**Angular**

A javascript framework for building client-side applications.

Angular 2 application is comprised of a **set of components** and **services** that provide functionality across those components

A component is the combination of an HTML template and a component class that controls a portion of the screen.

Language used in Angular application is **TypeScript**

**Setting up Environment**

* Install npm (node package manager) is a command line utility
  + <https://nodejs.org/en/download/current/>
* Setting up angular 2 application
  + Download **git** if not installed
  + Use git clone <https://github.com/DeborahK/Angular2-GettingStarted>
  + In the terminal type **npm install**
    - The above command will install the required modules needed for your angular application based on the dependencies specified in package.json
  + In the terminal type **npm start**
  + It will start a server to launch our application (lite-server) and it will run the **typescript compiler tsc - w (watch mode)**
  + Both typescript compiler and the file server watch for any file changes, so if we make any changes to our file the typescript recompiles them and the changes happens in the browser automatically
* Understanding the basic application
  + Index.html file contains
  + SystemJs :
    - SystemJS is a module loader that can import modules (ES6 modules).
    - loads all the application code files, so we don't need to add <script> for all the files that we use in our application.
  + **System.import('app').catch(function(err){ console.error(err); });**
    - Starts our application by loading all the required modules which is in the **app folder,**
  + **systemjs.config** : file which loads the main entry point to the application which is specified in packages, so it loads and run the **main.js** file.
  + **Main.ts**
  + **platformBrowserDynamic :** to compile the application dynamically
  + **appmodule** : application module
    - Defined using **export class AppModule { }**
    - Use **@NgModule** as decorator and pass meta data defining details of module.

**Creating components**

As said angular application is comprised of a set of components, we create each components and arrange them to form our application.

A component contains the following parts

* **A template** (view layout ) created using html, which defines what is rendered on the page
* **Class** for the code associated with the view, which will be created using **typescript**, it also contains all the data elements available for use in the view, methods were all the logic is written for the view.
* Metadata which provides additional information about the component to angular. It is defined using a **decorator**.
* **Component => Template + Class + Metadata**
* Simple component file would look like
* import {Component} from '@angular/core';  
    
  @Component({  
   selector:'ml-app',  
   template:`  
   <div>  
   <h1>{{pageTitle}}</h1>  
   <p>My first angular2 component</p>  
   </div>  
   `  
  })  
  export class AppComponent {  
   pageTitle : string = 'Marlabs - Angular Traning !!!';  
  }

**Creating component class**

We define the class using the **class** keyword followed by component name. The component name here is App and Component suffix will be appended to it, so it becomes **AppComponent.**

export class AppComponent {

pageTitle : string = “Marlabs”;

}

* By convention the root component of an application is called AppComponent.
* **pageTitle** : is the property with a value Marlabs
* With typescript we specify the datatype for each property that we create, here it is **string**. Methods are also added in the class body.

**Defining metadata**

Angular need metadata data to understand how to instantiate the component, construct the view and interact with the component.

Defined using angular **Component()** and in typeScript we attach that function to the class as a **decorator** and is always prefixed with a **@ symbol, @Component()**

We apply a decorator by placing immediately in front of the feature we are decorating.

Since we are decorating the class we place decorator in immediately in front of the class.

@Component({  
 selector: 'pm-app', // directive name  
 template: `  
 <h1>{{pageTitle}}</h1>

<div>Angular2: Getting Started</div>  
 `  
})  
export class AppComponent {

pageTitle : string = “Marlabs”;

}

**Importing Components**

In the component code we used the **@Component()** from angularto define our class as a component. Now we need to tell the module loader where to find this function.

This is done using the import statments.

**import {Component} from ‘@angular/core’;**

**Note :** Let's create our first component

import {Component} from '@angular/core';  
  
@Component({  
 selector: 'mat-app',  
 template : `  
 <div>  
 <h1>{{pageTitle}}</h1>  
 <div>My first component</div><br />  
 </div>  
 `  
})  
  
export class AppComponent {  
 pageTitle : string = `Marlabs - Angular training`;  
}

**Bootstrapping our App Component**

* Load the root component (bootstrapping)
* Setup index.html file to host our application

index.html contains the main page for the application, and could be the only webpage in the application and all the other html code will be displayed into this web page and hence called **single page application**

We use the directive we created **mat-app** in the index.html and give a **loading…** message.

<mat-app>Loading app….</mat-app>

When the **root element** is loaded the Loading app… message appears and as soon as the loading is complete the html code defined inside the template is inserted in between the selector element tag

**Angularjs Application Startup**

index.html => System.import(‘app) => Systemjs.config.js => main.ts => app.module.ts => app.component.ts

1. index.html file loads and executes the module main.ts
2. main.ts bootstraps the angular module (AppModule)
3. AppModule bootstraps the root application component (AppComponent)
4. Component template appears in the browser

**Creating ProductListComponent**

1. Create a folder inside app folder **products**
2. Create component file **product-list.component.ts**
3. import {Component} from '@angular/core';  
     
   @Component({  
    selector:'ml-products',  
    templateUrl:'app/products/product-list-component.html'  
   })  
   export class ProductListComponent {  
      
   }
4. Use **ml-products** inside AppComponent
   1. <ml-products></ml-products>
5. Need to include the component inside AppModule
   1. Import the component
      1. import { ProductListComponent } from './products/product-list.component';
   2. Include the component inside the declarations array
      1. declarations: [   
          AppComponent,  
          ProductListComponent  
          ]
6. Build the template for the component
   1. Use product-list.component.html file to add all the html content
   2. <div class="panel panel-primary">  
       <div class="panel-heading">  
       Product List  
       </div>  
       <div class="panel-body">  
       <div class="row">  
       <div class="col-md-2">Filter by</div>  
       <div class="col-md-4">  
       <input type="text" />  
       </div>  
       </div>  
       <div class="row">  
       <div class="col-md-6">  
       <h3>Filtered by : </h3>  
       </div>  
       </div>  
       <div class="table-responsive">  
       <table class="table">  
       <thead>  
       <tr>  
       <th>  
       <button class="btn btn-primary">Show image</button>  
       </th>  
       <th>Products</th>  
       <th>Code</th>  
       <th>Available</th>  
       <th>Price</th>  
       <th>5 Star rating</th>  
       </tr>  
       </thead>  
       <tbody>  
         
       </tbody>  
       </table>  
       </div>  
       </div>  
      </div>

**Data binding**

1. Coordinates the communication between the component class and the template and also involves passing data
2. Angular use **interpolation** for passing data.
3. Interpolation is a special syntax that Angular converts into a property binding **{{}},** it is a one way binding from the class property to the template.
4. {{pageTitle}} : evaluates the pageTitle variable and displays the the result.
5. So let's create a new variable inside productlist component.

**Adding logic using angular directives**

1. Let's create a new variable inside the component class to display products
   1. products : any[] = [  
       {  
       "productId": 1,  
       "productName": "Leaf Rake",  
       "productCode": "GDN-0011",  
       "releaseDate": "March 19, 2016",  
       "description": "Leaf rake with 48-inch wooden handle.",  
       "price": 19.95,  
       "starRating": 3.2,  
       "imageUrl": "http://openclipart.org/image/300px/svg\_to\_png/26215/Anonymous\_Leaf\_Rake.png"  
       },  
       {  
       "productId": 2,  
       "productName": "Garden Cart",  
       "productCode": "GDN-0023",  
       "releaseDate": "March 18, 2016",  
       "description": "15 gallon capacity rolling garden cart",  
       "price": 32.99,  
       "starRating": 4.2,  
       "imageUrl": "http://openclipart.org/image/300px/svg\_to\_png/58471/garden\_cart.png"  
       },  
       {  
       "productId": 5,  
       "productName": "Hammer",  
       "productCode": "TBX-0048",  
       "releaseDate": "May 21, 2016",  
       "description": "Curved claw steel hammer",  
       "price": 8.9,  
       "starRating": 4.8,  
       "imageUrl": "http://openclipart.org/image/300px/svg\_to\_png/73/rejon\_Hammer.png"  
       },  
       {  
       "productId": 8,  
       "productName": "Saw",  
       "productCode": "TBX-0022",  
       "releaseDate": "May 15, 2016",  
       "description": "15-inch steel blade hand saw",  
       "price": 11.55,  
       "starRating": 3.7,  
       "imageUrl": "http://openclipart.org/image/300px/svg\_to\_png/27070/egore911\_saw.png"  
       },  
       {  
       "productId": 10,  
       "productName": "Video Game Controller",  
       "productCode": "GMG-0042",  
       "releaseDate": "October 15, 2015",  
       "description": "Standard two-button video game controller",  
       "price": 35.95,  
       "starRating": 4.6,  
       "imageUrl": "http://openclipart.org/image/300px/svg\_to\_png/120337/xbox-controller\_01.png"  
       }  
       ];
   2. While defining products array, we used the data type as **any[]**
   3. **any[]** is the data type used when we aren't sure about the datatype in typescript.
2. Let's use ngIf directive to check whether the products array is declared and contains some elements.
   1. <table class="table" \*ngIf="products && products.length"></table>
3. Using angular directive \*ngFor, we can iterate through the array elements.
   1. <tr \*ngFor='let product of products'>  
       <td></td>  
       <td>{{product.productName}}</td>  
       <td>{{product.productCode}}</td>  
       <td>{{product.releaseDate}}</td>  
       <td>{{product.price}}</td>  
       <td>{{product.starRating}}</td>  
       </tr>

**Property binding**

1. Allows to set property of an element to value of a template expression
2. Binding targets as always enclosed in square brackets []
3. Binding source are always enclosed in quotes
4. [binding target] = ‘binding source’;
5. Example for a property binding
   1. <img [src]='product.imageUrl' [title]='product.productName' [style.width.px] = 'imageWidth' />
6. In the above example we are binding 3 properties
   1. [src]='product.imageUrl'
   2. [title]='product.productName'
   3. [style.width.px] = 'imageWidth'
      1. For this we need to create a new property inside the component class

**Event binding**

1. Allows to bind an event to an element
2. Target event is always enclosed in parentheses
3. Component class method will always be enclosed in quotes
4. (target event) = ‘method()’;
5. Example for event binding
   1. Create a new function toggleImage()
      1. toggleImage() : void {  
          }
   2. Create a new property to toggle image
      1. showImage : boolean = false;  
          toggleImage() : void {  
          this.showImage = !this.showImage;  
          }
   3. Bind the event to button
      1. <button (click)='toggleImage()' class="btn btn-primary">Show image</button>
   4. Apply the condition in <img /> using \*ngIf
      1. <img \*ngIf='showImage' />
6. Toggle the button text using javascript conditional operator
   1. <button (click)='toggleImage()' class="btn btn-primary">  
       {{showImage ? 'Hide Image' : 'Show Image'}}  
       </button>

**Two way binding**

1. Use ngModel directive to specify two way binding
2. Syntax **[(ngModel)] = ‘property name’;**
3. ngModel is enclosed in square bracket [] to indicate property binding
   1. So the value property created inside the component class will be bind to the ngModel
4. ngModel is again enclosed in parentheses () to indicate event binding
   1. So the user entered data will be send back to the property in the component class
5. So let's create a new in the component class
   1. listFilter : string = 'cart';
6. Let's use the same inside the filter input element and filtered by
   1. <input type="text" [(ngModel)] = 'listFilter' />
   2. <h3>Filtered by : {{listFilter}}</h3>
7. It throws an error once we try to run this
8. For using ngModel, its needs to included in the AppModule
   1. import { FormsModule } from '@angular/forms';
   2. imports: [   
       BrowserModule,  
       FormsModule  
       ]

**Pipes**

1. Allows to transform data
2. {{product.productName | uppercase}}
3. {{product.productCode | lowercase}}
4. {{product.price | currency:'INR':true | lowercase}}

**Interfaces**

1. We can create interfaces to strongly type a property, when there is no predefined type for a property
2. In the above example the products array we created does not have a predefined property
3. We used products : any[] = []; in this case we can make use of interface
4. An interface is a specification identifying a related set of properties and methods
5. Creating an interface in a separate file in products folder
6. Use interface keyword to create an interface
7. export interface IProduct {  
    productId: number,  
    productName: string,  
    productCode: string,  
    releaseDate: string,  
    description: string,  
    price: number,  
    starRating: number,  
    imageUrl: string  
   }
8. To use the interface as a data type, import the interface inside the product-list.component.ts file
   1. import {IProduct} from './product';
9. Now in the component class replace any[] with IProduct[]

**Encapsulating component styles**

1. styles: array of inline styles
2. styleUrl : array of multiple style urls to be applied for the component
3. You use both the properties inside the component decorator
   1. @Component({  
       selector:'ml-products',  
       templateUrl:'app/products/product-list.component.html',  
       styles : ['thead { color:#FF0000;}']  
      })

**Component life cycle hooks**

1. Angular provides a set of life cycle hooks we can use to tap into component life cycle and perform operations.
2. The most commonly used life cycle hooks  
   1. OnInit : perform component initialization, retrieve data etc
   2. OnChanges : called when a change happens
   3. OnDestroy : called when component gets destroyed
3. To use the we need to
   1. import the interface for the life cycle hooks  
      import {Component, OnInit} from '@angular/core';
   2. Make the component class to implement the interface  
      export class ProductListComponent implements OnInit {}
   3. Each life cycle hook has an associated function which is the **“ng”** followed by life cycle hook name.  
      1. OnInit = ngOnInit()
      2. OnChanges = ngOnChanges()
   4. Define the life cycle hook function  
      ngOnInit() : void {  
       console.log('OnInit called');  
       };

**Building a custom pipe**

1. To create a custom pipe let's create a new file product-filter.pipe.ts inside products folder
2. Create the pipe class by implementing **PipeTransform interface**export class ProductFilterPipe implements PipeTransform {  
      
   }
3. Add the pipe decorator  
   @Pipe({  
    name : 'productFilter'  
   })
4. Import both decorator and interface used in the pipe  
     
   import { PipeTransform, Pipe } from '@angular/core';
5. As said before each interface has a method associated with it  
     
   transform(value:IProduct[], filterBy:string) : IProduct[] {  
    return value;  
   }  
   1. First parameter : products array
   2. Second parameter : search string
6. Since we are using IProduct[] as the data type for products array import the same.  
   import {IProduct} from './product';
7. Inside the **transform function,** use javascript array filter function to filter array of items based on a user defined function
8. The custom pipe file will finally look like
   1. import { PipeTransform, Pipe } from '@angular/core';  
        
      import {IProduct} from './product';  
        
      @Pipe({  
       name : 'productFilter'  
      })  
      export class ProductFilterPipe implements PipeTransform {  
         
       transform(value:IProduct[], filterBy:string) : IProduct[] {  
       return filterBy ? value.filter(function(items) {  
       if(items.productName.toLowerCase().indexOf(filterBy.toLowerCase())!= -1) {  
       return value;  
       }  
       }) : value;  
       }  
         
      }
9. To use it in our html file
   1. <tr \*ngFor='let product of products | productFilter:listFilter'></tr>
   2. **customPipe : filter variable**
10. After creating the custom pipe to make it available for the application import and include it inside AppModule declarations array  
      
    import { ProductFilterPipe } from './products/product-filter.pipe';  
      
    declarations: [   
     AppComponent,  
     ProductListComponent,  
     ProductFilterPipe  
    ]

**Services**

1. Create a service class  
     
   export class ProductService {  
   }
2. Define the metadata with a decorator (@Injectable())
3. Import the decorator  
   import { Injectable } from '@angular/core';
4. Define a function inside the service class to retrieve products  
   getProducts() : IProduct[] {  
    return [  
    {  
    "productId": 1,  
    "productName": "Leaf Rake",  
    "productCode": "GDN-0011",  
    "releaseDate": "March 19, 2016",  
    "description": "Leaf rake with 48-inch wooden handle.",  
    "price": 19.95,  
    "starRating": 3.2,  
    "imageUrl": "http://openclipart.org/image/300px/svg\_to\_png/26215/Anonymous\_Leaf\_Rake.png"  
    },  
    {  
    "productId": 2,  
    "productName": "Garden Cart",  
    "productCode": "GDN-0023",  
    "releaseDate": "March 18, 2016",  
    "description": "15 gallon capacity rolling garden cart",  
    "price": 32.99,  
    "starRating": 4.2,  
    "imageUrl": "http://openclipart.org/image/300px/svg\_to\_png/58471/garden\_cart.png"  
    },  
    ];  
    }
5. To use service in a component it needs to imported first  
   import {ProductService} from './product.service';
6. To register a service we need to specify the service as a **provider** in the **component decorator** where we are using it.  
     
   @Component({  
    selector:'ml-products',  
    templateUrl:'app/products/product-list.component.html',  
    styles : ['thead { color:#FF0000;}'],  
    providers : [ProductService]  
   })
7. Injecting the service in product-list.component.ts file
   1. Specify the dependency inside the constructor function of the class  
      constructor(private \_productService : ProductService) {  
       this.products = this.\_productService.getProducts();  
      }
   2. Either you set the products array inside the constructor like above or  
      ngOnInit() : void {  
       this.products = this.\_productService.getProducts();  
      };

**Retrieving data using HTTP Promise**

1. import HttpModule in AppModule, include in imports array  
   import { HttpModule } from '@angular/http';  
     
   imports: [   
    BrowserModule,  
    FormsModule,  
    HttpModule  
    ]
2. Inside product.service.ts file import http service  
   import {Http} from '@angular/http';
3. Specify the dependency inside the constructor of service function  
    constructor(private \_http : Http) {  
      
    }
4. Modify the getProducts() function to return a promise  
   getProducts() : Promise<IProduct[]> {  
    return this.\_http.get('api/products/products.json')  
    .map((response) => <IProduct[]>response.json()).toPromise();  
    }  
     
   **Note : map method takes the raw http response object returned by the http get method and translates into an array of products.**
5. To use **toPromise()** function we need to import the operator  
   +**import 'rxjs/add/operator/toPromise';**
6. To use **map()** function it needs to be imported  
   **import 'rxjs/add/operator/map';**
7. Inside the map function we have the ES6 arrow function, which let's you create a function in different way
   1. var add = function(x,y) {  
       return x + y;  
      }  
        
      var resp = add(4,5);  
        
      var add = (x,y) => x + y;  
      var resp = add(4,5);  
        
      alert(resp);
8. Modify the OnInit() in product-list.component.ts file to incorporate the changes made in the getProducts()  
     
   ngOnInit() : void {  
    this.\_productService.getProducts().then(products\_resp => this.products = products\_resp);  
    };

**Retrieving data using HTTP Observables**

* Make necessary changes to **product-service.ts** file
* import { Injectable } from '@angular/core';  
  import { Http, Response } from '@angular/http';  
  import { Observable } from 'rxjs/Observable';  
  import 'rxjs/add/operator/map';  
    
  import { IProduct } from './product';  
    
  @Injectable()  
  export class ProductService {  
     
   constructor(private \_http : Http) {  
     
   }  
   getProducts() : Observable<IProduct[]> {  
   return this.\_http.get('api/products/products.json')  
   .map((response: Response) => <IProduct[]>response.json());  
   }  
  }
* **product-list.component.ts** file
* ngOnInit(): void {  
   console.log('Product list component initialized !!!');  
   this.\_productService.getProducts().subscribe(data => this.products = data);  
   }

**Nested components**

1. Let's create a nested component to modify the 5 star rating
2. Put the files in shared folder since it should be shared between multiple components
3. Let's create a new component file star.component.ts inside the shared folder
4. Just like any other component follow the same steps of creating a component  
     
   import {Component} from '@angular/core';  
     
   @Component({  
    selector:'ml-star',  
    templateUrl:'app/shared/star.component.html',  
    styleUrls:['app/shared/star.component.css']  
   })  
   export class StarComponent {  
      
   }
5. star.component.html files has two properties **starWidth**, **rating** used, so let's define the same in the component file  
   export class StarComponent {  
    rating : number = 4;  
    starWidth : number;  
   }
6. The value of **starWidth** has to be calculated based on the rating, so let's implement **OnChanges** life cycle hook to do that.  
     
   import {Component,OnChanges} from '@angular/core';  
     
   export class StarComponent implements OnChanges{  
    rating : number = 4;  
    starWidth : number;  
    ngOnChanges() : void {  
    this.starWidth = this.rating \* 86/5;  
    }  
   }
7. Import the star component inside AppModule  
   import { StarComponent } from './shared/star.component';  
   declarations: [   
    AppComponent,  
    ProductListComponent,  
    ProductFilterPipe,  
    StarComponent  
    ],
8. In product-list.component.html file instead of showing the rating value used the nested component  
   <ml-star></ml-star>
9. Content for html file  
     
   <div class="crop"   
    [style.width.px]="starWidth"  
    [title]="rating">  
    <div style="width: 86px">  
    <span class="glyphicon glyphicon-star"></span>  
    <span class="glyphicon glyphicon-star"></span>  
    <span class="glyphicon glyphicon-star"></span>  
    <span class="glyphicon glyphicon-star"></span>  
    <span class="glyphicon glyphicon-star"></span>  
    </div>  
   </div>
10. Content for css file  
      
    .crop {  
     overflow: hidden;  
    }  
    div {  
     cursor: pointer;  
    }

**Passing data to nested component**

1. If the nested component want to receive a data from the container it must expose a property to the container
2. To expose the property we use **@Input()** (decorator)
3. So first we need to import Input  
     
   import {Component,OnChanges,Input} from '@angular/core';
4. Expose the rating property using @Input() (decorator)  
   @Input() rating : number;
5. Now using property binding we can pass values to rating property  
   <ml-star [rating]='product.starRating'></ml-star>

**Sending data from nested component back to container**

1. If the nested component wants to send the information back to container, nested component needs to **raise an event**
2. The nested component exposes the event using **@Output()** (decorator)
3. The only way for nested component to send the data back to the container is by raising an event and the event payload becomes the data to be passed.
4. Event is defined using **EventEmitter object**
5. Let's import Output , EventEmiiter in the component  
     
   import {Component,OnChanges,Input,Output,EventEmitter} from '@angular/core';
6. Let's create the property which raises the event  
   @Output() ratingClicked : EventEmitter<string> = new EventEmitter<string>();
7. Create a new function which will emit an event  
    onClick() {  
    this.ratingClicked.emit('The rating '+this.rating+' was clicked');  
    }
8. Now call the onClick() function when a click event happens in the nested component  
   (click)='onClick()'
9. Now do the event binding in nested component used in the product-list.component.html file.  
     
   <ml-star [rating]='product.starRating' **(ratingClicked)='ratingClicked($event)'**></ml-star>
10. In the above statement **ratingClicked on the left side** is the **property exposed** by the nested component and **ratingClicked on the right side** is the function which needs to be created inside the product-list.component to receive the data.  
      
    ratingClicked(message:string) : void {  
     alert(message);  
    }
11. Object can also be passed along with event by specifying the datatype as **any**

**Communication between components using template reference variables**

* Below example shows how to access method / variables inside child component from parent component.  
  + <div \*ngFor="let product of products">  
     <app-product [product]="product" **#productVar**></app-product>  
     <button class="btn btn-primary" (click)="**productVar.sampleFn()**">Call function inside child component</button> <br /><br />  
     </div>

**Handling null values**

* Code shown below will throw an error if the variable product is not defined.  
  + {{product.starRating}}
* To fix the error with the code above ? symbol can be used.  
  + {{product?.starRating}}

**Hiding elements using hidden property binding**

* The code below hides the element if the condition is not meet.  
  + <div **[hidden]="!product?.starRating"**>  
     Rating : {{product?.starRating}}  
     </div>

**Using ngSwitch & ngSwitchCase**

* <span [ngSwitch]="true">  
   <span \*ngSwitchCase="product?.starRating > 2 && product?.starRating < 3">Okay</span>  
   <span \*ngSwitchCase="product?.starRating > 3 && product?.starRating < 4">Good</span>  
   <span \*ngSwitchCase="product?.starRating > 4 && product?.starRating < 5">Great</span>  
   </span>

**Using ngClass in angular**

* [ngClass]="{'bg-danger': product?.starRating > 2 && product?.starRating < 3,  
   'bg-info' : product?.starRating > 3 && product?.starRating < 4,  
   'bg-primary' : product?.starRating > 4 && product?.starRating < 5}"

**Content projection using ng-content**

* The code for collapse component is shown below.  
  + <div class="well" (click)="toggleInfo()">  
     <h3>{{productName}}</h3>  
     **<ng-content \*ngIf="toggleContent"></ng-content>**  
    </div>
* The code for product component where the collapse component is used.  
  + <**app-collapse-box** [productName]="product?.productName">  
     <div>  
     Product Code : {{product?.productCode}}  
     </div>  
    </**app-collapse-box**>
* In the above example using ng-content collapse component is able to read all the content the display within the same component.

**Multiple slot content projection**

* This type of content projection will help to implement some business logic while performing content projection.  
  + <app-collapse-box>  
     <div **collapse-title**>  
     {{product?.productName}}  
     <i class="glyphicon glyphicon-star" \*ngIf="product?.starRating > 4 && product?.starRating < 5"></i>  
     </div>  
     <div **collapse-body**>  
     <div>  
     Product Code : {{product?.productCode}}  
     </div>  
    </app-collapse-box>
  + <div class="well" (click)="toggleInfo()">  
     **<ng-content select="[collapse-title]"></ng-content>**  
     <**ng-content select="[collapse-body]"** \*ngIf="toggleContent"></ng-content>  
    </div>

**Injection Token for Dependency injection using**

* Let's try to inject toastr service as a dependency for the angular by using **Injection token**
* Let's install the toastr  
  + npm install --save toastr
* Modified angular.json file to include scripts and css files.  
  + "styles": [  
     "src/styles.css",  
     "node\_modules/bootstrap/dist/css/bootstrap.min.css",  
     "node\_modules/toastr/build/toastr.min.css"  
     ],  
     "scripts": [  
     "node\_modules/jquery/dist/jquery.min.js",  
     "node\_modules/toastr/build/toastr.min.js"  
     ]
* Now let's create a service class and create a key or a token that can be used to register with Angular’s dependency injector. So when we need an instance of the toastr class then token or key can be used as a way to lookup the instance.
* Angular provides a mechanism to create key or token to use it with dependency injector without creating a class, which is called **Injection Token**
* Injection token creates a token used for the dependency injection registry in order to find the instance of the service we are looking for.
* Contents for toastr service file.  
  + import { InjectionToken } from '@angular/core';  
      
    export let TOASTR\_TOKEN = new InjectionToken<any>('toastr');
* Service first imports the Injection token and then creates a new Injection token object and exporting the same.
* Once the service is created let's include that in app.module.ts file.  
  + import { TOASTR\_TOKEN } from './common/toastr.service';
  + declare let toastr : any;
  + providers: [  
     {  
     provide : TOASTR\_TOKEN,  
     useValue : toastr  
     }],
* Since **toastr service is not using a class** we use object syntax to include the dependency.
* The declare statement in between tells angular that **toastr** object is created else where and it pointing to it. And the same is used in **useValue** property.

**Adding jQuery using Injection token**

* Let's create a service for injecting jQuery.  
  + import { InjectionToken } from '@angular/core';  
      
    export let JQ\_TOKEN = new InjectionToken<Object>('jQuery');
* Let's modify app.module.ts file to get the reference of jQuery  
  + let jQuery : any = window['jQuery'];
* Let's inject using the provider property inside the decorator  
  + providers: [  
     {  
     provide : JQ\_TOKEN,  
     useValue : jQuery  
     }]

**Adding simple bootstrap modal**

* Let's create a simple modal component using angular CLI
* Once created let's modify the template file to have the html for modal using content projection.  
  + <div id="simple-modal" **class="modal fade"**>  
     <div **class="modal-dialog"**>  
     <div **class="modal-content"**>  
     <div **class="modal-header"**>  
     <button class="close" **data-dismiss="modal"** type="button">X</button>  
     <h4 **class="modal-title"**>  
     **<ng-content select="[modal-title]"></ng-content>**  
     </h4>  
     </div>  
     <div **class="modal-body"**>  
     **<ng-content select="[modal-body]"></ng-content>**  
     </div>  
     </div>  
     </div>  
    </div>
* Now let's update the navbar component to have the modal included.  
  + <**app-simple-modal**>  
     <div **modal-title**>  
     Products filtered by : {{searchProduct}}  
     </div>  
     <div **modal-body** class="list-group" \*ngIf="filteredProducts">  
     <a class="list-group-item" \*ngFor="let product of filteredProducts">  
     {{product.productName}}  
     </a>  
     </div>  
     <div **modal-body** class="alert alert-danger" **\*ngIf="filteredProducts.length==0"**>  
     No matching records found  
     </div>  
    </**app-simple-modal**>
  + In the above code modal-title and modal-body are used for content projection.
* Now to invoke the the modal, let's create a **directive** and add the directive to a button.  
  + ng g directive common/modal-trigger
* Once the directive is created let's modify the directive to include jQuery and reference of the current element (button) to attach event listener, so that the modal can be invoked when the button click event gets triggered.  
  + import { Directive, OnInit, **Inject, ElementRef** } from '@angular/core';  
    **import { JQ\_TOKEN } from './j-query.service';**  
      
    @Directive({  
     selector: '[appModalTrigger]'  
    })  
    export class ModalTriggerDirective implements OnInit {  
      
     **private elem : HTMLElement;**  
      
     constructor(**private ref : ElementRef**, **@Inject(JQ\_TOKEN) private jQuery : any**) { }  
      
     ngOnInit() {  
     **this.elem = this.ref.nativeElement;**  
     this.elem.addEventListener('click', () => {  
     **this.jQuery('#simple-modal').modal({});**  
     });  
     }  
      
    }

**Optimizing modal directive for multiple modals by making ID dynamic**

* Let's introduce an Input variable for simple modal component  
  + <app-simple-modal **modalId="searchProducts"**>  
     <div **modal-title**>Sample title</div>  
     <div **modal-body**>Sample content for modal element!!!</div>  
    </app-simple-modal>
* Now let's add the Input variable to simple modal component.  
  + @Input() modalId : string;
  + In the template file let's use the input variable.
  + <div **id="{{modalId}}"** class="modal fade" tabindex="-1">  
     <div class="modal-dialog">  
     <div class="modal-content">  
     <div class="modal-header">  
     <button class="close" data-dismiss="modal" type="button">X</button>  
     <h4 class="modal-title">  
     <ng-content select="[modal-title]"></ng-content>  
     </h4>  
     </div>  
     <div class="modal-body">  
     <ng-content select="[modal-body]"></ng-content>  
     </div>  
     </div>  
     </div>  
    </div>
* Now let's introduce the ID to modal directive **appModalTrigger**  
  + <button type="submit" class="btn btn-primary" **appModalTrigger="searchProducts"**>Submit</button>
  + In the above code ID is passed as a value to directive.
  + In the directive class the same ID can be accessed an Input variable.
  + **@Input('appModalTrigger') modalId : string;**
  + ngOnInit() {  
     this.elem.addEventListener('click', (e) => {  
     **this.jQuery(`#${this.modalId}`).modal({});**  
     });  
     }

**Routing**

1. Import the RouterModule inside AppModule  
   import { RouterModule } from '@angular/router';
2. Specify RouterModule inside declarations array  
     
   imports: [   
    BrowserModule,  
    FormsModule,  
    HttpModule,  
    RouterModule.forRoot([  
    {path:'products', component:ProductListComponent},  
    {path:'welcome', component:WelcomeComponent},  
    {path:'', redirectTo:'welcome', pathMatch:'full'}, // default path  
    {path:'\*\*', redirectTo:'welcome', pathMatch:'full'} // invalid path  
    ])  
    ]
3. Add the welcome component in AppModule
4. So the final AppModule would look like  
     
   import { NgModule } from '@angular/core';  
   import { BrowserModule } from '@angular/platform-browser';  
   import { FormsModule } from '@angular/forms';  
   import { HttpModule } from '@angular/http';  
   import { RouterModule } from '@angular/router';  
     
     
   import { AppComponent } from './app.component';  
   import { WelcomeComponent } from './home/welcome.component';  
   import { ProductListComponent } from './products/product-list.component';  
   import { ProductFilterPipe } from './products/product-filter.pipe';  
   import { StarComponent } from './shared/star.component';  
     
   @NgModule({  
    imports: [   
    BrowserModule,  
    FormsModule,  
    HttpModule,  
    RouterModule.forRoot([  
    {path:'products', component:ProductListComponent},  
    {path:'welcome', component:WelcomeComponent},  
    {path:'', redirectTo:'welcome', pathMatch:'full'},  
    {path:'\*\*', redirectTo:'welcome', pathMatch:'full'}  
    ])  
    ],  
    declarations: [   
    AppComponent,  
    ProductListComponent,  
    ProductFilterPipe,  
    StarComponent,  
    WelcomeComponent  
    ],  
    bootstrap: [ AppComponent ]  
   })  
   export class AppModule { }
5. Change the template property of AppComponent  
     
   template:`  
    <div class="navbar navbar-default">  
    <ul class="nav navbar-nav">  
    <li>  
    <a **[routerLink]="['/welcome']"**>Welcome</a>  
    </li>  
    <li>  
    <a **[routerLink]="['/products']"**>Products</a>  
    </li>  
    </ul>  
    </div>  
    <div>  
    <h1>{{pageTitle}}</h1>  
    **<router-outlet></router-outlet>**  
    </div>
6. Use routerLink to specify the URL and **router-outlet** to place the rendered views
7. Add <base href='/' /> in the index.html file to specify the default URL

**Pass parameters through URL**

1. Add a new route with params in RouterModule array  
   {path:'products/:id/:username', component:ProductDetailComponent},
2. Let's link the product name in product details component to product details page  
     
   <a [routerLink]="['/products', product.productId, 'marlabs']">{{product.productName}}</a>
3. To access the routeParams we need inject **ActivatedRoute** as dependency to the component
4. The product detail component will finally look like  
   import {Component, OnInit} from '@angular/core';  
   **import {ActivatedRoute, Router} from '@angular/router';**  
     
     
   @Component({  
    template:`  
    <div>  
    <h1>Product detailed page</h1>  
    <h3>{{pageTitle}} - {{productNumber}}</h3>  
    </div>  
    `;  
   })  
   export class ProductDetailComponent implements OnInit{  
    constructor(**private \_activateRoute : ActivatedRoute, private router: Router**) {  
    }  
    pageTitle : string = "Product No";  
    productNumber : number;  
    ngOnInit() : void {  
    this.productNumber = **this.\_activateRoute.snapshot.params['id']**+' = '+**this.\_activateRoute.snapshot.params['username']**;  
     
    /\*\* can use \*/  
    this.router.navigate(['/product-list']);  
    }  
   }
5. Using observable to handle the params  
     
   this.\_activeRoute.params.subscribe(data=> {  
    console.log(data['pId']);  
    })
6. Observables are needed to handle url params, if the component is not changing is state.

**Using trackby in angular 2**

* <tr \*ngFor="let product of products | productFilterPipe:listFilter; let pId=index;"></tr>
* Instead of using trackBy, create a variable and store the index value to that variable.

**Using angular CLI**

* Website : <https://cli.angular.io/>
* Install cli using : **npm install -g @angular/cli**
* Create new project using : **ng new project\_name**
* Building app : **ng build**
* For production use : **ng build --prod**

**Nested router outlets**

* Adding few routes in ap.module.ts file  
  + RouterModule.forRoot([  
     { path: 'home', component: WelcomeComponent },  
     **{ path: 'users',   
     component: ListuserComponent,  
     children: [  
     { path: 'edit/:id', component: EdituserComponent }  
     ]  
     }**,  
     { path: 'create', component: CreateuserComponent },  
     { path: 'show/:id', component: ShowuserComponent, outlet:"viewuser" },  
     { path: '', redirectTo: 'home', pathMatch: 'full' },  
     { path: '\*\*', component: PageNotFoundComponent }  
     ], { useHash: true })
* The code for list component.ts file is shown below.  
  + <div>  
      
     <p \*ngIf="showLoader">  
     Loading users, Please wait!!!  
     </p>  
      
     <table \*ngIf="!showLoader" class="table table-bordered">  
     <tr>  
     <th>  
     Username  
     </th>  
     <th>  
     Location  
     </th>  
     <th></th>  
     <th></th>  
     </tr>  
     <tr \*ngFor="let user of a\_users let userid = index;">  
     <td>{{user.username}}</td>  
     <td>{{user.location}}</td>  
     <td>  
     <a [routerLink]="['/show', userid]">View User</a>  
     </td>  
     <td>  
     **<a [routerLink]="['edit', userid]">Edit User</a>**  
     </td>  
     </tr>  
     </table>  
     **<router-outlet></router-outlet>**  
    </div>
* Once the user clicks on the edit user link, the EditComponents gets loaded in the router-outlet within the list component file.
* For creating such nested routes, we need to specify the routes as the child routes in **app.module.ts** file

**Route Guards**

* Angular route guards are interfaces which can tell the router whether or not it should allow navigation to a requested route
* Different types of route guards  
  + CanActivate
  + CanActivateChild
  + CanDeactivate
  + CanLoad
  + Resolve
* Let's look at canActivate route guard to protect our routes.
* To create a route guard service, please use the command below.  
  + ng g guard guard-name
  + ng g guard auth -- for authentication
* Example for a route guard is show below.  
  + import { Injectable } from '@angular/core';  
    import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router } from '@angular/router';  
    import { Observable } from 'rxjs/Observable';  
    import { AuthService } from './auth.service';  
      
    @Injectable()  
    export class AuthGuard implements CanActivate {  
     constructor(private router : Router,  
     private \_authService : AuthService) {}  
      
     canActivate(  
     next: ActivatedRouteSnapshot,  
     state: RouterStateSnapshot): Observable<any> | Promise<boolean> | boolean {  
     if(this.\_authService.checkLogin()) {  
     return true;  
     }  
     this.router.navigate(['login']);  
     return false;  
     }  
    }
* Once the route guards class is created, let's add it to the routes which needs to be validated.
* It can be applied at the parent level  
  + { path: 'users',   
     component: ListuserComponent,  
     canActivate:[AuthGuard],  
     children: [  
     { path: 'edit/:id', component: EdituserComponent }  
     ]  
     }
* Or at the individual routes level  
  + { path: 'create', component: CreateuserComponent, canActivate:[AuthGuard]},  
      
    { path: 'show/:id', component: ShowuserComponent, canActivate:[AuthGuard]}
* Both canActivate and canActivateChild can be applied for routes that has child routes.  
  + RouterModule.forRoot([  
     { path: 'home', component: HomeComponent },  
     { path: 'products', component: ListComponent,  
     canActivate:[AuthGuard],  
     canActivateChild:[AuthGuard],  
     children: [  
     { path:'edit/:pId', component: EditComponent},  
     { path:'create', component: CreateComponent}  
     ]  
     },  
     { path:"login", component:LoginComponent },  
     { path: '',redirectTo: '/home',pathMatch: 'full'},  
     { path: '\*\*', component: PageNotFoundComponent }  
     ])
* Example for route guard which has both canactivate and canactivatechild.  
  + canActivate(  
     next: ActivatedRouteSnapshot,  
     state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {  
      
     if(this.\_authService.authCheck()) {  
     return true;  
     } else {  
     this.\_router.navigate(['/login']);  
     return false;  
     }  
      
     }  
      
     canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {  
     return this.canActivate(route, state);  
     }

**Candeactivate Route Guard**

* import { Injectable } from '@angular/core';  
  import { ActivatedRouteSnapshot, RouterStateSnapshot, CanDeactivate } from '@angular/router';  
  import { Observable } from 'rxjs';  
  **import { CreateComponent } from '../products/create/create.component';**  
    
  @Injectable({  
   providedIn: 'root'  
  })  
  export class AuthGuard implements CanDeactivate<CreateComponent> {  
     
   canDeactivate(**component : CreateComponent**) {  
   if(**component.checkGuard**) {  
   return true;  
   } else {  
   **return window.confirm('Are you sure that you want to leave without saving** changes?');  
   }  
   }  
  }

**Styling Active Links in routes**

* The code below uses routerLinkActive and routerLinkActiveOptions to style active links in the routes  
  + <ul class="nav navbar-nav">  
     <li **routerLinkActive="active" [routerLinkActiveOptions]="{exact:true}**"><a href="#" [routerLink]="['/products']">Products</a></li>  
     <li **routerLinkActive="active" [routerLinkActiveOptions]="{exact:true}**"><a href="#" [routerLink]="['/create']">Create Products</a></li>  
     </ul>

**JWT integration**

* Install JWT using   
  + npm install jsonwebtoken
* Require jwt in your NodeJS application  
  + jwt = require('jsonwebtoken')
* Let's create a REST api for authenticating user.  
  + app.post('/authenticate', function(req, res) {  
     var token = jwt.sign({'uname':req.body.username}, config.secret\_key, {  
     expiresIn: config.token\_expiry  
     });  
     res.send({token, username:req.body.username});  
    });
* Once the username and password is received create the JWT token using **jwt.sign(),** and finally send the token back to the client.
* Once for all the request to be authenticate, it needs to pass this token.
* jwt.sign(), we are passing the payload, secret\_key, token\_expiry from config.
* Let's create the module to return expiry and secret\_key.
* Config.js file  
  + module.exports = {  
     'secret\_key':'marlabs-secret-key',  
     'token\_expiry':'1h'  
    }
  + Require the module in your application.  
    - config = require('./config')
* Let's create a new module to authenticate the routes. **api.js**  
  + var jwt = require('jsonwebtoken'),  
     config = require('./config');  
      
    exports.authenticate = function(req, res, next) {  
     var token = req.body.token || req.query.token || req.headers['authorization'];  
     if (token) {  
     jwt.verify(token, config.secret\_key, function (err, decoded) {  
     if (err) {  
     return res.status(201).json({ success: false, message: 'Authenticate token expired, please login again.', errcode: 'exp-token' });  
     } else {  
     req.decoded = decoded;  
     next();  
     }  
     });  
     }  
    }
* We are expecting the token to be part of the request either in body / query / header.
* Once the token is received, it needs to be verified which is done using **jwt.verify()**
* So let's use this module in our application so that the routes gets validated.  
  + api = require('./api')
  + app.post('/authenticate', function(req, res) {  
     var token = jwt.sign({'uname':req.body.username}, config.secret\_key, {  
     expiresIn: config.token\_expiry  
     });  
     res.send({token, username:req.body.username});  
    });  
      
    **app.use(api.authenticate);**  
      
    app.get('/getusers', function(req, res) {  
     setTimeout(function() {  
     res.send(users);  
     }, 500);  
    });
* Once the authenticate is called which is highlighted in bold, it will authenticate all the routes defined after that.
* Now let's see how to integrate the JWT token in the client side, so that it can be passed through all the request which needs to be validated.
* Once the token is received in the client we are going to use **CookieService** from **ngx-cookie-service** to store the token in the cookie.
* Let's install ngx-cookie-service  
  + npm install ngx-cookie-service --save
* Once installed import the library in app.module so that all the components can use.  
  + import { CookieService } from 'ngx-cookie-service';
  + providers: [  
     CookieService  
    ]
* Once the token is received as a response to the request, let's save the token using cookieservice.  
  + this.\_http.post('http://localhost:8081/authenticate', this.login).subscribe(data=> {  
     if(data['token']) {  
     **this.cookieService.set('mToken', data['token']);**  
     this.\_router.navigate(['/']);  
     }  
     });
* So now the token can be accessed using the **cookieService**
* Let's add it to the header section of the request which needs to be authenticated.
* We have a function which sends a request to the server to get information about users  
  + getUsers() {  
     return this.http.get('http://localhost:8081/getusers', **this.requestoptions**);  
     }
* The above function needs to be authenticated, which means that the request needs to have the token attached to the header section.
* That is what being passed through the **requestoptions** variable.
* **import { HttpInterceptor, HttpHeaders } from '@angular/common/http';**
* Let's look at the code.  
  + jwtToken: string;  
     requestoptions : any;  
      
     newUserSubject = new Subject<any>();  
      
     constructor(private http: HttpClient, private \_authService: AuthService) {  
     this.jwtToken = \_authService.getToken();  
     this.requestoptions = {  
     headers: new **HttpHeaders()**.set('Authorization', this.jwtToken)  
     };  
     }
* So the above code will create a new variable **requestoptions** and stores the header information with JWT token.
* Now whenever you make an api call you can use the requestoptions variable to pass the header information. The functions shown below uses the same.  
  + getUsers() {  
     return this.http.get('http://localhost:8081/getusers', **this.requestoptions**);  
     }
  + saveUser(data) {  
     return this.http.post('http://localhost:8081/saveuser', data, **this.requestoptions**);  
    }
* Now the token will be passed through the request headers will can be verified on the server.

**HttpInterceptor**

* Interface which will transform the outgoing request and pass it to the next interceptor in the chain by calling next.handle(transformedReq)
* Let's see how to configure an HttpInterceptor for our application.
* In the example above we used a variable **requestoptions** which holds the header information to pass the token for each and every outgoing request.
* The problem with the above approach is that we have to pass the this information for each and every request manually.
* Instead we can use HttpInterceptor to configure this in one place so that it gets applied to all the request.
* Include http interceptor inside app.module.  
  + import { HttpClientModule, HTTP\_INTERCEPTORS } from '@angular/common/http';
* Let's create the service  
  + Need to first import httpinterceptor from http module.
  + import { Injectable } from '@angular/core';  
    **import { HttpInterceptor, HttpHeaders } from '@angular/common/http';**  
    import { AuthService } from '../users/auth.service';
  + Create the service class
  + @Injectable()  
    export class AuthInterceptorService implements HttpInterceptor{  
    }
  + Let's add the interface method.
  + intercept(req, next) {  
     return next.handle(req);  
    }
  + In the above method we are not doing anything other than passing the req to the next interceptor in the chain my calling the next.handle() method.
* So now let's add the logic to modify the req to include JWT token in the header.
* We have already imported **AuthService,** so let's inject that in the constructor.  
  + constructor(private \_authService : AuthService) {  
     }
* The code below adds the header information in the request.  
  + intercept(req, next) {  
     this.jwtToken = this.\_authService.getToken();  
     var authRequest = req.clone({  
     headers:new HttpHeaders().set('Authorization', this.jwtToken)  
     });  
     return next.handle(authRequest);  
     }
* Including interceptor service inside providers array in app.module  
  + providers: [  
     AuthGuard,  
     CookieService,  
     AuthService,  
     {  
     provide:HTTP\_INTERCEPTORS,  
     useClass: AuthInterceptorService,  
     multi:true  
     }  
     ]
* Accessing the response from http interceptor.  
  + https://medium.com/@ryanchenkie\_40935/angular-authentication-using-the-http-client-and-http-interceptors-2f9d1540eb8

**Communication between sibling components**

* For communicating between sibling components we make use of rxjs Subject or BehaviorSubject.
* BehaviorSubject can take initial values which differentiates it from Subject.
* It has main two methods next(), subscribe().
* Let's consider two components a list and a create component. Once a new user is added via create component the list component needs to know about this change and update the list.
* For this we will first create a shared service which has methods for create and listing users.  
  + getUsers() {  
     return this.\_http.get('http://localhost:8081/getusers');  
     }
  + saveUser(data) {  
     this.\_http.post('http://localhost:8081/saveuser', data).subscribe(data=>{  
     });  
     }
* We are going to inject the service in both components so that both have access to these methods.
* So the list component is using the **getUsers()** method where as create component is using **saveUser()** method.
* Now once the saveUser() is called we have to notify the list component about this change.
* So let's introduce Subject from rxjs library, which helps us to share data between components.
* Let's create a new instance of Subject.  
  + import { Subject } from 'rxjs/Subject';
  + newUserDataSubject = new Subject<any>();
  + newUserDataSubject$ = this.newUserDataSubject.asObservable();
* So now let's modify the saveUser() method to emit a new observable using the next().  
  + saveUser(data) {  
     this.\_http.post('http://localhost:8081/saveuser', data).subscribe(data=>{  
     this.newUserDataSubject.next(data);  
     });  
     }
* So now once the a user is created an observable is generated with the new data.
* In our list component we need to subscribe to this observable to get the new data in the component.  
  + this.\_dataService.newUserDataSubject$.subscribe(data=>{  
     this.users = data;  
     });

**ViewChild**

* Child components in our view can be accessed from our parent component easily with @ViewChild
* To get access to a component and its methods, we can use the @ViewChild decorator.
* In our previous example from the list component if you want to access the methods of create component or any child component, we could viewChild.
* Just import the child component inside parent component class and make an instance using viewchild.  
  + import { CreateuserComponent } from '../createuser/createuser.component';
  + export class ListuserComponent implements OnInit {  
      
     @ViewChild(CreateuserComponent) **CreateUserChild**: CreateuserComponent;  
      
     constructor(private \_apiService : ApiService) {  
     }  
      
     ngOnInit() {  
     this.**CreateUserChild**.\_apiService.newUserSubject.subscribe(data => {  
     this.a\_users = data;  
     });  
     }  
      
    }

**Http Client**

* Import http client module  
  + import { HttpClientModule } from '@angular/common/http';
  + Supply HttpClientModule to imports array of app module
* Import httpclient to service, components in which it needs to be used.  
  + import { HttpClient } from '@angular/common/http';
* Methods available  
  + this.http.post(url, hero, httpOptions)
  + this.http.get(url);
* Both the methods returns observables

**Observables**

<https://www.youtube.com/watch?v=ZmEvKLYF0os>

Example shown below

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

import { Observable } from 'rxjs/observable';

@Component({

selector: 'app-home',

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

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

})

export class HomeComponent implements OnInit {

observableVar = new Observable((observer) => {

let count = 0;

let interval = setInterval(() => {

observer.next(count++);

}, 1000);

return () => {

clearInterval(interval);

}

});

constructor() { }

ngOnInit() {

let data\_observable = this.observableVar.subscribe((data) => {

if(data==5) {

data\_observable.unsubscribe();

}

console.log('data = '+data);

});

}

startObservable() {

}

}

**Angular 5 service worker**

<https://medium.com/codingthesmartway-com-blog/angular-5-service-worker-b722e571e306>

**View encapsulation**

<https://blog.thoughtram.io/angular/2015/06/29/shadow-dom-strategies-in-angular2.html>

**Hot vs Cold observables**

<https://angular-2-training-book.rangle.io/handout/observables/cold_vs_hot_observables.html>