What are the differences between template and templateUrl properties and when to use one over the other

> Angular2 recommends to extract templates into a separate file, if the view template is longer than 3 lines.
>
> Let's understand why is it better to extract a view template into a seprate file, if it is longer than 3 lines.

With an inline template

1. We loose Visual Studio editor intellisense, code-completion and formatting features.
2. TypeScript code is not easier to read and understand when it is mixed with the inline template HTML.

With an external view template

We have Visual Studio editor intellisense, code-completion and formatting features and

Not only the code in "app.component.ts" is clean, it is also easier to read and understand

Class binding in angular 2

Suggested Videos
Part 9 - Property binding in Angular 2 | Text | Slides
Part 10 - html attribute vs dom property | Text | Slides
Part 11 - Angular attribute binding | Text | Slides

In this video we will discuss **CSS Class binding in Angular** with examples.

For the demos in this video, we will use same example we have been working with so far in this video series. In **styles.css** file include the following 3 CSS classes. If you recollect styles.css is already referenced in our host page - index.html.

```
.boldClass{
    font-weight:bold;
}

.italicsClass{
    font-style:italic;
}

.colorClass{
    color:red;
}
```

In **app.component.ts**, include a button element as shown below. Notice we have set the class attribute of the button element to 'colorClass'.

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

@Component({
    selector: 'my-app',
    template: `
                <button class='colorClass'>My Button</button>
            `
})
export class AppComponent {
}
```

At this point, run the application and notice that the 'colorClass' is added to the button element as expected.

Replace all the existing css classes with one or more classes

Modify the code in app.component.ts as shown below.

1. We have introduced a property 'classesToApply' in **AppComponent** class
2. We have also specified class binding for the button element. The word 'class' is in a pair of square brackets and it is binded to the property 'classesToApply'

3. This will replace the existing css classes of the button with classes specified in the class binding

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

@Component({
    selector: 'my-app',
    template: `
                <button class='colorClass' [class]='classesToApply'>My Button</button>
              `
})
export class AppComponent {
    classesToApply: string = 'italicsClass boldClass';
}
```

Run the application and notice 'colorClass' is removed and these classes (italicsClass & boldClass) are added.

Adding or removing a single class : To add or remove a single class, include the prefix 'class' in a pair of square brackets, followed by a DOT and then the name of the class that you want to add or remove. The following example adds boldClass to the button element. Notice it does not remove the existing colorClass already added using the class attribute. If you change applyBoldClass property to false or remove the property altogether from the AppComponent class, css class boldClass is not added to the button element.

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

@Component({
    selector: 'my-app',
    template: `
                <button class='colorClass' [class.boldClass]='applyBoldClass'>My
Button</button>
              `
})
export class AppComponent {
    applyBoldClass: boolean = true;
}
```

With class binding we can also use ! symbol. Notice in the example below applyBoldClass is set to false. Since we have used ! in the class binding the class is added as expected.

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

@Component({
    selector: 'my-app',
    template: `
                <button class='colorClass' [class.boldClass]='!applyBoldClass'>My
Button</button>
              `
})
export class AppComponent {
    applyBoldClass: boolean = false;
}
```

You can also removed an existing class that is already applied. Consider the following example. Notice we have 3 classes (colorClass, boldClass & italicsClass) added to the button element using the class attribute. The class binding removes the boldClass.

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

@Component({
    selector: 'my-app',
    template: `
                <button class='colorClass boldClass italicsClass'
                    [class.boldClass]='applyBoldClass'>My Button</button>
              `
})
export class AppComponent {
    applyBoldClass: boolean = false;
}
```

To add or remove multiple classes use ngClass directive as shown in the example below.

1. Notice the colorClass is added using the class attribute
2. ngClass is binded to addClasses() method of the AppComponent class
3. addClasses() method returns an object with 2 key/value pairs. The key is a CSS class name. The value can be true or false. True to add the class and false to remove the class.
4. Since both the keys (boldClass & italicsClass) are set to true, both classes will be added to the button element
5. let is a new type of variable declaration in JavaScript.
6. let is similar to var in some respects but allows us to avoid some of the common gotchas that we run into when using var.
7. The differences between let and var are beyond the scope of this video. For our example, var also works fine.
8. As TypeScript is a superset of JavaScript, it supports let

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

@Component({
    selector: 'my-app',
    template: `
                <button class='colorClass' [ngClass]='addClasses()'>My Button</button>
              `
})
export class AppComponent {
    applyBoldClass: boolean = true;
    applyItalicsClass: boolean = true;

    addClasses() {
        let classes = {
            boldClass: this.applyBoldClass,
            italicsClass: this.applyItalicsClass
        };

        return classes;
```

```
    }
}
```

We have included our css classes in a external stylesheet - **styles.css**. Please note we can also include these classes in the styles property instead of a separate stylesheet as shown below.

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

@Component({
    selector: 'my-app',
    template: `
                <button class='colorClass' [ngClass]='addClasses()'>My Button</button>
              `,
    styles: [`
            .boldClass{
                font-weight:bold;
            }

            .italicsClass{
                font-style:italic;
            }

            .colorClass{
                color:red;
            }
             `]
})
export class AppComponent {
    applyBoldClass: boolean = true;
    applyItalicsClass: boolean = true;

    addClasses() {
        let classes = {
            boldClass: this.applyBoldClass,
            italicsClass: this.applyItalicsClass
        };

        return classes;
    }
}
```

Angular ngFor trackBy

Using trackyBy with ngFor directive :

ngFor directive may perform poorly with large lists

A small change to the list like, adding a new item or removing an existing item may trigger a cascade of DOM manipulations

if we used track by,it doesnot re render of element only added or removed element will be added or removed.

Pipe

```
<td>{{employee.annualSalary | currency:'USD':true:'1.3-3'}}</td>
```

1. The first parameter is the currencyCode
2. The second parameter is boolean - True to display currency symbol, false to display currency code
3. The third parameter ('1.3-3') specifies the number of integer and fractional digits

Output :

Annual Salary
$5,500.000
$5,700.950
$5,900.000
$6,500.826

Create Custome PIPe

- Import pipe decorator and pipe transporm interface
- Register pipe in angular module where we need it.
- include in the declareation array.

Here is what we want to do. Depending on the gender of the employee, we want to display Mr. or Miss. prefixed to the employee name as shown below.

Code	Name	Gender	Annual Salary	Date of Birth
emp101	Mr.Tom	Male	5500	25/6/1988
emp102	Mr.Alex	Male	5700.95	9/6/1982
emp103	Mr.Mike	Male	5900	12/8/1979
emp104	Miss.Mary	Female	6500.826	14/10/1980
emp105	Miss.Nancy	Female	6700.826	15/12/1982

Step 1 : To achieve this let's create a custom pipe called employeeTitlePipe. Right click on the "employee" folder and add a new TypeScript file. Name it "employeeTitle.pipe.ts". Copy and paste the following code.

Code Explanation :

- Import Pipe decorator and PipeTransform interface from Angular core
- Notice "EmployeeTitlePipe" class is decorated with Pipe decorator to make it an Angular pipe
- name property of the pipe decorator is set to employeeTitle. This name can then be used on any HTML page where you want this pipe functionality.
- EmployeeTitlePipe class implements the PipeTransform interface. This interface has one method transform() which needs to be implemented.
- Notice the transform method has 2 parameters. value parameter will receive the name of the employee and gender parameter receives the gender of the employee. The method returns a string i.e Mr. or Miss. prefixed to the name of the employee depending on their gender.

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
    name: 'employeeTitle'
})
export class EmployeeTitlePipe implements PipeTransform {
    transform(value: string, gender: string): string {
        if (gender.toLowerCase() == "male")
            return "Mr." + value;
        else
            return "Miss." + value;
    }
}
```

Step 2 : Register "EmployeeTitlePipe" in the angular module where we need it. In our case we need it in the root module. So in app.module.ts file, import the EmployeeTitlePipe and include it in the "declarations" array of NgModule decorator

```
import { EmployeeTitlePipe } from './employee/employeeTitle.pipe'

@NgModule({
    imports: [BrowserModule],
    declarations: [AppComponent, EmployeeComponent,
                   EmployeeListComponent, EmployeeTitlePipe],
```

```
    bootstrap: [AppComponent]
})

export class AppModule { }
```

Step 3 : In "employeeList.component.html" use the "EmployeeTitlePipe" as shown below. Notice we are passing employee gender as an argument for the gender parameter of our custom pipe. Employee name gets passed automatically.

```
<tr *ngFor='let employee of employees;'>
    <td>{{employee.code}}</td>
    <td>{{employee.name | employeeTitle:employee.gender}}</td>
    <td>{{employee.gender}}</td>
    <td>{{employee.annualSalary}}</td>
    <td>{{employee.dateOfBirth}}</td>
</tr>
```

Why do we need a service in Angular

A service in Angular is generally used when you need to reuse data or logic across multiple components.

Anytime you see logic or data-access duplicated across multiple components, think about refactoring that piece of logic or data-access code into a service.

Using a service ensures we are not violating one of the Software principles - DRY ((Don't repeat yourself).

The logic or data access is implemented once in a service, and the service can be used across all the components in our application.

Difference between constructor and ngOnInit

A class constructor is automatically called when an instance of the class is created.

It is generally used to initialise the fields of the class and it's sub classes.

ngOnInit is a life cycle hook method provided by Angular.

ngOnInit is called after the constructor and is generally used to perform tasks related to Angular bindings.

For example, ngOnInit is the right place to call a service method to fetch data from a remote server.

We can also do the same using a class constructor, but the general rule of thumb is, tasks that are time consuming should use ngOnInit instead of the constructor.

As fetching data from a remote server is time consuming, the better place for calling the service method is ngOnInit.

single instance of the service which enables us to share data and functionality across multiple components in our application.

So in summary DI provides these benefits

Create applications that are easy to write and maintain over time as the application evolves

Easy to share data and functionality as the angular injector provides a Singleton i.e a single instance of the service

Easy to write and maintain unit tests as the dependencies can be mocked

Step 1 : First register the service with the angular injector

Step 2 : Specify a dependency using the constructor of your component class.

With these 2 steps in place, the angular **injector** will automatically inject an instance of the the service into the component's constructor when an instance of the component is created.

To register a service with the root injector we use providers property of @ngModule decorator and to register a service with the injector at a component level use providers property of @Component decorator.

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

import { UserPreferencesService } from './employee/userPreferences.service';

@NgModule({

   providers: [UserPreferencesService]

})

export class TestModule { }
```

Angular router navigation events

To see these navigation events in action, set **enableTracing** option to true as shown below.

```
RouterModule.forRoot(appRoutes, { enableTracing:
true
})
```

Enabling tracing logs all the router navigation events to the browser console.

What are the use cases of these navigation events

1. Monitor routes
2. Troubleshoot when routing does not work as expected
3. Display a loading message if there is a delay when navigating from one route to another (We will discuss this in our next video)

Angular resolve guard

You don't want to display a partial page to the user, while waiting for the data. Before we activate the LIST route and display the view template associated with the LIST route, we want to pre-fetch the data.

Once the data is available, that's when we want to render the view template, so the end user does not see a partial page.

So in short, we want to delay rendering the routed component view template until all necessary data have been fetched.

Validation

<input id="email" required [email]="employee.email!==""

Add email attribute when mail not empty

Note To remember

<input type="radio" required #contactPreference="ngModel" name="contactPreference"

value="email" [(ngModel)]="employee.contactPreference">

we ca get value using reference variable

contactPreference.value=>email

angular value vs ngvalue

<option [ngValue]="null">Select Department</option>.
Notice we are using ngValue instead of value.

If you use value, null is treated as a string and not as a null. Hence the required validation does not work.

Along with using ngValue, also make sure you set the department property on the employee model object to null.

Select List Custom validator

To make this custom validator reusable,
we want to be able to do pass the default option value from the template to our custom validator as shown below.

Notice we are using our custom validator selector and passing it the default option value.
In this case we are passing -101. If you have another SELECT list, and if it's default option value is -1, you simply pass that value.

<select appSelectValidator="-101" #department="ngModel" ....>

This show the custome validator with hard corded default value

```
import { Validator, AbstractControl, NG_VALIDATORS } from '@angular/forms';
import { Directive } from '@angular/core';

@Directive({
   selector: '[appSelectValidator]',
   providers: [
      {
         provide: NG_VALIDATORS,
         useExisting: SelectRequiredValidatorDirective,
         multi: true
      }]
})
export class SelectRequiredValidatorDirective implements Validator {
   validate(control: AbstractControl): { [key: string]: any } | null {
      return control.value === '-1' ? { 'defaultSelected': true } : null;
   }
}
```

```
<div class="form-group"
    [class.has-error]="department.touched && department.errors?.defaultSelected">
  <label for="department" class="control-label">Department</label>
  <select id="department" #department="ngModel" name="department"
          [(ngModel)]="employee.department" appSelectValidator
          class="form-control">
    <option value="-1">Select Department</option>
    <option *ngFor="let dept of departments" [value]="dept.id">
     {{dept.name}}
    </option>
  </select>
  <span class="help-block"
       *ngIf="department.touched && department.errors?.defaultSelected">
    Department is required
  </span>
</div>
```

```
import { Validator, AbstractControl, NG_VALIDATORS } from '@angular/forms';
// Import input from @angular/core package
import { Directive, Input } from '@angular/core';

@Directive({
    selector: '[appSelectValidator]',
    providers: [{
        provide: NG_VALIDATORS,
        useExisting: SelectRequiredValidatorDirective,
        multi: true
    }]
})
export class SelectRequiredValidatorDirective implements Validator {
    // Create input property to receive the
    // specified default option value
    @Input() appSelectValidator: string;
    validate(control: AbstractControl): { [key: string]: any } | null {
        // Remove the hard-coded value and use the input property instead
        return control.value === this.appSelectValidator ?
                                { 'defaultSelected': true } : null;
    }
}
```

```
<select appSelectValidator="-101" #department="ngModel" ....>
```

Password and confirm password validation

trigger validation manually if not trigger validation automatically

```
<input name="password"
    (change)="confirmPassword.control.updateValueAndValidity()" ...>
```

Group Validation

```
<div ngModelGroup="passwordGroup" #passwordGroup="ngModelGroup"
    appConfirmEqualValidator [class.has-error]="passwordGroup.errors?.notEqual
    && !confirmPassword.errors?.required">

  <div class="form-group"
      [class.has-error]="password.touched && password.invalid">
    <label for="password" class="control-label">Password</label>
    <input name="password" required type="text" class="form-control"
        [(ngModel)]="employee.password" #password="ngModel">
    <span class="help-block"
        *ngIf="password.touched && password.errors?.required">
      Password is required
    </span>
  </div>

  <div class="form-group"
      [class.has-error]="confirmPassword.touched && confirmPassword.invalid">
    <label for="confirmPassword" class="control-label">Confirm Password</label>
    <input name="confirmPassword" required type="text" class="form-control"
        [(ngModel)]="employee.confirmPassword" #confirmPassword="ngModel">
    <span class="help-block"
        *ngIf="confirmPassword.touched && confirmPassword.errors?.required">
      Confirm Password is required
    </span>
    <span class="help-block" *ngIf="confirmPassword.touched &&
        passwordGroup.errors?.notEqual && !confirmPassword.errors?.required">
      Password and Confirm Password does not match
    </span>
  </div>

</div>
```

Custom Validator Code :

```
import { Validator, NG_VALIDATORS, AbstractControl } from '@angular/forms';
import { Directive } from '@angular/core';

@Directive({
    selector: '[appConfirmEqualValidator]',
    providers: [{
        provide: NG_VALIDATORS,
        useExisting: ConfirmEqualValidatorDirective,
        multi: true
    }]
})
export class ConfirmEqualValidatorDirective implements Validator {
    validate(passwordGroup: AbstractControl): { [key: string]: any } | null {
        const passwordField = passwordGroup.get('password');
        const confirmPasswordField = passwordGroup.get('confirmPassword');
        if (passwordField && confirmPasswordField &&
            passwordField.value !== confirmPasswordField.value) {
            return { 'notEqual': true };
        }

        return null;
    }
}
```

```
<h1 #h1Variable></h1>

<div *ngFor="let employee of employees">
  <div (click)="h1Variable.innerHTML = childComponent.getNameAndGender()">
    <app-employee-display [employee]="employee" #childComponent>
    </app-employee-display>
  </div>
</div>
```

Code Explanation :

#childComponent is the template reference variable to the child component. Using this template variable we can call child component public property (employee) and method (getNameAndGender())

<div (click)="handleClick(childComponent.getNameAndGender())">. Using the template reference variable we are calling the child component method

getNameAndGender(). The value this method returns is assigned to the innerHTML property of the <h1> element. #h1Variable is the template reference variable for <h1> element.

At this point when you click on an employee panel, you will see that employee's name and gender displayed by the <h1> element.

Calling the child component property using template reference variable :
we are calling the child component public property employee using the template reference variable childComponent.

There are 2 ways to pass data from Child Component to Parent Component

1. Output Properties
2. Template Reference Variable

Reactive Forms

```
// When any of the form control value in employee form changes
// our validation function logValidationErrors() is called
this.employeeForm.valueChanges.subscribe((data) => {
  this.logValidationErrors(this.employeeForm);
});
```

```
<div class="form-group" [ngClass]="{'has-error': formErrors.email}">
  <label class="col-sm-2 control-label" for="email">Email</label>
```

```
id="{{'skillName'+i}}"
```

- Notice the id attribute of the skillName textbox. We are dynamically computing it's ID, by appening **i** variable value to the string skillName

```
id="{{'skillName'+i}}"
```

- This will generate an ID of skillName0 for the first skillName input element, ID of skillName1 for the second skillName input element, so on so forth, to ensure unique ID values are assigned to all the dynamically generated skillName input elements.
- In the above expression we are using interpolation. We could also achieve the same using property binding syntax instead of interpolation.

```
[id]="'skillName'+i"
```

- If you are new to property binding, please check out our following video on **property binding in Angular**

```
https://www.youtube.com/watch?v=RGYfTx9AAQA
```

- Also notice, we are dynamically setting the value of the for attribute of the Skill label.

```
attr.for="{{'skillName'+i}}"
```

- Since the for attribute does not have a corresponding DOM property, we are using Angular's attribute binding.
- If you are new to attribute binding, please check out our following video on **Attribute binding in Angular**

```
https://www.youtube.com/watch?v=OZJiQ5kj9us
```

- With the attribute binding, we are using interpolation. We could also achieve the same using property binding syntax.

```
[attr.for]="'skillName'+i"
```

Cross Field Validation

```
// validationMessages object.
ngOnInit() {
  this.employeeForm = this.fb.group({
    fullName: ['', [Validators.required, Validators.minLength(2),
Validators.maxLength(10)]],
    contactPreference: ['email'],
    emailGroup: this.fb.group({
      email: ['', [Validators.required, emailDomain('dell.com')]],
      confirmEmail: ['', [Validators.required]],
    }, { validator: matchEmails }),
    phone: [''],
    skills: this.fb.group({
      skillName: ['', Validators.required],
      experienceInYears: ['', Validators.required],
      proficiency: ['', Validators.required]
    }),
  });

  this.employeeForm.valueChanges.subscribe((data) => {
    this.logValidationErrors(this.employeeForm);
  });

  this.employeeForm.get('contactPreference').valueChanges.subscribe((data: string)
=> {
    this.onContactPrefernceChange(data);
  });
}
```

```
// Nested form group (emailGroup) is passed as a parameter. Retrieve email and
// confirmEmail form controls. If the values are equal return null to indicate
// validation passed otherwise an object with emailMismatch key. Please note we
// used this same key in the validationMessages object against emailGroup
// property to store the corresponding validation error message
function matchEmails(group: AbstractControl): { [key: string]: any } | null {
  const emailControl = group.get('email');
  const confirmEmailControl = group.get('confirmEmail');

  if (emailControl.value === confirmEmailControl.value || confirmEmailControl.pristine)
{
    return null;
  } else {
    return { 'emailMismatch': true };
  }
}
```

Configuring Preloading in Angular :

Step 1 : Import PreloadAllModules type from @angular/router package

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

Step 2 : Set preloadingStrategy to PreloadAllModules.

We do this in the configuration object that we pass as a second parameter to the forRoot() method of the RouterModule class.

```
@NgModule({
  imports: [
    RouterModule.forRoot(appRoutes, { preloadingStrategy: PreloadAllModules })
  ],
  exports: [ RouterModule ]
})
export class AppRoutingModule { }
```

The value for preloadingStrategy property can be one of the following

Value	Description
NoPreloading	This is the default and does not preload any modules
PreloadAllModules	Preloads all modules as quickly as possible in the background
Custom Preload Strategy	We can also specify our own custom preloading strategy. We will discuss why and how to implement custom preloading strategy in our next video.

The main benefit of Lazy loading is, it can significantly reduce the initial application load time.
With lazy loading configured, our application does not load every module on startup.

Only the root module and any other essential modules that the user expects to see when the application first starts are loaded.

However there is a downside for lazy loading.

When a route in a lazy loaded module is first requested, the user has to wait for that module to be downloaded.
We do not have this problem with eager loading, because all the modules are already downloaded,
so the end user gets to see the component associated with the route without any wait time.

PreLoading Strategy

preloadingStrategy options, we can either preload, all lazy loaded modules or none of them.

Let us say in our Angular project, we have 2 lazy loaded modules

- EmployeeModule
- AdminModule

If we set, preloadingStrategy property to PreloadAllModules, then both the lazy loaded modules will be preloaded in the background.

```
RouterModule.forRoot(appRoutes, {preloadingStrategy: PreloadAllModules})
```

On the other hand, if we set preloadingStrategy property to NoPreloading, then none of the lazy loaded modules will be preloaded.

```
RouterModule.forRoot(appRoutes, {preloadingStrategy: NoPreloading})
```

So with the following 2 built-in preloadingStrategy options, we can either preload, all lazy loaded modules or none of them.

- PreloadAllModules
- NoPreloading

But, what if we want to preload some of the modules and not the other. In our case, let's say we want to preload EmployeeModule, but not the AdminModule. This is when, we create our own **Custom Preloading Strategy**.

1. PreloadAllModules
2. NoPreloading

Steps for creating Custom Preloading Strategy in Angular

Step 1 : To create a Custom Preloading Strategy, create a service that implements Angular's built-in PreloadingStrategy abstract class

To generate the service, use the following Angular CLI command

```
ng g s CustomPreloading
```

This command generates a file with name **custom-preloading.service.ts**. Modify the code in that file as shown below.

```
import { Injectable } from '@angular/core';
import { PreloadingStrategy, Route } from '@angular/router';
import { Observable, of } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
// Since we are creating a Custom Preloading Strategy, this service
// class must implement PreloadingStrategy abstract class
export class CustomPreloadingService implements PreloadingStrategy {

  constructor() { }

  // PreloadingStrategy abstract class has the following preload()
  // abstract method for which we need to provide implementation
  preload(route: Route, fn: () => Observable<any>): Observable<any> {
    // If data property exists on the route of the lazy loaded module
    // and if that data property also has preload property set to
    // true, then return the fn() which preloads the module
    if (route.data && route.data['preload']) {
      return fn();
    // If data property does not exist or preload property is set to
    // false, then return Observable of null, so the module is not
    // preloaded in the background
    } else {
      return of(null);
    }
  }
}
```

Step 2 : Import CustomPreloadingService and set it as the Preloading Strategy

In **app-routing.module.ts**, import CustomPreloadingService

In **app-routing.module.ts**, import CustomPreloadingService

```
import { CustomPreloadingService } from './custom-preloading.service';
```

Set CustomPreloadingService as the Preloading Strategy

```
RouterModule.forRoot(appRoutes, {
  preloadingStrategy: CustomPreloadingService
})
```

Modify the **'employees'** route, and set preload property to true or false. Set it to true if you want the EmployeeModule to be preloaded else false.

```
const appRoutes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  {
    path: 'employees',
    // set the preload property to true, using the route data property
    // If you do not want the module to be preloaded set it to false
    data: { preload: true },
    loadChildren: './employee/employee.module#EmployeeModule'
  },
  { path: '**', component: PageNotFoundComponent }
];
```

Creating Angular Service

```
import { Injectable } from '@angular/core';
import { IEmployee } from './IEmployee';
import { HttpClient, HttpErrorResponse, HttpHeaders } from '@angular/common/http';

import { Observable, throwError } from 'rxjs';
import { catchError } from 'rxjs/operators';

@Injectable()
```