs-3">Course Name</div>
        <div class="col-xs-9 pull-left">{{ course.courseName }}</div>
      </div>
      <div class="row">
        <div class="col-xs-3">Duration</div>
        <div class="col-xs-9 pull-left">{{ course.duration }}</div>
      </div>
      <br>
      <button class="btn btn-default" (click)="submitted=false">Edit</button>
    </div>
  </div>

  input.ng-valid[required]  {
    border-left: 5px solid #42A948; /* green */
  }
  input.ng-dirty.ng-invalid:not(form)  {
    border-left: 5px solid #a94442; /* red */
  }


  <app-course-form></app-course-form>
```

Template Driven Forms - updateOn Option:

- Angular runs the control validation process whenever a form controls value changes. For example, if you have an input bounded to a form control, Angular runs the control validation for every keystroke.

- A form with heavy validation requirements, updating on every keystroke can sometimes be too expensive.

- Angular provides a new option that improves performance by delaying form control updates until *blur* or *submit* event.

- The possible values for updateOn are:

1) change : Default. The value updates on every change.

2) blur : The value updates once the form lost its focus.

3) submit : The value updates once form is submitted .

These validations can be used on both types of forms at input level or at form level using updateOn property.

```
<div class="container">
    <div [hidden]="submitted">
      <h1>Course Form</h1>
      <form (ngSubmit)="onSubmit()" #courseForm="ngForm">
        <div class="form-group">
          <label for="id">Course Id</label>
          <input type="text" class="form-control" required
[(ngModel)]="course.courseId"
             [ngModelOptions]="{ updateOn: 'blur' }" name="id" #id="ngModel">
          <div [hidden]="id.valid || id.pristine" class="alert alert-danger">
            Course Id is required
          </div>
        </div>
        <div class="form-group">
          <label for="name">Course Name</label>
          <input type="text" class="form-control" required
[(ngModel)]="course.courseName"
             [ngModelOptions]="{ updateOn: 'submit' }" name="name"
#name="ngModel">
          <div [hidden]="name.valid || name.pristine" class="alert alert-danger">
            Course Name is required
          </div>
        </div>
        <div class="form-group">
          <label for="duration">Course Duration</label>
          <input type="text" class="form-control" required
[(ngModel)]="course.duration" name="duration" #duration="ngModel">
          <div [hidden]="duration.valid || duration.pristine" class="alert alert-
danger">
            Duration is required
          </div>
        </div>
        <button type="submit" class="btn btn-default"
[disabled]="!courseForm.form.valid">Submit</button>
```

```
        <button type="button" class="btn btn-default"
(click)="courseForm.reset()">New Course</button>
      </form>
    </div>
    <div [hidden]="!submitted">
      <h2>You submitted the following:</h2>
      <div class="row">
        <div class="col-xs-3">Course ID</div>
        <div class="col-xs-9  pull-left">{{ course.courseId }}</div>
      </div>
      <div class="row">
        <div class="col-xs-3">Course Name</div>
        <div class="col-xs-9 pull-left">{{ course.courseName }}</div>
      </div>
      <div class="row">
        <div class="col-xs-3">Duration</div>
        <div class="col-xs-9 pull-left">{{ course.duration }}</div>
      </div>
      <br>
      <button class="btn btn-default" (click)="submitted=false">Edit</button>
    </div>
    <br/>
    <div>Course Id updates on blur: {{course.courseId}}</div>
    <div>Course Name updates on submit: {{course.courseName}}</div>
  </div>
```

Line 9 & 18 : ngModelOptions token in the input tag is used to update the form validation at input level .

Here course Id value will be updated on blur event as shown below

Here Course Name value will be updated when submit button is clicked i.e., when submit event is triggered

Model Driven Forms or Reactive Forms:

- We can create forms in a reactive style in Angular.
- We need to create form control objects in a component class and should bind them with HTML form elements in the template
- Component can observe the form control state changes and react to them

- We will be using **FormBuilder** class to create reactive forms which has simplified syntax. We need to import **ReactiveFormsModule** to create reactive forms.
- We can use built-in validators using **Validators** class. For example, if we want to use 'required' validator, it can be accessed as **Validators.required**

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { RegistrationFormComponent } from './registration-form/registration-
form.component';
@NgModule({
  declarations: [
    AppComponent,
    RegistrationFormComponent
  ],
  imports: [
    BrowserModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
@Component({
  selector: 'app-registration-form',
  templateUrl: './registration-form.component.html',
  styleUrls: ['./registration-form.component.css']
})
export class RegistrationFormComponent implements OnInit {
  registerForm: FormGroup;
  constructor(private formBuilder: FormBuilder) { }
  ngOnInit() {
    this.registerForm = this.formBuilder.group({
      firstName: ['', Validators.required],
      lastName: ['', Validators.required],
      address: this.formBuilder.group({
        street: [],
        zip: [],
        city: []
      })
    });
  }
}
```

Line 2: Import FormBuilder class to create a reactive form. Also import FormGroup class to create a group of form controls and Validators for validation

Line 11:Create a property registerForm of type FormGroup

Line 13: Inject a formBuilder instance using constructor

Line 16: formBuilder.group() method creates a FormGroup. It takes an object whose keys are FormControl names and values are their definitions

Line 17-21: Create form controls such as firstName, lastName, and address as sub group with fields street, zip and city. These fields are form controls. Configure built-in validators for each control using [' ', Validators.required] syntax where the first parameter is the default value for the control and second parameter is an array of validations. If multiple validators are to be applied, then we should give it as [' ', [Validators.required,Validators.maxlength(10)]]

```
<div class="container">
    <h1>Registration Form</h1>
    <form [formGroup]="registerForm">
      <div class="form-group">
        <label>First Name</label>
        <input type="text" class="form-control" formControlName="firstName">
        <p *ngIf="registerForm.controls.firstName.errors" class="alert alert-
danger">This field is required!</p>
      </div>
      <div class="form-group">
        <label>Last Name</label>
        <input type="text" class="form-control" formControlName="lastName">
        <p *ngIf="registerForm.controls.lastName.errors" class="alert alert-
danger">This field is required!</p>
      </div>
      <div class="form-group">
        <fieldset formGroupName="address">
          <label>Street</label>
          <input type="text" class="form-control" formControlName="street">
          <label>Zip</label>
          <input type="text" class="form-control" formControlName="zip">
          <label>City</label>
          <input type="text" class="form-control" formControlName="city">
        </fieldset>
      </div>
      <button type="submit" (click)="submitted=true">Submit</button>
    </form>
  <br/>
    <div [hidden]="!submitted">
      <h3> Employee Details </h3>
      <p>First Name: {{ registerForm.get('firstName').value }} </p>
      <p> Last Name: {{ registerForm.get('lastName').value }} </p>
      <p> Street: {{ registerForm.get('address.street').value }}</p>
      <p> Zip: {{ registerForm.get('address.zip').value }} </p>
      <p> City: {{ registerForm.get('address.city').value }}</p>
    </div>
  </div>
```

Line 3: formGroup is a directive which binds HTML form with the FormGroup property created inside a component class. We have created a FormGroup in component with name registerForm. Here form tag is binded with FormGroup name called registerForm

Line 6 , 11: Two textboxes for first name and last name are binded with the form controls created in the component using formControlName directive

Line 24: When submit button is clicked, it initializes submitted property value to true

Line 27 : div tag will be hidden if form is not submitted

Line 29-33: Using get() method of FormGroup, we are fetching values of each FormControl and rendering it

```
.ng-valid[required]  {
    border-left: 5px solid #42A948; /* green */
  }
  .ng-invalid:not(form)  {
    border-left: 5px solid #a94442; /* red */
  }
```

Line 1-3: ng-valid css class changes left border of textbox to green if form control has valid input

Line 5-7: ng-invalid cass class changes left border of textbox to red if form control has invalid data

```
<app-registration-form></app-registration-form>
```

Reactive Forms - updateOn Option:

Similar to template driven forms, updateOn option can also be applied on Reactive forms

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators, FormControl } from '@angular/forms';
@Component({
  selector: 'app-registration-form',
  templateUrl: './registration-form.component.html',
  styleUrls: ['./registration-form.component.css']
})
export class RegistrationFormComponent implements OnInit {
  registerForm: FormGroup;
  constructor(private formBuilder: FormBuilder) { }
  ngOnInit() {
    this.registerForm = this.formBuilder.group({
      firstName: ['', { updateOn: 'blur', validators: [Validators.required] }],
      lastName: ['', Validators.required]
    });
  }
}
```

Custom Validation:

We can create custom validators when there are no built-in validators are available to implement the required functionality.

We can create custom validators for both template driven and reactive forms. Let us see how to create custom validator for template driven forms.

Check in lex;

Services:

A service in Angular is a class which contains some functionality that can be reused across the application. A service is a singleton object. Angular services are a mechanism of abstracting shared code and functionality throughout the application.

Angular Services come as objects which are wired together using dependency injection.

Angular provides few inbuilt services. We can also create custom services.

Why Services?

Services can be used to share the code across components of an application.

Services can be used to make http requests.

Creating a Service?

Create a class with @Injectable() decorator.

@Injectable() decorator makes the class injectable into application components.

```
@Injectable()
class MyService
{
}
```

njecting a Service

There are two steps to inject a service.

1. Add providers property in the module class so that service class is available to the entire application.

providers: [MyService]

If service is needed only for specific component(s), it can be added in those component classes instead of module.

2. Add a constructor in a component class with Service class as argument.

constructor(private service: MyService){ }

myService will be injected into the component through constructor injection by the framework

```
export class Book {
    id: number;
    name: string;
}

import { Book } from './book';
export var BOOKS: Book[] = [
    { "id": 1, "name": "HTML 5" },
    { "id": 2, "name": "CSS 3" },
    { "id": 3, "name": "Java Script" },
    { "id": 4, "name": "Ajax Programming" },
    { "id": 5, "name": "jQuery" },
    { "id": 6, "name": "Mastering Node.js" },
    { "id": 7, "name": "Angular JS 1.x" },
    { "id": 8, "name": "ng-book 2" },
    { "id": 9, "name": "Backbone JS" },
    { "id": 10, "name": "Yeoman" }
];
```

```
import { Injectable } from '@angular/core';
import { Book } from './book';
import { BOOKS } from './books-data';
@Injectable()
export class BookService {
  getBooks() {
    return Promise.resolve(BOOKS);
  }
}
```

```
import { Component, OnInit } from '@angular/core';
import { Book } from './book';
import { BookService } from './book.service';
@Component({
  selector: 'app-book',
  templateUrl: './book.component.html',
  styleUrls: ['./book.component.css']
})
export class BookComponent implements OnInit {
  books: Book[];
  constructor(private bookService: BookService) { }
  getBooks() {
    this.bookService.getBooks().then(books => this.books = books);
  }
  ngOnInit() {
    this.getBooks();
  }
}
```

```
<app-book></app-book>
```

RxJS Observables:

RxJS is a reactive streams library used to work with asynchronous streams of data.

Observables, in RxJS, are used to represent asynchronous streams of data. Observables are more advanced version of Promises in JavaScript

Why RxJS Observables?

Angular team has recommended Observables for asynchronous calls because of the following reasons:

1. Promises emit a single value whereas observables (streams) emit many values
2. Observables can be cancellable where Promises are not cancellable. If any http response is not required, observables allows us to cancel the subscription whereas promises execute either success or failure callback even if we don't need the result
3. Observables support functional operators such as map, filter, reduce etc.,

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';
@Component({
  selector: 'app-root',
  styleUrls: ['./app.component.css'],
  templateUrl: './app.component.html'
})
export class AppComponent {
  data: Observable<number>;
  myArray: number[] = [];
  errors: boolean;
  finished: boolean;
  fetchData() {
    this.data = new Observable(observer => {
      setTimeout(() => { observer.next(11); }, 1000),
        setTimeout(() => { observer.next(12); }, 2000),
        setTimeout(() => { observer.complete(); }, 3000)
    });
    let sub = this.data.subscribe((value) => this.myArray.push(value),
      error => this.errors = true,
      () => this.finished = true);
  }
}
```

Line 2 : imports Observable class from rxjs module

Line 11: data is of type Observable which holds numeric values

Line 16: fetchData() is invoked on click of a button

Line 17: A new Observable is created and stored in the variable data

Line 18-20: next() method of Observable sends the given data through the stream. With a delay of 1,2 and 3 seconds, a stream of numeric values will be sent. Complete() method completes the Observable stream i.e., closes the stream.

Line 23: Observable has another method called subscribe which listens to the data coming through the stream. Subscribe() method has three parameters. First parameter is a success callback which will be invoked upon receiving successful data from the stream. Second parameter is a error callback which will be invoked when Observable returns an error and third parameter is a complete callback which will be invoked upon successful streaming of values from Observable i.e., once complete() is invoked. Here upon successful response, the data is pushed to the local array called myArray, if any error occurs, a Boolean value called true is stored in errors variable and upon complete() will assign a Boolean value true in finished variable.

```
<b> Using Observables!</b>
<h6 style="margin-bottom: 0">VALUES:</h6>
<div *ngFor="let value of myArray"> {{ value }}</div>
<div style="margin-bottom: 0">ERRORS: {{ errors }}</div>
<div style="margin-bottom: 0">FINISHED: {{ finished }}</div>
<button style="margin-top: 2rem;" (click)="fetchData()">Fetch Data</button>
```

Line 4: ngFor loop is iterated on myArray which will display the values on the page

Line 6: {{ errors }} will render the value of errors property if any

Line 8: Displays finished property value when complete() method of Observable is executed

Line 10: Button click event is binded with fetchData() method which is invoked and creates an observable with a stream of numeric values

Server Communication using HttpClient:

Most front-end applications communicate with backend services using HTTP Protocol

When we make calls to an external server, we want our user to continue to be able to interact with the page. That is, we don't want our page to freeze until the HTTP request returns from the external server. So, all HTTP requests are asynchronous.

We need to use HttpClient from @angular/common/http communicate with backend services.

Additional benefits of HttpClient include testability features, typed request and response objects, request and response interception, Observable apis, and streamlined error handling.

We need to import HttpClientModule from @angular/common/http in the module class to make http service available to the entire module. Import HttpClient service class into a component's constructor. We can make use of http methods like get, post, put and delete.

JSON is the default response type for HttpClient

The following statement is used to fetch data from a server

```
this.http.get(url)
```

http.get by default returns an observable

Let us use the same example used for custom services.

Add HttpClientModule to the app.module.ts to make use of HttpClient class.

```
...
import { HttpClientModule } from '@angular/common/http';
...
@NgModule({
  imports: [BrowserModule, HttpClientModule],
  ...
})
export class AppModule { }
```

Line 2: imports HttpClientModule from @angular/common/http module

Line 6: Includes HttpClientModule class to make use of Http calls

Create books.json under assets folder and move books data to it

```
[
    {   "id": 1, "name": "HTML 5"   },
    {   "id": 2, "name": "CSS 3"    },
    {   "id": 3, "name": "Java Script"  },
    {   "id": 4, "name": "Ajax Programming" },
    {   "id": 5, "name": "jQuery"   },
    {   "id": 6, "name": "Mastering Node.js"  },
    {   "id": 7, "name": "Angular JS 1.x"  },
    {   "id": 8, "name": "ng-book 2" },
    {   "id": 9, "name": "Backbone JS" },
    {   "id": 10, "name": "Yeoman" }
]
```

Add getBooks() method to BookService class in book.service.ts file as shown below

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { catchError, tap } from 'rxjs/operators';
import { Book } from './book';
@Injectable()
export class BookService {
  private booksUrl = './assets/books.json';
  constructor(private http: HttpClient) { }
  getBooks(): Observable<Book[]> {
   return this.http.get<Book[]>(this.booksUrl).pipe(
      tap(data => console.log('Data fetched:'+JSON.stringify(data))),
      catchError(this.handleError));
  }
}
```

Line 2: Imports HttpClient class from @angular/common/http module.

Line 3: Imports Observable class from rxjs module

Line 4: Imports rxjs operators

Line 11: Stores the json file path in a variable called booksUrl

Line 13: Injects HttpClient class into service class

Line 16-18: Makes an asynchronous call (ajax call) by using get() method of HttpClient class. This method makes an asynchronous call to the json file and fetches the data. HttpClient receives the JSON response as of type object. To know the actual structure of the response, we need to create an interface and specify that interface name as type parameter i.e., get<Book[]>. Pipe function lets you define a comma separated sequence of operators. Here we define a sequence of observables by listing operators as arguments to pipe function instead of dot operator chaining. Tap operator is to execute some statements once response is ready which is mostly used for debugging purposes and catchError operator is used to handle the errors .

Line 18: handleError is error handling method which throws the error message back to the component

Error handling

What happens if the request fails on the server, or if a poor network connection prevents it from even reaching the server?

There are two types of errors that can occur. The server might reject the request, returning an HTTP response with a status code such as 404 or 500. These are error responses.

Or something could go wrong on the client-side such as network error that prevents the request from completing successfully or an exception thrown in an RxJS operator. These errors produce JavaScript ErrorEvent objects.

HttpClient captures both kinds of errors in its HttpErrorResponse and we can inspect that response to find out what really happened.

We need to do error inspection, interpretation and resolution in service not in the component.

Add the following error handling code in book.service.ts file

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { catchError, tap } from 'rxjs/operators';
import { Observable } from 'rxjs';
import { HttpErrorResponse } from '@angular/common/http';
import { Book } from './book';
@Injectable()
export class BookService {
  private booksUrl = './assets/books.json';
  ...
  private handleError(err:HttpErrorResponse) {
    let errMsg:string='';
    if (err.error instanceof Error) {
       // A client-side or network error occurred. Handle it accordingly.
       console.log('An error occurred:', err.error.message);
        errMsg=err.error.message;}
       else {
       // The backend returned an unsuccessful response code.
       // The response body may contain clues as to what went wrong,
       console.log(`Backend returned code ${err.status}`);
          errMsg=err.error.status;
     }
        return Observable.throw(errMsg);
  }
}
```

Line 5 : HttpErrorResponse module class should be imported to understand the nature of error thrown

Line 18-21 : An instance of Error object will be thrown if any network or client side error is thrown

Line 25-26 : Handling of errors due to unsuccessful response codes from backend

Modify the code in book.component.ts file as shown below

```
export class BookComponent implements OnInit {
  books: Book[];
  errorMessage: string;
  constructor(private bookService: BookService) { }
  getBooks() {
    this.bookService.getBooks().subscribe(
      books => this.books = books,
      error => this.errorMessage = <any>error);
  }
  ngOnInit() {
    this.getBooks();
  }
}
```

Line 7: Inject the BookService class into the component class through constructor

Line 9-13: Invokes the service class method getBooks() which makes an http call to books.json file and the response is returned. Once response is returned, observable has a subscribe method, similar to then method in promises, which has a success callback and failure callback

```
book.component.html
```

```
<ul class="books">
  <li *ngFor="let book of books">
    <span class="badge">{{book.id}}</span> {{book.name}}
  </li>
</ul>
<div class="error" *ngIf="errorMessage">{{errorMessage}}</div>
```

```
Line 2-6 : Displays books details

Line 8: Displays error message when http get operation fails
```

Retrying Http Requests:

- Few errors can be momentary and are most unlikely to repeat . Such errors could be cleared up on making the same call few seconds later.
- Most of these errors might occur when dealing with an external source like a database or web service which can have network or other temporary issues .
- Such requests can be mitigated by using retry function.

Modify **book.service.ts** file used in HttpClient demo as shown below

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { catchError, tap, retry } from 'rxjs/operators';
import { Observable } from 'rxjs';
import { HttpErrorResponse } from '@angular/common/http';
import { Book } from './book';
@Injectable()
export class BookService {
  private booksUrl = './assets/nothing.json';
  constructor(private http: HttpClient) { }
  getBooks(): Observable<Book[]> {
    return this.http.get<Book[]>(this.booksUrl).pipe(
      retry(3),
      tap(data => console.log('fetched Data from json')),
      catchError(this.handleError));
  }
  private handleError(err:HttpErrorResponse) {
    let errMsg:string='';
    if (err.error instanceof Error) {
       // A client-side or network error occurred. Handle it accordingly.
       console.log('An error occurred:', err.error.message);
        errMsg=err.error.message;}
       else {
       // The backend returned an unsuccessful response code.
       // The response body may contain clues as to what went wrong,
       console.log(`Backend returned code ${err.status}`);
          errMsg=err.error.status;
     }
        return Observable.throw(errMsg);
  }
}
```

Line 3: Imports rxjs retry operator

Line 18 : If initial request to the server is not successful, retry() will try to connect to the url three times.

Output

Retries connecting to the URL thrice after its first failure

Routes:

After configuring the routes, the next step is to decide how to navigate. Navigation will happen based on some user action like clicking a hyperlink etc.,

We can now use RouterLink directive to the anchor tag for navigation as shown below

`app.component.ts`

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

```
app.component.html

<h1>{{title}}</h1>
<nav>
    <a [routerLink]='["/dashboard"]' routerLinkActive="active">Dashboard</a>
    <a [routerLink]='["/books"]' routerLinkActive="active">Books</a>
</nav>
<router-outlet></router-outlet>
```

Line 3-4: Create hyperlinks and a routerLink directive and specify the paths to navigate. Here, if user clicks on Dashboard, it will navigate to /dashboard. routerLinkActive applies the given css class to the link when it is clicked to make it look as active link(active is a css class defined in app.component.css which changes the link color to blue in this case)

Line 6 : <router-outlet> is the place where the output of the component associated with the given path will be displayed. For example, if user click on Books, it will navigate to /books which will execute BooksComponent class as mentioned in the configuration details and the output will be displayed in the router-outlet class

To navigate programmatically, we can use navigate() method of Router class. Inject the router class into the component and invoke navigate method as shown below

```
this.router.navigate([url, parameters])
```

url is the route path to which we want to navigate

Parameters are the route values passed along with the url

Route Parameters:

Parameters passed along with URL are called route parameters.

Generate dashboard component using the following CLI command

```
D:\MyApp>ng generate component dashboard
```

Add the following code in dashboard.component.ts file

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { Book } from '../book/book';
import { BookService } from '../book/book.service';
@Component({
  selector: 'app-dashboard',
  templateUrl: './dashboard.component.html',
  styleUrls: ['./dashboard.component.css']
})
export class DashboardComponent implements OnInit {
  books: Book[] = [];
  constructor(
    private router: Router,
    private bookService: BookService) { }
  ngOnInit() {
    this.bookService.getBooks()
      .subscribe(books => this.books = books.slice(1, 5));
  }
  gotoDetail(book: Book) {
    this.router.navigate(['/detail', book.id]);
  }
}
```

Line 2: Import Router class from @angular/router module

Line 16: Inject into the component class through a constructor

Line 24: this.router.navigate() method is used to navigate to a specific URL programmatically. Navigate() method takes two arguments- first one is the path to navigate and the second one is the route parameter value to pass. Here the path will be detail/<book_id>

Accessing Route Parameters

To access route parameters, use ActivatedRoute class

Generate dashboard component using the following CLI command

```
D:\MyApp>ng generate component book-detail
```

Add the following code in book-detail.component.ts file

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap } from '@angular/router';
import { switchMap } from 'rxjs/operators';
import { Book } from '../book/book';
import { BookService } from '../book/book.service';
@Component({
  selector: 'app-book-detail',
  templateUrl: './book-detail.component.html',
  styleUrls: ['./book-detail.component.css']
})
export class BookDetailComponent implements OnInit {
  book: Book;
  error: any;
  sub: any;
  constructor(private bookService: BookService, private route: ActivatedRoute) {
}
  ngOnInit() {
    this.sub = this.route.paramMap.switchMap((params: ParamMap) =>
      this.bookService.getBook(+params.get('id')))
      .subscribe((book: Book) => this.book = book);
  }
  ngOnDestroy() {
    this.sub.unsubscribe();
  }
  goBack() {
    window.history.back();
  }
}
```

Line 2: Imports ActivatedRoute class to access route parameters

Line 19: Injects ActivatedRoute class into the component class through a constructor

Line 22-24: ActivatedRoute class has a paramMap observable method which holds the route parameters. It has switchMap() method to process the route parameters. ParamMap has get() method to fetch a specific parameter value

Route Guards:

In Angular application, user can navigate to any url directly. That's not the right thing to do always.

Consider the following scenarios

- User must login first to access a component
- User is not authorized to access a component
- User should fetch data before displaying a component
- Pending changes should be saved before leaving a component

We should go for **route guards** to handle these scenarios

A guard's return value controls the behavior of router

- If it returns true, the navigation process continues
- If it returns false, the navigation process stops

Angular has canActivate interface which can be used to check if a user is logged in to access a component

We need to override canActivate() method in the guard class as shown below

Using canActivate, we can permit access to only authenticated users

```
Class GuardService implements CanActivate{
    canActivate( ): boolean {
    }
}
```

Asynchronous Routing:

When an Angular application has lot of components, it will increase the size of the application. In such scenario, at some point, application takes lot of time to load

To overcome this problem, we can go for asynchronous routing i.e, we need to load modules lazily only when they are required instead of loading them at the beginning of the execution

Lazy Loading has following benefits:

Modules are loaded only when user requests for it
We can speed up load time for users who will be visiting only certain areas of the application

Lazy Loading Route Configuration:

To apply lazy loading on modules, create a separate routing configuration file for that module and map empty path to the component of that module.

In the example in the previous concept, consider BookComponent. If we want to load it lazily, create book-routing.module.ts file and map empty path to BookComponent(Line 9-10)

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { BookComponent }  from './book.component';
import { LoginGuardService } from '../login/login-guard.service';
const bookRoutes: Routes = [
    {
      path: '',
      component: BookComponent,
      canActivate: [ LoginGuardService ]
    }
];
@NgModule({
  imports: [ RouterModule.forChild(bookRoutes) ],
  exports: [ RouterModule ]
})
export class BookRoutingModule{ }
```

In the root routing configuration file app-routing.module, bind 'book' path to the BookModule using loadChildren property as shown below

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { DashboardComponent } from './dashboard/dashboard.component';
import { BookDetailComponent } from './book-detail/book-detail.component';
import {LoginComponent } from './login/login.component';
const appRoutes: Routes = [
    { path: 'dashboard', component: DashboardComponent },
    { path: '', redirectTo: '/login', pathMatch: 'full' },
    { path: 'books', loadChildren: './book/book.module#BookModule'},
    { path: 'detail/:id', component: BookDetailComponent },
    {path:'login', component: LoginComponent}
];
@NgModule({
    imports: [
        RouterModule.forRoot(appRoutes)
    ],
    exports: [
        RouterModule
    ]
})
export class AppRoutingModule { }
```

Line 11: Binds books path to BookModule using loadChildren property

Note: Remove BookComponent class from app.module.ts file

Finally, it loads the requested route to the destination book component.

Angular In-Memory Web API:

Angular in-memory web API can be used to simulate a web server. Let us create a mock service using it and implement post, put and delete operations using Http class.

D:\MyApp>npm install angular-in-memory-web-api –save

Angular Testing using Jasmine:

Testing routing means, testing the component to make sure that the navigation happens with the right address under the given conditions.

Angular uses "navigate" method which attempts to navigate to the specified URL string using the router.(Line 20-22)

We will stub the regular Router class with RouterStub that can use the method "navigate". (Line 27-36)

We will spy on the method navigate that belongs to RouterStub.(Line 40)

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { FormsModule, ReactiveFormsModule }        from '@angular/forms';
import { LoginComponent } from './login.component';
import { LoginService } from './login.service';
import { Router, NavigationExtras }  from '@angular/router';
import { Login } from './Login';
describe('LoginComponent', () => {
  let component:    LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;
  let loginService:LoginService;
  let router: Router;
  class LoginServiceStub {
    checkUser(x:Login) :boolean {
      return true;
    };
  };

  class RouterStub {
    navigate(commands: any[], extras?: NavigationExtras) { };
  };
  let loginServiceStub:LoginServiceStub = new LoginServiceStub();
  let routerStub:RouterStub = new RouterStub();
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ LoginComponent ],
      imports:[ ReactiveFormsModule, FormsModule ],
      providers: [{provide: LoginService, useValue:loginServiceStub},
                {provide: Router, useValue:routerStub}
                ],
    })
    .compileComponents();
  }));
  beforeEach(() => {
    loginService = TestBed.get(LoginService);
  router = TestBed.get(Router);
    fixture = TestBed.createComponent(LoginComponent);
    component = fixture.componentInstance;
    component.ngOnInit();
  fixture.detectChanges();
  });
    // Checking everything is created correct or not
  it('should be created', () => {
    expect(component).toBeTruthy();
  });

  // Checking form is invalid if it is empty
  it('should have invalid form when it is empty', () => {
    expect(component.loginForm.valid).toBeFalsy();
  });

  describe('should have username field', () => {
    let errors = {};
    let userName;

    beforeEach( ()=>{
      userName= component.loginForm.controls['userName'];
      errors = userName.errors || {};
    });

    // Checking userName is invalid if it is empty
    it('which is invalid when it is empty', () => {
      expect(userName.valid).toBeFalsy();
    });
    // Checking required error is present if username is invalid
    it('which contains required error when it is empty', () => {
      expect(errors['required']).toBeTruthy();
    });
  });
  describe('should have username field', () => {
    let errors = {};
```

```
    let userName;

    beforeEach( ()=>{
      userName= component.loginForm.controls['userName'];
      userName.setValue("Kalpana");
      errors = userName.errors || {};
    });

    // Checking userName is valid if it is filled
    it('which is valid when it is filled', () => {
      expect(userName.valid).toBeTruthy();
    });
    // Checking required error is not present if username is valid
    it('which should not contains required error when it is filled', () => {
      expect(errors['required']).toBeFalsy();
    });
  });
  describe('should have password field', () => {
    let errors = {};
    let password;
    beforeEach( ()=>{
      password= component.loginForm.controls['password'];
      errors = password.errors || {};
    });
    // Checking password is invalid if it is empty
    it('which is invalid when it is empty', () => {
      expect(password.valid).toBeFalsy();
    });
    // Checking required error is present if password is invalid
    it('which contains required error when it is empty', () => {
      expect(errors['required']).toBeTruthy();
    });
  });
  describe('should have password field', () => {
    let errors = {};
    let password;
    beforeEach( ()=>{
      password= component.loginForm.controls['password'];
      password.setValue("Kalpana");
      errors = password.errors || {};

    });
    // Checking password is valid if it is filled
    it('which is valid when it is filled', () => {
      expect(password.valid).toBeTruthy();
    });
    // Checking required error is not present if password is valid
    it('which should not contains required error when it is filled', () => {
      expect(errors['required']).toBeFalsy();
    });
  });

  // Checking form is valid if all fields are filled properly
  it('should have valid from when all fields are correct', () => {
      component.loginForm.controls['userName'].setValue("Kalpana");
      component.loginForm.controls['password'].setValue("Kalpana");
    expect(component.loginForm.valid).toBeTruthy();
  });


  describe('should have noOfAttempts', () => {
    let spy;
    beforeEach( ()=>{
      spy = spyOn(loginService, 'checkUser')
        .and.returnValue(false);
      component.loginForm.controls['userName'].setValue("Kalpana");
      component.loginForm.controls['password'].setValue("Kalpana");
      component.userLogin();

    });
    // Checking noOfAttempts for first time with wrong values
    it('which should be increased by 1 on giving wrong credentials', () => {
      expect(component.noOfAttempts).toBe(1);
    });
    // Checking noOfAttempts for second time with wrong values
    it('which should be increased by 2 on giving wrong credentials for second
time', () => {
      component.userLogin();
      expect(component.noOfAttempts).toBe(2);
    });
    // Checking noOfAttempts for third time with wrong values
    it('which should be increased by 2 on giving wrong credentials for second
time', () => {
      component.userLogin();
      component.userLogin();
      expect(component.noOfAttempts).toBe(3);
    });
  });
  describe('should have error', () => {
    let spy;
    beforeEach( ()=>{
      spy = spyOn(loginService, 'checkUser')
        .and.returnValue(false);
      component.loginForm.controls['userName'].setValue("Kalpana");
      component.loginForm.controls['password'].setValue("Kalpana");
      component.userLogin();

    });
    // Checking error for first time with wrong values
    it('which should show proper error message for first time invalid entry', ()
=> {
      expect(component.error).toBe('Username or Password mismatch. Only two more
attempts you have....');
    });
    // Checking error for second time with wrong values
    it('which should show proper error message for second time invalid entry', ()
=> {
      component.userLogin();
      expect(component.error).toBe('Again Username or Password mismatch. Last
attempt. Please enter correctly');
    });
    // Checking error for third time with wrong values
    it('which should show proper error message for third time invalid entry', ()
=> {
      component.userLogin();
      component.userLogin();
      expect(component.error).toBe('Sorry you have exceeded all the
attempts...Come again later...');
    });
  });
  describe('should', () => {
    let spy;
    beforeEach( ()=>{
      spyOn(loginService, 'checkUser').and.returnValue(true);
      spy = spyOn(router,'navigate');
      component.loginForm.controls['userName'].setValue("Kalpana");
      component.loginForm.controls['password'].setValue("Kalpana");
      fixture.detectChanges();
      component.userLogin();
    });

    // Checking password is valid if it is filled
    it('call the router on giving the correct credentials', () => {
      expect(spy.calls.any()).toBe(true);
    });
    // Checking required error is not present if password is valid
    it('have noOfAttempts which is 0 on giving correct credentials', () => {
      expect(component.noOfAttempts).toBe(0);
    });
  });
});
```

Angular 7:

ng update @angular/cli @angular/core

CLI Prompts:

Angular CLI will now prompt users to select few built-in features like routing, stylesheet selection while running ng new command as shown below. This helps developers to enable these features while application creation itself.

Application Performance:

Developers used to do a common mistake of including reflect-metadata polyfill in production which is needed only in development mode. This has been corrected in the latest version of Angular where the polyfill will be automatically removed from production build.

Also, Angular 7 will take advantage of CLI feature called Bundle Budget which will notify the developers when application reaches size limits. New applications will warn if bundle size is more than 2MB and will error at 5MB. Developers can change these size limits in angular.json file as shown below.

Dependency Updates:

Angular 7 now supports the following dependencies support also.

- Typescript 3.1
- RxJS 6.3
- Node 10

Angular Elements:

Angular Elements now supports content projection also.

<my-custom-element>This content can now be projected</my-custom-element>