Wednesday, August 1, 2018 10:35 AM

Angular 2

Advantages
- Powerful data binding
- Modular by design
- Expressive HTML

- Built for speed
- Modern
- Simplified API
- Enhances Productivity

Angular App is a set of components and services.
Component = Template + Class(Properties and Methods)+ Metadata

It has a root angular module with components and can consist of feature angular modules with components as childs.

To add a dependent library project in angular 2
- Create .npmrc file with the registry (Ex. registry=https://artifactory.tools.deloitteinnovation.us/artifactory/api/npm/npm-virtual)
- Add dependency to package.json under dependencies ("connectme-lib": "^9.0.0")

ES 5 -> No transpiling required, runs in the browser
(ECMA6) ES 2015 -> Has new features(classes, let, modules, arrow etc.), it transpiles to ES5 for the browsers to understand
TypeScript -> Superset of JS, Strongly Typed(contains Typescript definition files-*.d.ts), Great IDE tooling, should be transpiled, transpiles to plain JS, Class-based object-orientation, Interfaces can be used

TS Playground
http://www.typescriptlang.org/Playground/

npm - Node Package Manager is a command line utility that interacts with repositories of open source projects, installs libraries, packages and applications

Angular Application Setup
Can be done through CLI or we need the below steps to be executed
- Create folder and add package definition & configuration files
- Install the packages required
- Create app's angular module
- Create main.ts
- Create host web page(index.html)

Cloning GIT repo
https://github.com/DeborahK/Angular-GettingStarted.git
git clone <repo-name>

package.json

devDependencies -> libraries required for development
dependencies -> libraries required to run the app
scripts -> consists of 'start', 'build', test, 'lint', 'e2e'
Ex: '***start***' : '***ng serve -o***' // -o is to run the app in the default browser
In CLI type npm start to run the above command

ES2015 modules

- Code files that import or export something
- Organize our code files
- Modularize our code
- Promote code reuse

Need to name as a module, the file becomes the module itself
Ex.
export class Product {

}
will be transpiled to
function Product() {
}

To import it in another class
import {Product} from './product';

Angular Modules
- Code files that organize the application in cohesive blocks of functionality
- Organize our application
- Modularize our application
- Promote application boundaries

Root Module, Feature Module and Shared Module consists components

Components
- Includes a template to create a view for the app, includes bindings and directives
- class (contains properties for data, methods for logic), created with typescript to support the view

Ex. app.component.ts
```
export class AppComponent {
 pageTitle: string = 'My App';
}
```
- Metadata -> provides additional information of the component to angular, defined with a decorator(function which adds metadata to a class, its members or its method arguments).
- Decorators are prefixed with @ symbol
- Angular provides in-built decorators

@Component is an inbuilt decorator
Ex. for Metadata & template
```
@Component({ // Decorator
  selector: 'app-root', // directive name used in html
  template: `
     <div><h1>{{pageTitle}}</h1></div>
      <div>Content goes here!</div>` // view layout
});
```
Also templateUrl can be used
the path to the html in the templateUrl will be relative to the index.html

app.component.ts
```
import {Component} from '@angular/core';
//Metadata and template
@Component({
  selector: 'app-root',
  template: `
     <div><h1>{{pageTitle}}</h1></div>
      <div>Content goes here!</div>` // ES 2015 Back ticks helps us to define the html code in multiple lines
//templateUrl: './product-list.component.html' // path to the template url instead of defining inline template
});
//Class
export class AppComponent {
 pageTitle: string = 'My App';
}
```
export keyword - > exports the class to be used in some other class, and as it uses export it is an ESModule
```
//Import statement
import {Component} from '@angular/core';
import {<member name>} from '<angular lib module name>' ;
```
Angular is modular
Ex. (@angular/core, @angular/http, @angular/router, @angular/animate)

Bootstrapping out component
- To let angular know where to start
- How does angular know about the directive(custom html element - Ex. app-root as in the above example)?
    - When the compiler sees the directive it looks into an angular module for the definition, angular modules helps us organize our functionality into cohesive blocks provides application boundaries.
    - We also use the bootstrap option to let know Angular on which component to start with. It should use the directive mentioned in index.htm

- Host the application

Contains index.html, which is the main page for the app. This is often the only web page of the app, hence angular app is known as Single Page Application(SPA)

- Defining the angular module

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser'; // required to run application in browser
import {NgModule} from './app.component';
//metadata and decorator
@NgModule({
   imports: [BrowserModule],
   declarations: [AppComponent],
   bootstrap: [AppComponent]
})
//class
export class AppModule {}
```

Life cycle of component
Below are few of the life cycle hooks pf a component
1. Constructor will be executed first
2. ngOnInit
3. AfterContentInit - content will be ready before the view is initialized
4. AfterViewInit
5. ngDestroy - will be called after navigation to another component

Interpolation
- To use the class properties and methods in the html.

Ex.

```
@Component({
 selector: 'app-root',
 template: `
    <div><h1>{{getTitle()}}</h1></div>
    <div><h1>{{pageTitle}}</h1></div>
     <div>Content goes here!</div>` // ES 2015 Back ticks helps us to define the html code in multiple lines
//templateUrl: './product-list.component.html' // path to the template url instead of defining inline template
});
//Class
export class AppComponent {
 pageTitle: string = 'My App';
 getTitle(): string {
  return 'Product List';
 }
}
```

Angular Built-in directives
- Structural Directives (*ngIf, *ngFor, * represents that it is a structural directive)
- Browser Module exposes the ngIf and ngFor directives.
- Ex.

```
<div class="row" *ngFor="let product of products">
     <div class="col col-md-2">{{product.name}}</div>
</div>
```

- for .. Of 'vs' for .. In

ES 2015 has both for .. of and for .. in loop
for .. of is similar to forEach loop
for .. of iterates over iterable objects, such as an array.
Ex.

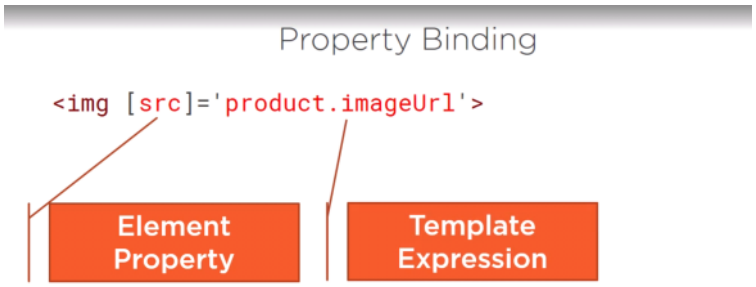
```
let names = ['di', 'boo', 'jay'];
for (let name of names) {
  console.log(name); // di, boo, jay
}
```

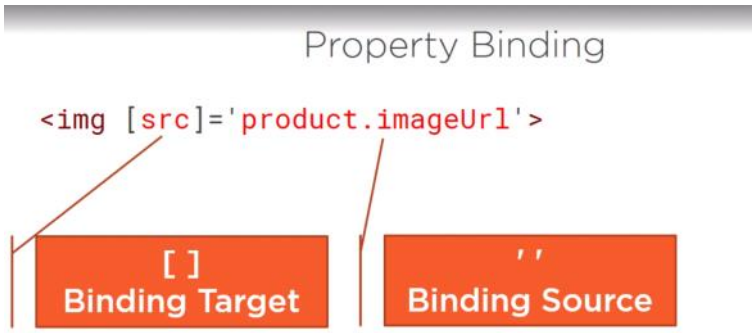
for .. in iterates over properties of object. // iterates the index
Ex.

```
let names = ['di', 'boo', 'jay'];
for (let name of names) {
  console.log(name); // 0, 1, 2
}
```

Data binding

Property binding allows us to set a property element using an template expression.





Property binding can also be done through interpolation

```
<img src={{product.imageUrl}}>

<img src='http://openclipart.org/{{product.imageUrl}}'>
```

Property Bindings Example (1-way binding)

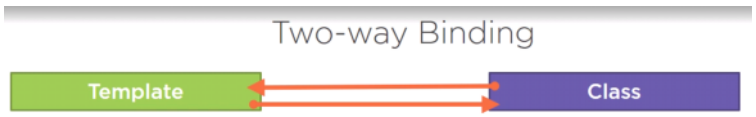
```
<div class="col-md-2 dv-table-column dv-image">
     <img [src]="product.imageUrl" [title]="product.productName" [style.width.px]="imageWidth"
     [style.margin.px]="imageMargin"/>
</div>
import { Component } from '@angular/core';
@Component({
     selector: 'pm-products',
     templateUrl: './product-list.template.html'
})
export class ProductListComponent {
```

```
    pageTitle: string = 'Product List';
    imageWidth: number = 50;
    imageMargin: number = 2;
    products: any[] = [];
}
```

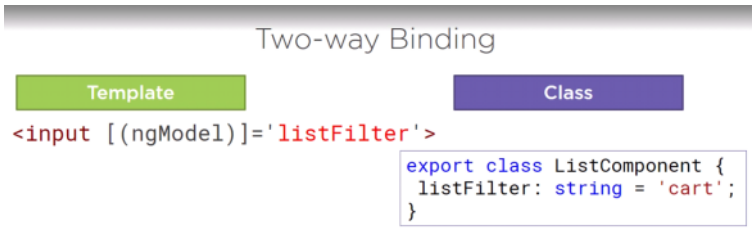
Event Binding Example

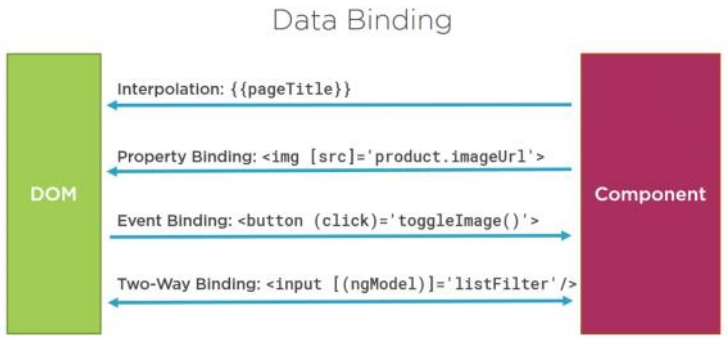
```
<a (click)="toggleImage()">{{showImage ? 'Hide' : 'Show'}} Image</a>

import { Component } from '@angular/core';
@Component({
    selector: 'pm-products',
    templateUrl: './product-list.template.html'
})
export class ProductListComponent {
    pageTitle: string = 'Product List';
    imageWidth: number = 50;
    imageMargin: number = 2;
    products: any[] = [];
    toggleImage(): void {
        this.showImage = !this.showImage;
    }
}
```

Two way Binding Example

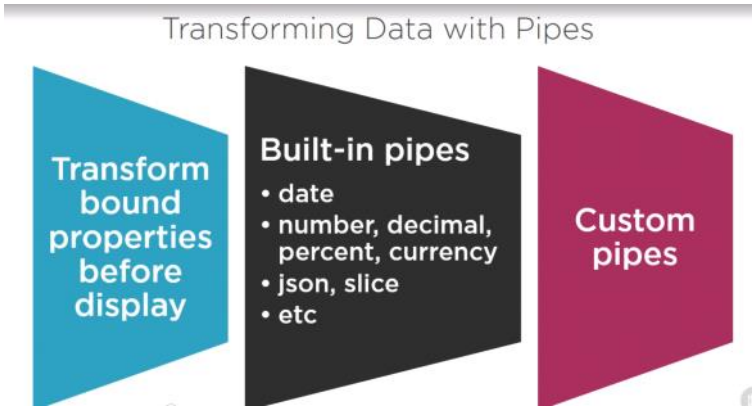


We use ngModel to use two-way binding
ngModel directive is part of FormsModule
Why FormsModule is added to imports array in app.module.ts and not in declarations array ?
- Our directives, filters(pipes) and components are declared in the declarations.
- Directives, components and pipes used from external sources/ 3rd party apps are used in imports array.
[] - indicates property binding from the class property to the input element
{} - indicates event binding to notify the user entered data back to the class property





Transforming data with pipes



Pipe Examples

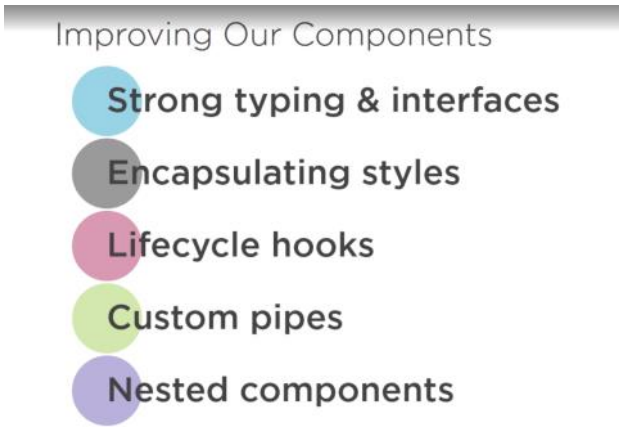
```
{{ product.productCode | lowercase }}

<img [src]='product.imageUrl'
     [title]='product.productName | uppercase'>
```

```
{{ product.price | currency | lowercase }}

{{ product.price | currency:'USD':'symbol':'1.2-2' }}
```

Pipes can also have parameters

Ex. Currency has three parameters
- Desired currency code
- string defining how to show the currency symbol
- digit info(1 - min no. digits to left of decimal, 2 - min of fractional digits to right of decimal, 2 - max no. of fractional digits to right of decimal)

Components



- Strongly typing & interfaces -> Strong typing helps minimize errors through better syntax checking and improved tooling.

If there is no pre-defined type for a property we can define the type ourself using an interface.

- Encapsulating Styles - If the component needs special styles we can encapsulate those styles within the component to ensure that they will not leak out any other component in the application
- Lifecycle hooks - A component has a lifecycle managed by angular. Angular provides a set of lifecycle hooks we can use. To make the component m ore flexible and responsive.
- Custom pipes - Used to transform bound data to be displayed in the view.
- Nested components - To improve the overall quality of the app by reusing the component developed

Strong Typing and Interfaces

- We define interface for custom types
- ES5 and ES2015 does not support interfaces, typescript supports interfaces.
- To specify custom types we use an interface.

Interface

A specification identifying a related set of properties and methods.

A class commits to supporting the specification by implementing the interface.

Use the interface as a data type.

Ex.

Interface Is a Specification

```
export interface IProduct {
    productId: number;
    productName: string;
    productCode: string;
    releaseDate: Date;
    price: number;
    description: string;
    starRating: number;
    imageUrl: string;
    calculateDiscount(percent: number): number;
}
```

Interface Name

interface keyword

- Export keyword helps us to import the interface wherever required.
- To use the interface in a component, we need to import the interface.

Using an Interface as a Data Type

```
import { IProduct } from './product';

export class ProductListComponent {
  pageTitle: string = 'Product List';
  showImage: boolean = false;
  listFilter: string = 'cart';

  products: IProduct[] = […];

  toggleImage(): void {
      this.showImage = !this.showImage;
   }
}
```

Encapsulating Component Styles

Encapsulating Component Styles

styles

```
@Component({
    selector: 'pm-products',
    templateUrl: './product-list.component.html',
    styles: ['thead {color: #337AB7;}']})
```

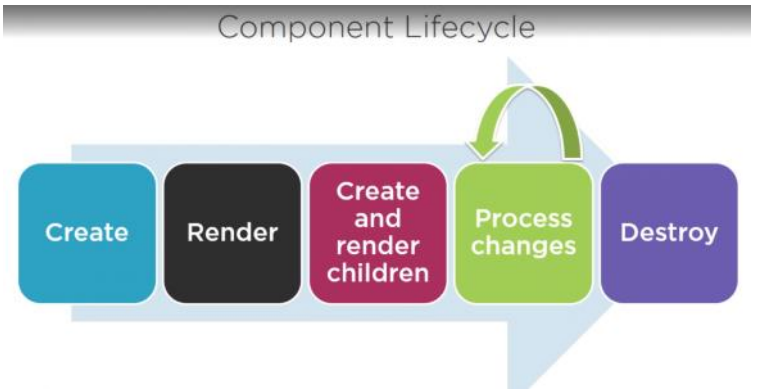
styleUrls

```
@Component({
    selector: 'pm-products',
    templateUrl: './product-list.component.html',
    styleUrls: ['./product-list.component.css']})
```

Styles/styleUrls is an array, so we can use multiple styles in them.

Lifecycle hooks

- The lifecycle hooks are provided by angular as an interface
- We need to import the respective lifecycle hooks and use it in our code
- We really don't have to implement the interface to use lifecycle hooks, we can simply write code for the hooks but it is reco mmended to use it in this way.



Using a Lifecycle Hook

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

  export class ProductListComponent
1                 implements OnInit {
    pageTitle: string = 'Product List';
    showImage: boolean = false;
    listFilter: string = 'cart';
    products: IProduct[] = […];

3   ngOnInit(): void {
        console.log('In OnInit');
    }
  }
```

Building custom pipes

- We use Pipe as a decorator, and we implement the PipeTransform interface
- PipeTransform has the inbuilt transform method has two arguments
  - value - for the value to be transformed
  - Character - character to be replaced

Building a Custom Pipe

```
@Pipe({
    name: 'convertToSpaces'
})
export class ConvertToSpacesPipe
                implements PipeTransform {

  transform(value: string,
            character: string): string{
  }
}
```

Using a Custom Pipe

Template

```
<td>{{ product.productCode | convertToSpaces:'-'}}</td>
```

Pipe

```
transform(value: string, character: string): string {

}
```

Using a Custom Pipe

Template

```
<td>{{ product.productCode | convertToSpaces:'-'}}</td>
```

Module

```
@NgModule({
  imports: [
      BrowserModule,
      FormsModule ],
  declarations: [
      AppComponent,
      ProductListComponent,
      ConvertToSpacesPipe ],
  bootstrap: [ AppComponent ]
})
```

Disadvantages of built-in pipes for filtering values in an array

> "Angular doesn't offer such pipes because they perform poorly and prevent aggressive minification."
>
> angular.io

- Angular recommends filtering logic to be part of the component.
- Using getter and setter method we can achieve the filtering
- Creating getter and setter methods is as below

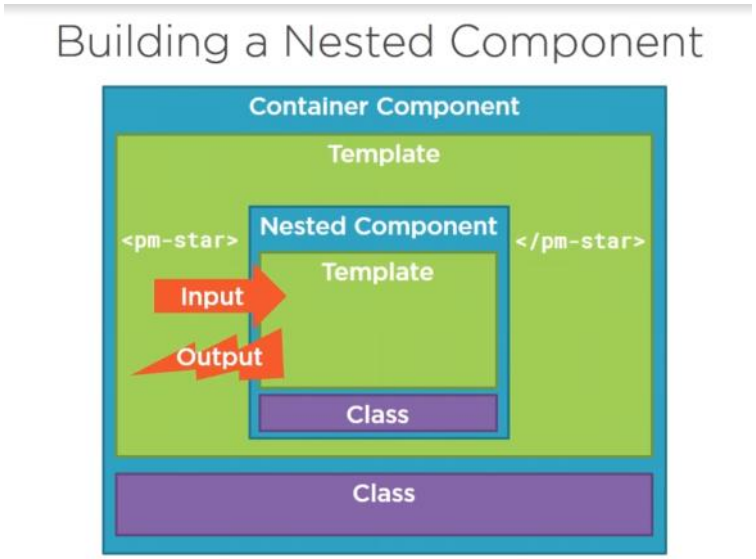
```
_listFilter: string;

get listFilter(): string {
  return this._listFilter;
}
set listFilter(value: string) {
  this._listFilter = value;
  this.filteredProducts = this.listFilter ? this.performFilter(this.listFilter)
}
```

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

@Component({
    selector: 'pm-products',
    templateUrl: './product-list.component.html',
    styleUrls: ['./product-list.component.css']
})
export class ProductListComponent implements OnInit {
    pageTitle: string = 'Product List';
    imageWidth: number = 50;
    imageMargin: number = 2;
    showImage: boolean = false;

    _listFilter: string;
    get listFilter(): string {
        return this._listFilter;
    }
    set listFilter(value:string) {
        this._listFilter = value;
        this.filteredProducts= this.listFilter ? this.performFilter(this.listFilter) : this.products;
    }
```

```
  performFilter(filterBy: string): IProduct[] {
      filterBy = filterBy.toLocaleLowerCase();
      return this.products.filter((product: IProduct) =>
            product.productName.toLocaleLowerCase().indexOf(filterBy) !== -1);
  }
```

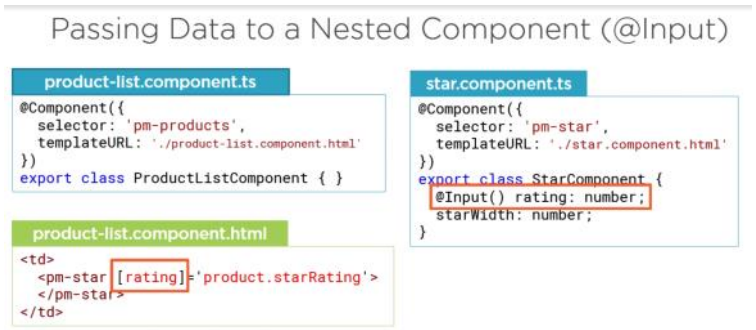
- Best place is to set the default values for the complex properties in in the constructor.
- Constructor is the function that is executed when the component is first initialized.

Building Nested Components

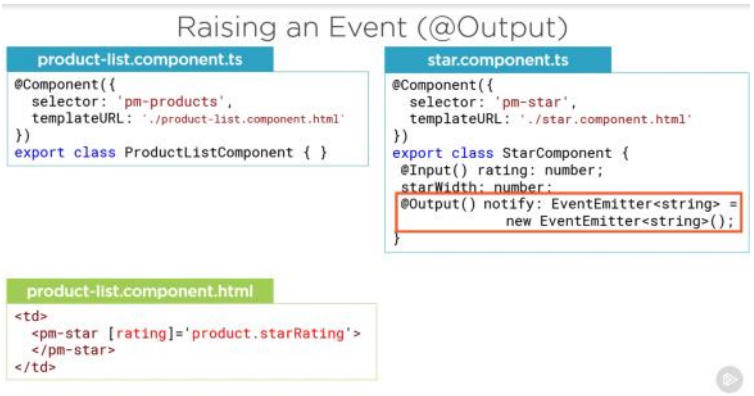


The nested component receives input through properties and send output to the parent container through events.

@Input decorator



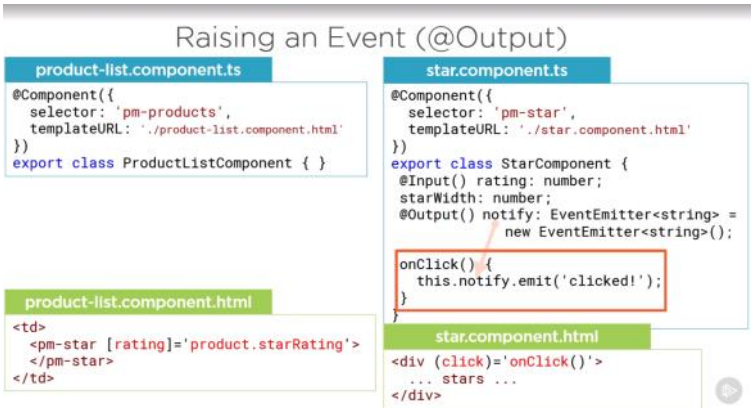
@Output Decorator
The only way data can be passed from nested component to container is through an event.



@Output(): notify: EventEmitter<string> = new EventEmitter<string>();

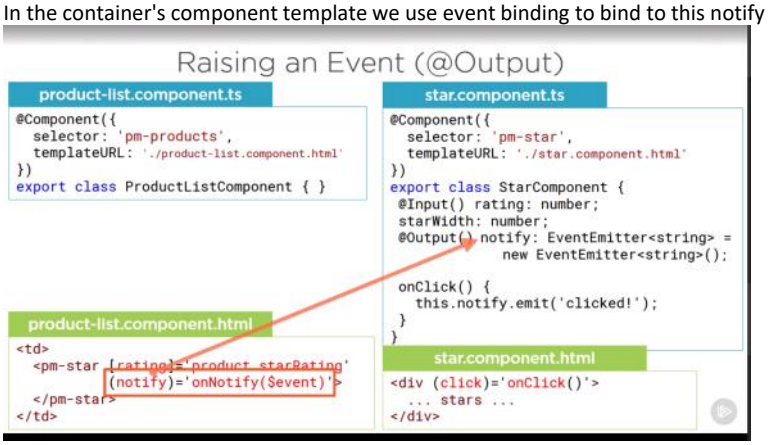
Typescript allows us to work with generics, generics allows us to identify a specific type that the object instance will work with.
When creating an EventEmitter, the generic argument identifies the type of the event payload.
If we want pass a string value to the container in the event payload, we define string here. (Ex. EventEmitter<string>). To pass multiple values we can specify an object.

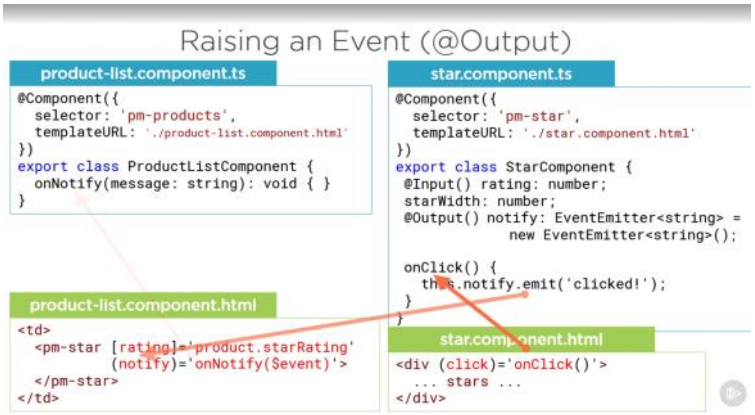
@Output(): notify: EventEmitter<string> = new EventEmitter<string>();
In the above example, we defined a notify event with an string payload.



We use event binding in the star component template to call the star component's onClick method.
In the onClick method, we use the notify event property, and call its emit method to raise the notify event to the container.
If we want to pass the data in the event payload, we pass that data into the emit method.
In the above example, the onClick method raises the notify event and sets the event payload to a string message.
The container component receives that event with the specified payload.

In the container's component template we use event binding to bind to this notify as below

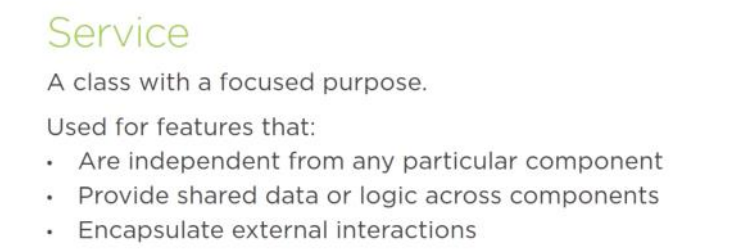


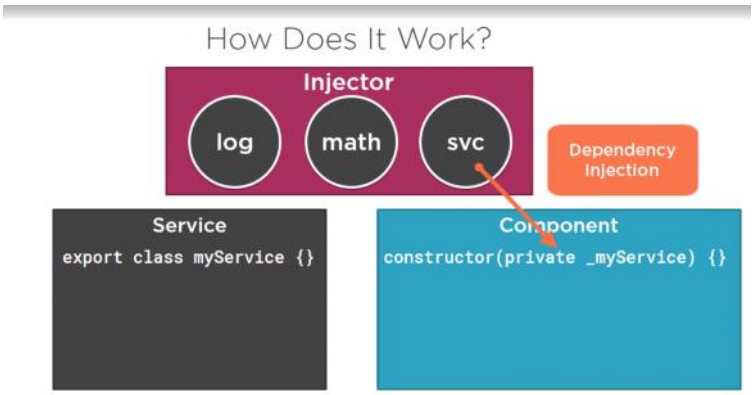


(notify)='onNotify($event)' => $event passes along any information associated with a generated event.

More in Angular Component Communication plural sight course.

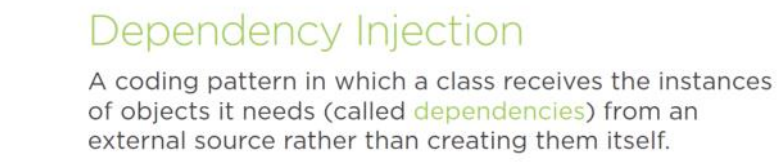
Services and Dependency Injection





As in the above example, the component can work with services in two ways

- The component can create an instance of the service (Ex. *let svc = new myService();*), but this is local to the component and it can't share data and other resources, also it will be difficult to mock the service for testing.
- Alternatively we can register the service with angular, Angular creates a single instance of the service class, called a sing leton and holds onto it. Specifically angular provides a built-in injector. We register our services with the angular injector, which maintains a container of the created service instances. The injector creates and manages the single instance, or singleton of each registered service as required.
  In the above example the angular injector is managing instances of three different services (log, math and svc -myService). If our component needs a service, the component class defines the service as a dependency. The Angular injector then provides, or injects, the service class instance when the component class is instantiated. This process is called Dependency Injection. Since angular manages the single instance, any data or logic in that instance is shared by all the classes that use it.



In Angular, the external source is the Angular injector.

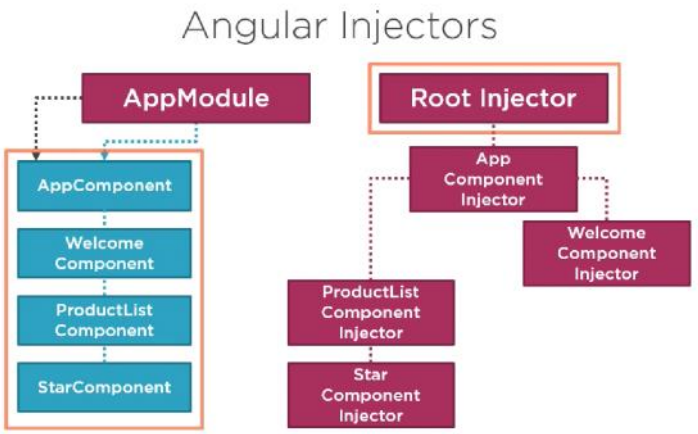
Building a service

product.service.ts

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

@Injectable()
export class ProductService {

  getProducts(): IProduct[] {
  }

}
```

@Injectable() // Decorator for the service metadata

Registering a service



Registering a Service

Root Injector	Component Injector
Service is available throughout the application	Service is available ONLY to that component and its child (nested) components
Recommended for most scenarios	Isolates a service used by only one component
	Provides multiple instances of the service

Registering a Service - Root Application

product.service.ts

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

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

  getProducts(): IProduct[] {
  }

}
```

@Injectable ({
providedIn: 'root'
)} // This code makes the service available to the entire application
//providedIn feature is available from Angular 6 and above

In older versions the service registered in a module as below

app.module.ts

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

But it is recommended practice to use providedIn feature in the service instead. This provides better "Tree shaking".
Tree shaking is a process whereby the Angular compiler shakes out unused code for smaller deployed bundles.

app.module.ts

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

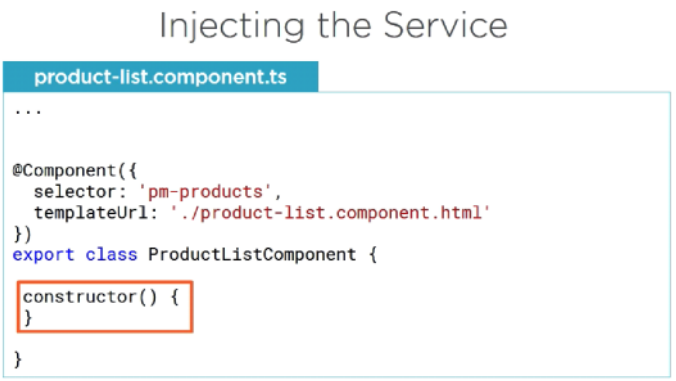
product-list.component.ts

```
@Component({
  templateUrl: './product-list.component.html',
  providers: [ProductService]
})
export class ProductListComponent { }
```

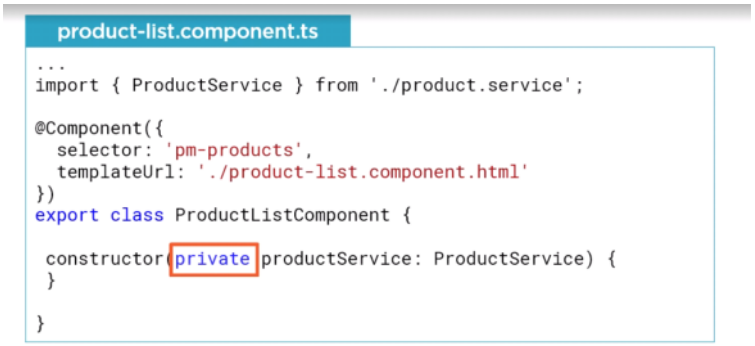
- To register the service for a specific component and its child , we register the service in the component through providers.(Ex. providers:

[ProductService] as in the above screenshot)

Injecting a service



- Every class has a constructor which is executed when the instance of the class is created.
- If there is no explicit constructor defined for the class, an implicit constructor is used.
- But, to inject dependencies such as an instance of the service, we need an explicit constructor.
- Services are injected via constructor, which is used for initialization of the services.
- We identify our dependencies by specifying them as parameters to the constructor function, as below



- In the constructor function we identify the dependency and assign it to a local variable(productService).
- We use the accessor keyword to the service
- The constructor function will be executed before the onInit

Checklist: Registering a Service

Select the appropriate level in the hierarchy
- Root application injector if the service is used throughout the application
- Specific component's injector if only that component uses the service

Service Injectable decorator
- Set the `providedIn` property to `'root'`

Component decorator
- Set the `providers` property to the service

Checklist: Dependency Injection

Specify the service as a dependency

Use a constructor parameter

Service is injected when component is instantiated

Retrieving data using HTTP

- Reactive Extensions(RxJs) represent a data sequence as an observable sequence, commonly called as an observable.

Observables and Reactive Extensions

-

Reactive Extensions (RxJS)

Help manage asynchronous data

Treat events as a collection
- An array whose items arrive asynchronously over time

Subscribe to receive notifications

Are used within Angular

Observable Operators

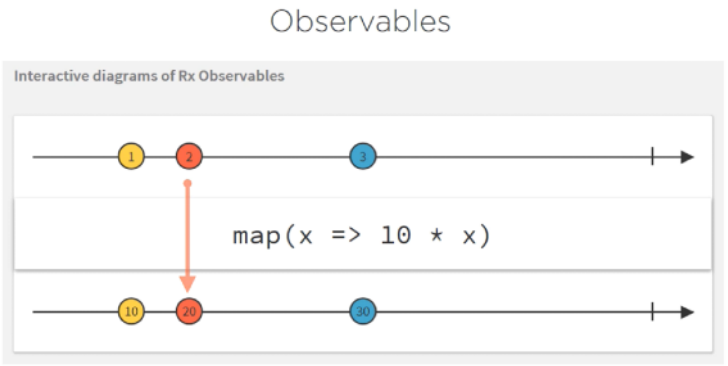
-

Methods on observables that compose new observables

Transform the source observable in some way

Process each value as it is emitted

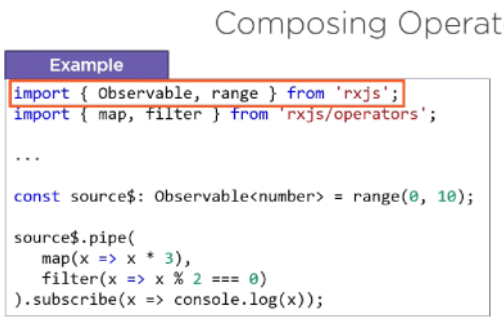
Examples: map, filter, take, merge, ...

- 

- Map allows us to transform the incoming data, argument to map operator is an arrow function that tells to take each data item and transform it to 10 times its value as in the above example. 1 will be mapped to 10, 2 will be mapped to 20. This depiction is called a Marble di agram and is helpful for visualizing observable sequences. The above is a screenshot from RxMarbles.

- 

- We use pipe methods to compose multiple observable operators.
- Rx has two primary packages, one is Observable and the other is Observable creation like range.
- Operators like map, filter are imported from rxjs/operators.
- Observable<number> = range(0,10) will generate observable stream of numbers from 0 to 9.

- 

- By convention we add a $ as suffix to the variables that hold an observable. Also we can distinguish observables in the code.
- As in the above example we can use multiple operators(map, filter) separated by commas.
- Map results in 0,3,6,9,etc … and filter makes sure none of the mapped results has a reminder 0 when divide by 2.
- Subscribe does not emit the values until it has a subscriber.(Result : 0, 6, 12, 18, 24)

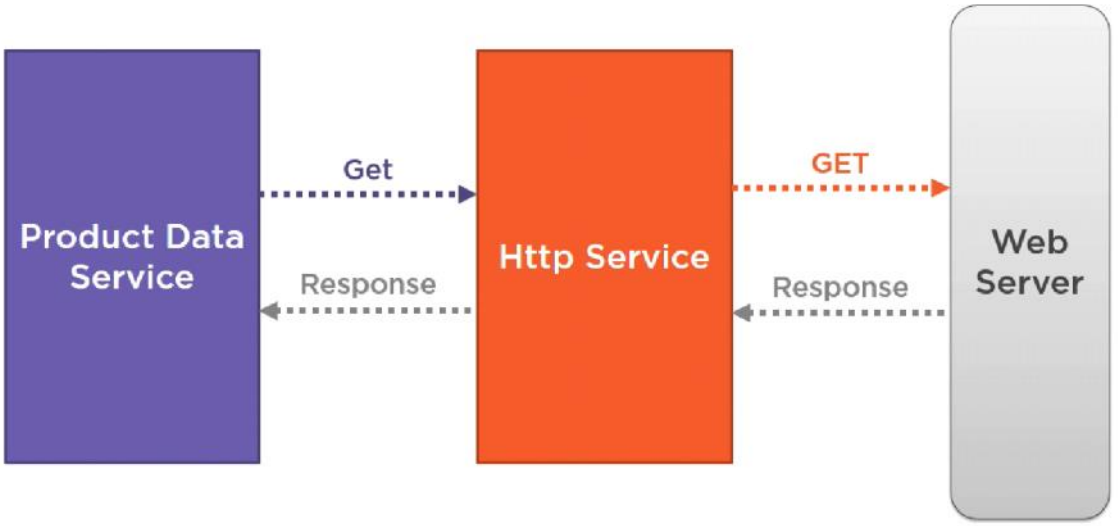
- Observables are different from promises.
  Lazy in this context refers to the observable being called only when subscribed by a component to start receiving values.

Promise vs Observable

Promise	Observable
Provides a single future value	Emits multiple values over time
Not lazy	Lazy
Not cancellable	Cancellable
	Supports map, filter, reduce and similar operators

Sending an http request

Sending an Http Request



Sending an Http Request

product.service.ts

```
...
import { HttpClient } from '@angular/common/http';

@Injectable({
   providedIn: 'root'
})
export class ProductService {
  private productUrl = 'www.myWebService.com/api/products';

  constructor(private http: HttpClient) { }

  getProducts() {
   return this.http.get(this.productUrl);

  }
}
```

Registering the Http Service Provider

app.module.ts

```
...
import { HttpClientModule } from '@angular/common/http';

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

Sending an Http Request

product.service.ts

```
...
import { HttpClient } from '@angular/common/http';


@Injectable({
   providedIn: 'root'
})
export class ProductService {
  private productUrl = 'www.myWebService.com/api/products';

  constructor(private http: HttpClient) { }

  getProducts() {
   return this.http.get<IProduct[]>(this.productUrl);

  }
}
```

- The above get method with IProduct[] will help us map the response object with our IProduct interface type.
- We used the same kind of syntax to define the event payload when passing event information from a nested component to its con tainer.
- In the above getProducts() method does not have any return type mentioned, since we are using strong typing we should have a function return value.
- But as getting the IProduct Array as response and returning it will not suffice, the method should return an observable of IP roduct[] array as the HTTP calls are asynchronous operations. Also that the HTTP calls are single asynchronous operations, which means that the observab le sequence returned from the get method contains only 1 element. This element is the HTTP response object mapped to the type specified in the gen eric parameter.
- Observables also take advantage of generics to define the type of data it is observing in the observable sequence, which in o ur case is the array of products(IProduct[])

Sending an Http Request

product.service.ts

```
...
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
   providedIn: 'root'
})
export class ProductService {
  private productUrl = 'www.myWebService.com/api/products';

  constructor(private http: HttpClient) { }

  getProducts(): Observable<IProduct[]> {
   return this.http.get<IProduct[]>(this.productUrl);

  }
}
```

Exception handling in http requests

Exception Handling

product.service.ts

```
...
import { HttpClient, HttpErrorResponse } from '@angular/common/http';
import { Observable } from 'rxjs';
import { catchError, tap } from 'rxjs/operators';
...

  getProducts(): Observable<IProduct[]> {
   return this.http.get<IProduct[]>(this.productUrl).pipe(
              tap(data => console.log('All: ' + JSON.stringify(data))),
              catchError(this.handleError)
   );
  }

  private handleError(err: HttpErrorResponse) {
  }
```

- We have two key observable operators to handle exceptions
  a. Tap - Tap taps into the observable stream and allows us to look at the emitted values in the stream without transforming the stream. So tap is great to use for debugging or logging
  b. catchError - This catches any error .
- Tap and catchError can be imported from rxjs/operators

Subscribing to an Observable

- An observable doesn't start emitting values until subscribe is called.
- The subscribe method takes upto 3 arguments, each providing a handler function.
  - The first argument is often called a next function because it processes the next emitted value. Since observables handle multiple values over time, the next function is called for each value the observable emits. Ex.: x.subscribe(nextFn)
  - The second argument is an error handler function, which executes when there is an error. Ex.: x.subscribe(nextFn, errorFn)
  - The third argument is used when the user wants to completion of the observable. This handler is executed on completion.Ex.: x.subscribe(nextFn, errorFn, completeFn)
  - The subscribe function returns a subscription which could be used unsubscribe and cancel the subscription if needed. Ex.: let sub = x.subscribe(nextFn, errorFn, completeFn);

```
constructor(private productService: ProductService) {

}

onRatingClicked(message: string): void {
  this.pageTitle = 'Product List: ' + message;
}

performFilter(filterBy: string): IProduct[] {
  filterBy = filterBy.toLocaleLowerCase();
  return this.products.filter((product: IProduct) =>
    product.productName.toLocaleLowerCase().indexOf(filterBy) !== -1);
}

toggleImage(): void {
  this.showImage = !this.showImage;
}

ngOnInit(): void {
  this.productService.getProducts().subscribe(
    products => this.products = products,
    error => this.errorMessage = <any>error
  );
  this.filteredProducts = this.products;
}
}
```

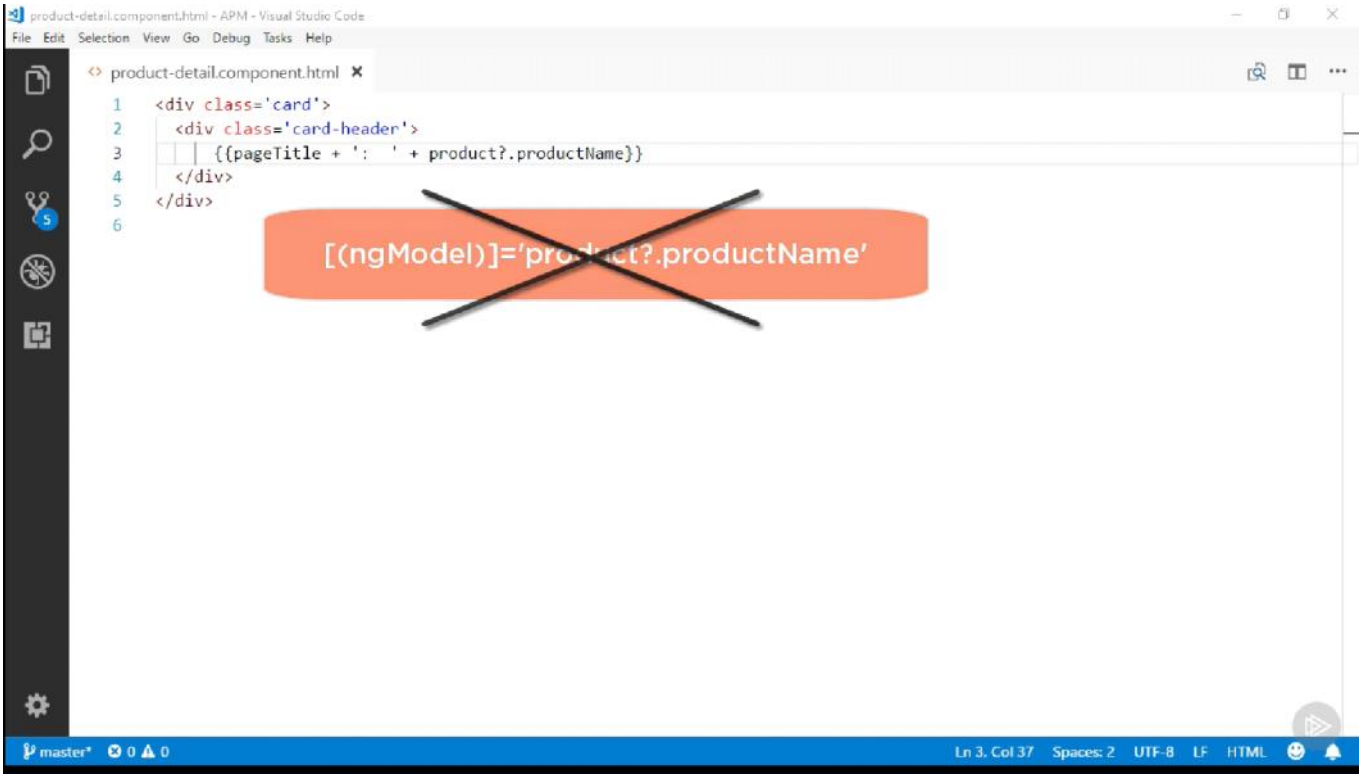
- <any>error is a casting operator which is used to type cast the error returned from the observable to the ***any*** data type.

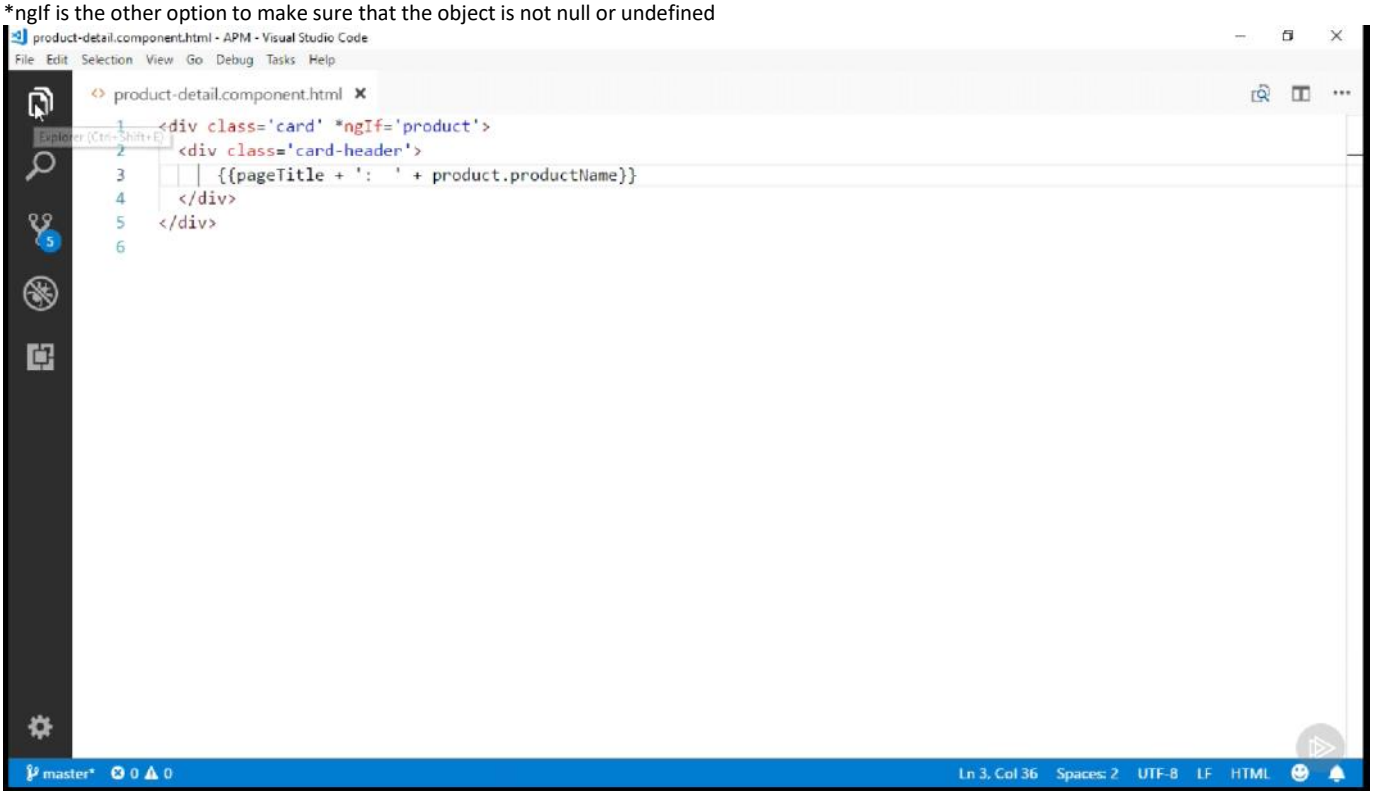
Routing

```
<div class='card'>
  <div class='card-header'>
    {{pageTitle + ':  ' + product?.productName}}
  </div>
</div>
```

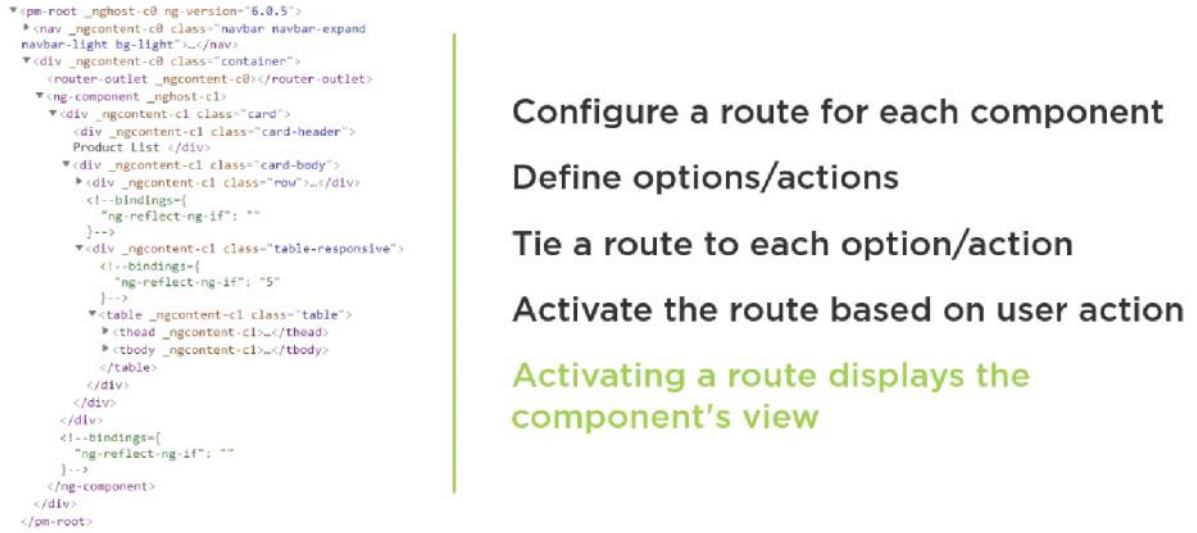
safe navigation operator

- When the object is undefined, we can handle undefined and null objects using a safe navigation operator.
- Safe navigation operator does not work when used with ngModel two-way binding as shown in the below screenshot

- 

- *ngIf is the other option to make sure that the object is not null or undefined
- 

How Routing Works

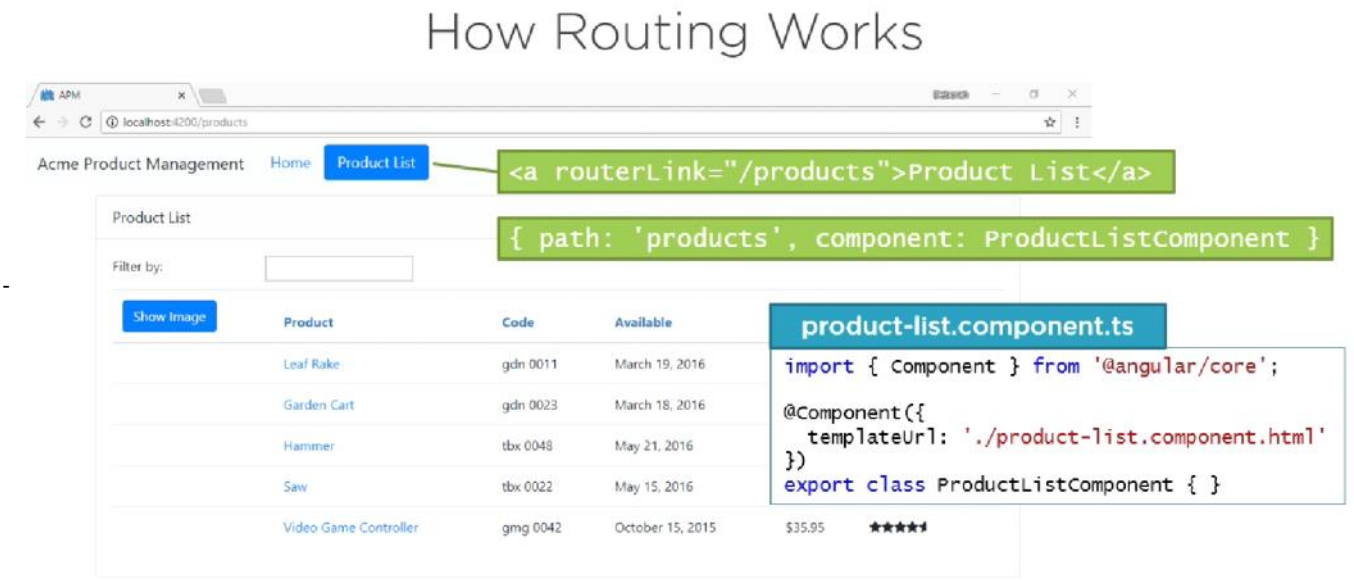
- 

- We have a built-in routing directive called "**routerLink**"

How Routing Works

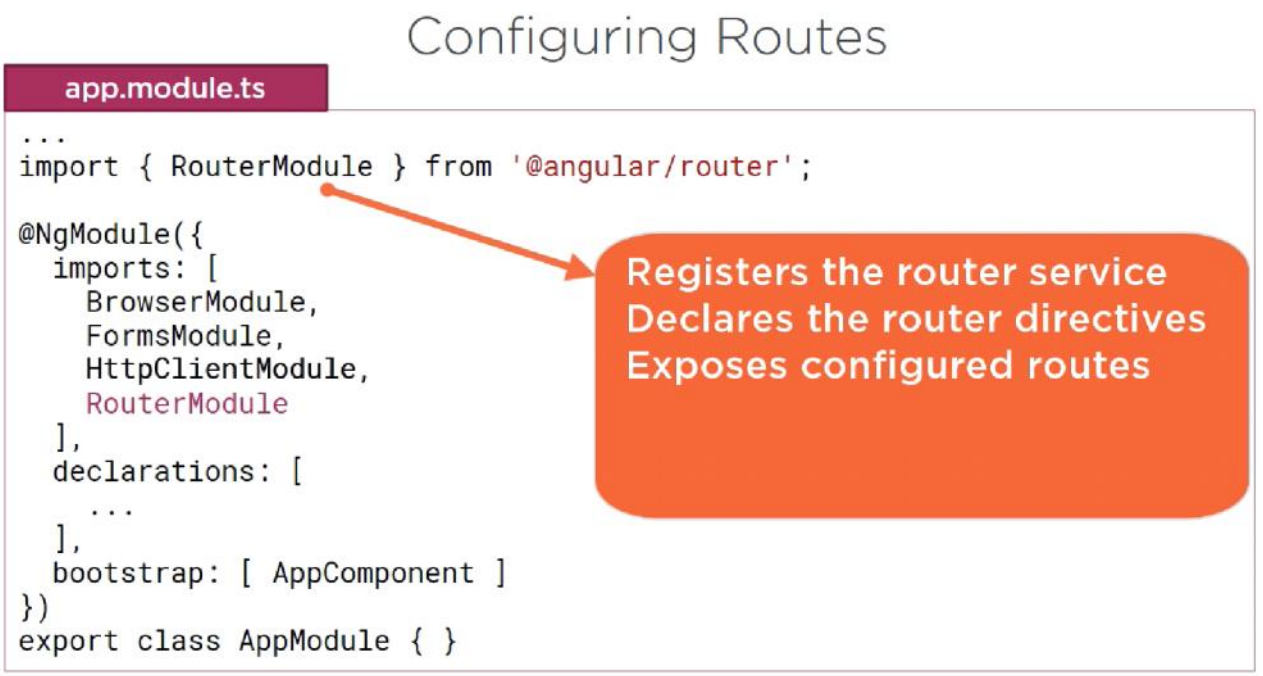
- 

- As shown in the above screenshot if the user clicks on the Product List button the angular router navigates to the product's route.
- The above also changes the path segment and we see '/products' in the address bar.
- By default the Angular uses HTML5 style urls, which don't require the hash symbol to indicate the local navigation.
- By using HTML5 style urls, you need to configure your web server to perform URL rewriting which depends on the web server on how it can be done.
- Angular also supports # style routing which does not require url rewriting.
- When the browser's url changes, the Angular router looks for a route definition matching the path segment, which is ***products*** in this example.
- The route definition includes the component to load when this route is activated. In this case it will be ***ProductListComponent***, below is the screenshot



Configuring Routes

- Angular app has one router which is managed by Angular's router service .



- Registering the router service provider in App module will declare the router directives(routerLink and router -outlet) also.
- Router module also exposes the routes we configure, before we navigate to a route we need to make sure that the routes are av ailable to the application. We do this by passing the routes to RouterModule as in the below screenshot.

Configuring Routes

app.module.ts

```
...
import { RouterModule } from '@angular/router';

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

- We call the RouterModule's forRoot method and pass our array of routes to that method. This establishes the routes for the ro ot of our application.
- If we want to use the hash style instead of HTML5 style routes we change the code to set useHash: true, as shown below

Configuring Routes

app.module.ts

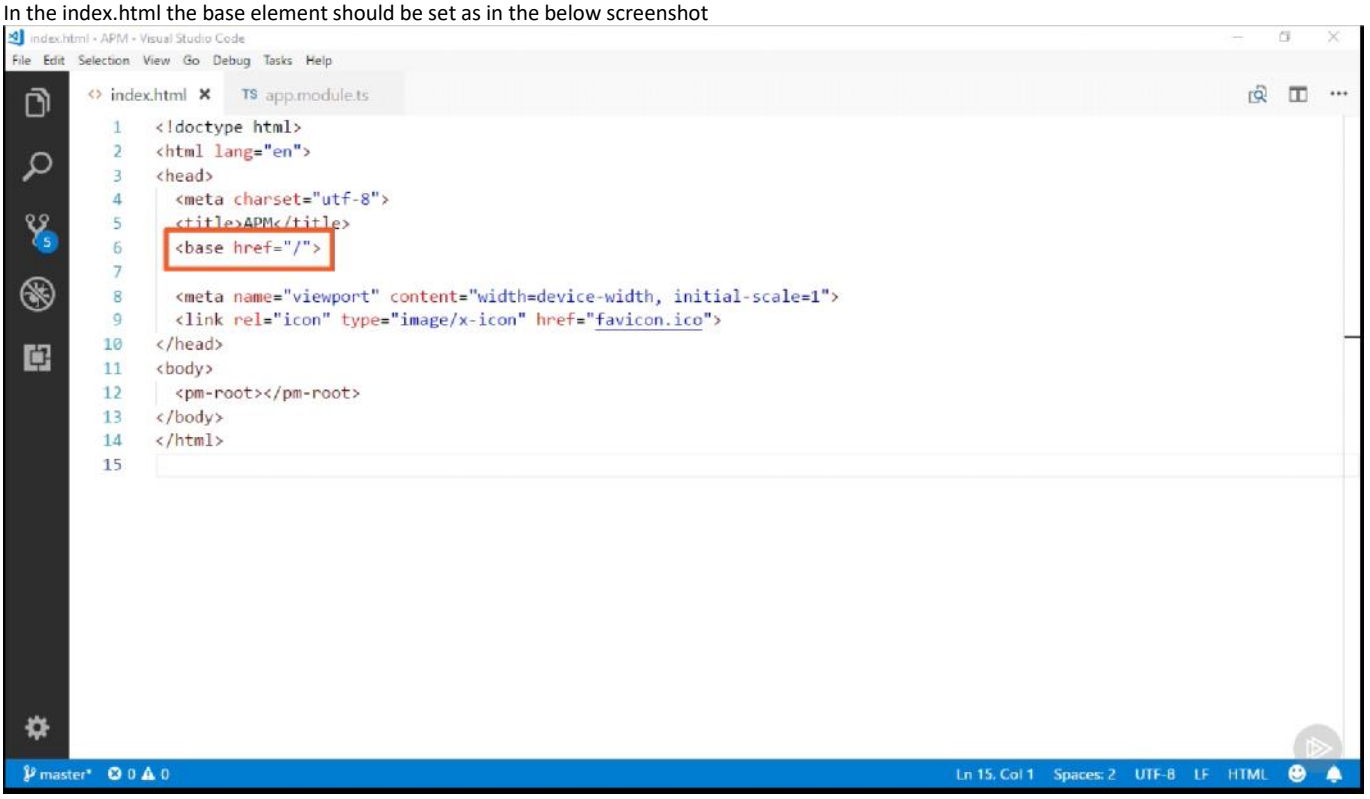
```
...
import { RouterModule } from '@angular/router';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    RouterModule.forRoot([], { useHash: true })
  ],
  declarations: [
    ...
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Configuring Routes

```
[
 { path: 'products', component: ProductListComponent },
 { path: 'products/:id', component: ProductDetailComponent },
 { path: 'welcome', component: WelcomeComponent },
 { path: '', redirectTo: 'welcome', pathMatch: 'full' },
 { path: '**', component: PageNotFoundComponent }
]
```

- Above are different examples of routes
- ***{ path: '', redirectTo: 'welcome', pathMatch: 'full' }*** // default route
- ***{ path: '**', component: 'welcome', pathMatch: 'full' }*** // default route
- A redirect route requires a pathMatch property to tell the router how to match the URL path segment to the path of a route. W e only need this default route when the entire client-side portion of the path is empty, so we set the pathMatch to 'full'.
- The asterisks in the last route denotes a wildcard path. The router matches this route if the requested url doesn't match any prior paths defined in the configuration. This is useful for displaying a 404 Not Found Page or redirecting to another route.
- Note: There will be no leading slashes in our path segments, and the order of the routes in this array matters. The router us es a first match win strategy when matching the routes. This implies that more specific routes should always be before less specific routes, such as wild c ard route.

- In the index.html the base element should be set as in the below screenshot



```
@NgModule({
  declarations: [
    AppComponent,
    ProductListComponent,
    ConvertToSpacesPipe,
    StarComponent,
    ProductDetailComponent,
    WelcomeComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    RouterModule.forRoot([
      { path: 'products', component: ProductListComponent },
      { path: 'products/:id', component: ProductDetailComponent },
      { path: 'welcome', component: WelcomeComponent },
      { path: '', redirectTo: 'welcome', pathMatch: 'full' },
      { path: '**', redirectTo: 'welcome', pathMatch: 'full' }
    ])
  ],
  bootstrap: [AppComponent]
})
```

Navigating the Application Routes

Menu option, link, image or button that activates a route

✓ Typing the Url in the address bar / bookmark

✓ The browser's forward or back buttons

Tying Routes to Actions

app.component.ts

```
...

@Component({
    selector: 'pm-root',
    template: `
     <ul class='nav navbar-nav'>
       <li><a [routerLink]="['/welcome']">Home</a></li>
       <li><a [routerLink]="['/products']">Product List</a></li>
     </ul>
    `
})
```

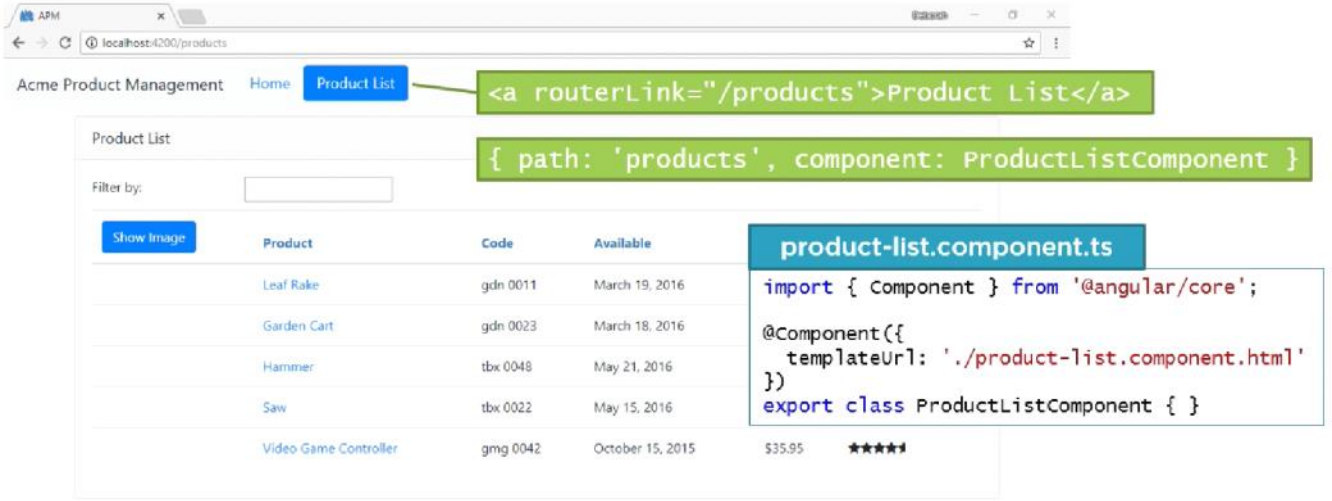
Placing the Views

app.component.ts

```
...

@Component({
    selector: 'pm-root',
    template: `
     <ul class='nav navbar-nav'>
       <li><a [routerLink]="['/welcome']">Home</a></li>
       <li><a [routerLink]="['/products']">Product List</a></li>
     </ul>
     <router-outlet></router-outlet>
    `
})
```

- <router-outlet></router-outlet> -> Router outlet is a container where we want the routed component to display it's view.
- We place the above directive in the host component's template so that the routed component's appears in that location.
- Below is the explanation of how routing works
    - When the user navigates to a particular feature tied to a route with the ***routerLink*** directive, the router link uses the linked parameters array to compose the URL segment.
    - The browser's location URL is changed to the application URL, plus the composed URL segment.
    - The router searches through the list of valid route definitions and picks the first match.
    - The router locates or creates an instance of the component associated with that route.
    - The component's view is injected in the location defined by the ***router-outlet*** directive and the page is displayed.

How Routing Works



Checklist: Displaying Components

Nest-able components

- Define a selector
- Nest in another component
- No route

Routed components

- No selector
- Configure routes
- Tie routes to actions

- We need to know when to nest a component and when to use a route.
- Nested components require a selector and requires no route but the routed component requires no selector and routes needs to be configured to display view in the router-outlet

Passing Parameters to a Route

app.module.ts

```
@NgModule({
  imports: [
    ...,
    RouterModule.forRoot([
      { path: 'products', component: ProductListComponent },
      { path: 'products/:id', component: ProductDetailComponent },
      { path: 'welcome', component: WelcomeComponent },
      { path: '', redirectTo: 'welcome', pathMatch: 'full' },
      { path: '**', redirectTo: 'welcome', pathMatch: 'full' }
    ])
  ],
  declarations: [...],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Passing Parameters to a Route

product-list.component.html

```
<td>
  <a [routerLink]="['/products', product.productId]">
     {{product.productName}}
  </a>
</td>
```

app.module.ts

```
{ path: 'products/:id', component: ProductDetailComponent }
```

Reading Parameters from a Route

product-detail.component.ts

```
import { ActivatedRoute } from '@angular/router';

    constructor(private route: ActivatedRoute) {
       console.log(this.route.snapshot.paramMap.get('id'));
    }
```

app.module.ts

```
{ path: 'products/:id', component: ProductDetailComponent }
```

- To get the parameter from the URL, we use the ***ActivatedRoute*** service provided by the router. As we need an instance of this service we define it as a dependency in the constructor as shown below

Reading Parameters from a Route

product-detail.component.ts

```
import { ActivatedRoute } from '@angular/router';

    constructor(private route: ActivatedRoute) {
       console.log(this.route.snapshot.paramMap.get('id'));
    }
```

app.module.ts

```
{ path: 'products/:id', component: ProductDetailComponent }
```

- The line defined in the constructor would create a private variable called route and assigns it to the instance of the Activa tedRoute provided by the Angular service injector.
- We use this instance of the ActivatedRoute service to get the desired parameter from the URL.
- There are two ways to get the parameter from the URL

- Use a snapshot - this is used when we need only the initial value of the parameter (Ex. User always navigates to the product list page to navigate to another product, so here the snapshot approach would be sufficient)

Reading Parameters from a Route

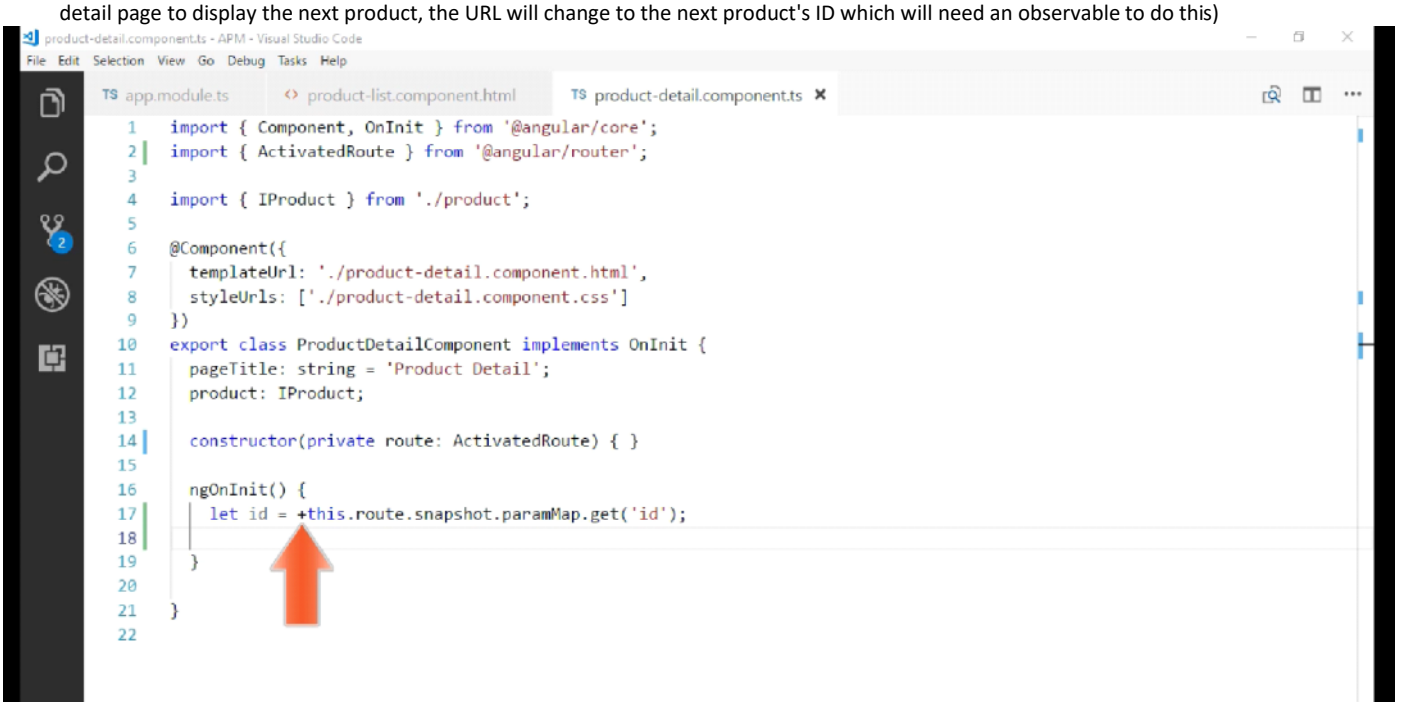
product-detail.component.ts

```
import { ActivatedRoute } from '@angular/router';

    constructor(private route: ActivatedRoute) {
       console.log(this.route.snapshot.paramMap.get('id'));
    }
```

app.module.ts

```
{ path: 'products/:id', component: ProductDetailComponent }
```

- Use an observable - This is used when we expect the parameter to change without leaving the page (Ex. If we have a next button on product detail page to display the next product, the URL will change to the next product's ID which will need an observable to do this)



- The plus(+) used in the let assignment(pointed in the above screenshot) is a Javascript shortcut to convert the parameter str ing into numeric ID.

Activating a Route with Code

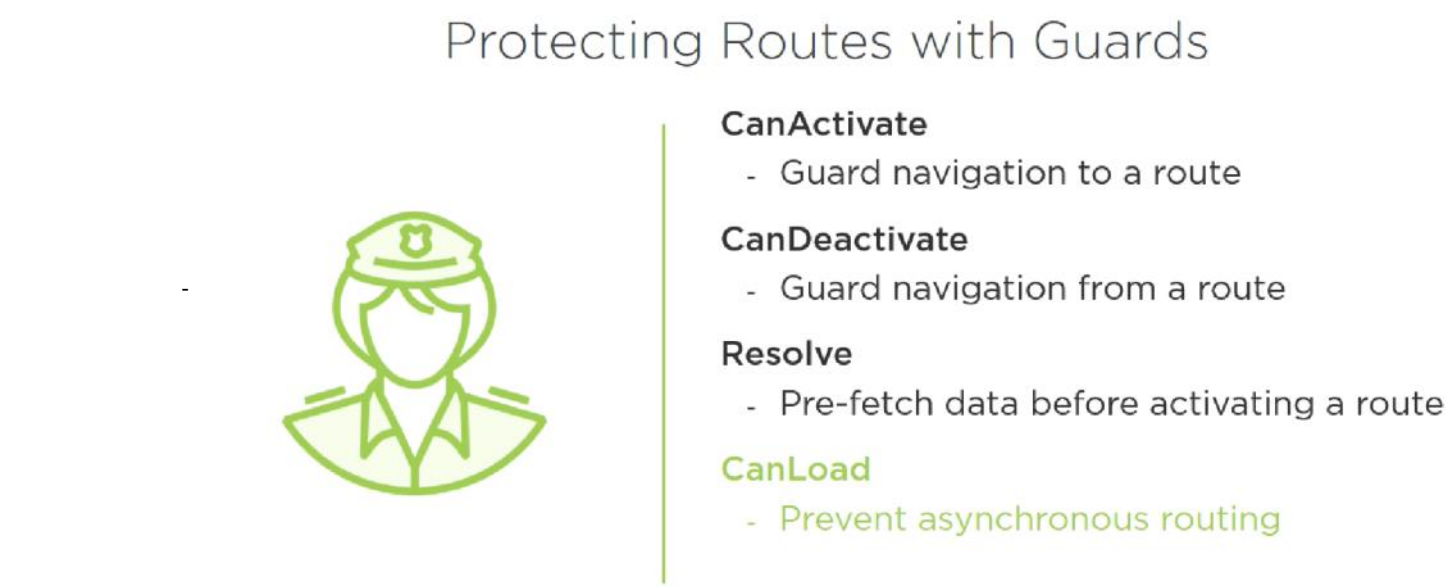
product-detail.component.ts

```
import { Router } from '@angular/router';
...
    constructor(private router: Router) { }

    onBack(): void {
      this.router.navigate(['/products']);
    }
```

- We need the Router service to activate a route through code
- We inject the router dependency though the constructor parameter, which injects the router instance into the component class.
- Each time we inject a service dependency into a class, we should check if the service is registered with the Angular Injector . In the case of router, it is registered in router module, which will be added to the app module imports array.
- We use the navigate method of the router service and add the path parameter similar to what we did when using the routerLink.

Protecting routes with guards

- When we access limit to a route, like when we need routes to be accessible only to specific users(Ex. administrators) or when we want the user to confirm a navigation operation, such as asking before whether to save before navigating away from an edit page. In these sit uations we can use guards.
- Angular router provides several guards
    - CanActivate - to guard navigation to a route
    - CanDeactivate - to guard navigation away from a current route
    - Resolve - to pre-fetch data before activating a route
    - CanLoad - to prevent asynchronous routing

- 

- Building a guard follows a common pattern throughout angular as below
    - Create a class - ProductDetailGuard which implements CanActivate to use the CanActivate guard. Inside the class we implement the canActivate method, which can return a boolean value for simple cases, true to activate the route and false to cancel the route activation. For more complex cases we return an Observable or a promise from this method.
    - Add a decorator - We use @Injectable decorator as we will be using this guard as a service
    - Import required

Building a Guard

product-detail.guard.ts

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

@Injectable({
  providedIn: 'root'
})
export class ProductDetailGuard implements CanActivate {

    canActivate(): boolean {
      ...
    }
}
```

Using a Guard

app.module.ts

```
@NgModule({
  imports: [
    ...,
    RouterModule.forRoot([
      { path: 'products', component: ProductListComponent },
      { path: 'products/:id',
        canActivate: [ProductDetailGuard ],
        component: ProductDetailComponent },
      ...])
  ],
  declarations: [...],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

- To use the guard, we add the guard to a product detail route. We add canActivate and set it to an array containing the guards to execute before this route is activated.
- To generate a guard through CLI we use the following command
  - ***ng g g products/product-detail*** - > 1st g is for generating, 2nd g is guard followed by the path and name, which creates the below
  - 

- The Angular CLI registers the generated guard with the root application injector, using the ***providedIn*** property of the ***@Injectable*** decorator
- By default the CLI generates code for ***CanActivate*** guard. It implements the ***CanActivate*** interface and builds the start of the ***canActivate*** method. We can change this as required if we want to implement a different type of guard.
- The ***canActivate*** method has two parameters as mentioned below
  - ***ActivatedRouteSnapshot*** - to provide current route information
  - ***RouterStateSnapshot*** - to provide router state information, this method can return an observable, promise or a boolean value.
- Here, in our example we use this Router guard to check the route URL to ensure if the id passed is not null and greater than 0, using the canActivate method. If id is invalid we will be navigating back to the ProductsList page.
- As navigation requires the router, so we use a constructor we inject the router service.
- In ***canActivate*** method, we have to read the parameter from the route which we can do it using the "***ActivatedRouteSnapshot***" which is provide as a parameter in the ***canActivate*** method, the ***ActivatedRouteSnapshot*** contains information about a route at any particular moment in time.

app.module.ts | product-detail.guard.ts

products/10

0 1

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

@Injectable({
  providedIn: 'root'
})
export class ProductDetailGuard implements CanActivate {

  constructor(private router: Router) { }

  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    let id = +next.url[1].path;
    if (isNaN(id) || id < 1) {
      alert("Invalid product Id");
      this.router.navigate(['/products']);
      return false;
    };
    return true;
  }
}
```

app.module.ts - APM - Visual Studio Code

File Edit Selection View Go Debug Tasks Help

app.module.ts | product-detail.guard.ts

```ts
@NgModule({
  declarations: [
    AppComponent,
    ProductListComponent,
    ConvertToSpacesPipe,
    StarComponent,
    ProductDetailComponent,
    WelcomeComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    RouterModule.forRoot([
      { path: 'products', component: ProductListComponent },
      { path: 'products/:id',
        canActivate: [ ProductDetailGuard ],
        component: ProductDetailComponent },
      { path: 'welcome', component: WelcomeComponent },
      { path: '', redirectTo: 'welcome', pathMatch: 'full' },
      { path: '**', redirectTo: 'welcome', pathMatch: 'full' }
    ])
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

master* 0 0 Ln 31, Col 45 Spaces: 2 UTF-8 CRLF TypeScript 2.9.1