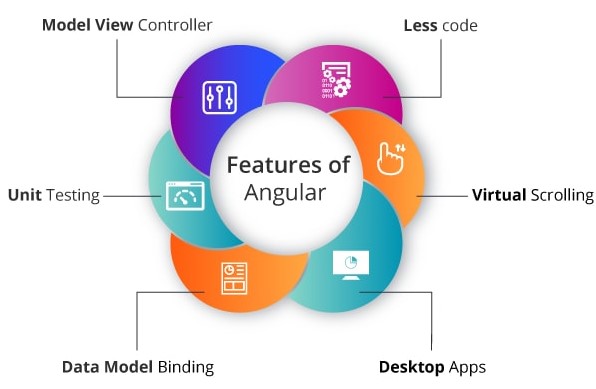
# **What is Angular?**

The name Angular derives simply from the fact that the HTML tags are enclosed by angle brackets.

Originally called AngularJS, Angular is Google’s JavaScript (TypeScript-based) open-source front-end web application framework. It is designed specifically for creating dynamic web applications. With this framework, you can develop front-end based applications without having to use other plug-ins or frameworks.

Angular is very similar to the [JavaScript](https://www.besanttechnologies.com/training-courses/web-designing-training/javascript-training-institute-in-chennai) framework as it is open-source. As there is great support from the **Angular CLI and Angular App**e Google team and many ideas are being imported, Angular is said to be up to date. All the latest trends available in today’s market are incorporated with this wonderful framework.



#### Less code:

Plenty of scripts is necessary just to design an application when performing DOM manipulation. But the very minimal amount of code is enough for DOM manipulation if using Angular.

#### Model View Controller:

The angular framework is constructed on a well-known idea called Model-View-Controller (MVC). I hope you all are aware that MVC is a design pattern that is used in the entire web applications in today’s modern trend.

#### Unit Testing:

We have an excellent test framework known as Karma designed by the Google development team. This is very much helpful for performing designing unit tests for AngularJS applications.

#### Data Model Binding:

While binding data to the HTML controls, special code is not required. Simply by adding very few snippets of code is possible to bind data, which is done through Angular.

#### Desktop Apps:

Using Angular, you can easily create applications that are desktop-installed across various operating systems such as Windows, Mac, and Linux.

#### Virtual Scrolling:

To load and unload items from DOM, Virtual Scrolling in Angular is principally used. This process is completely based on visible parts of lists.

# **What is TypeScript ?**

TypeScript is a new language developed by Microsoft that extends JavaScript. It is a superset of JavaScript ES2015 and incorporates the new version of JavaScript functionality. Using TypeScript, you can write state-of-the-art JavaScript without using Babel. TypeScript also has a powerful typing system that allows static analysis of code through type annotations and type interfaces.

Because TypeScript is heavily influenced by Java and the .NET framework (C # and VB), TypeScript is easier to learn than pure JavaScript if developers are experienced with these languages. Among the major frameworks, Angular was the first to have actively adopted TypeScript.

# **What is ES6 or ECMAScript ?**

ES5 is an abbreviation of **ECMAScript 5** and also known as **ECMAScript** 2009. The sixth edition of the **ECMAScript** standard is **ES6** or **ECMAScript 6**. It is also known as ECMAScript 2015. [ES6](https://www.javatpoint.com/es6) is a major enhancement in the [JavaScript](https://www.javatpoint.com/javascript-tutorial) language that allows us to write programs for complex applications.

# **Difference between ES5 and ES6**

|  |  |  |
| --- | --- | --- |
| **Based on** | **ES5** | **ES6** |
| **Definition** | ES5 is the fifth edition of the ECMAScript (a trademarked scripting language specification defined by ECMA International) | ES6 is the sixth edition of the ECMAScript (a trademarked scripting language specification defined by ECMA International). |
| **Release** | It was introduced in 2009. | It was introduced in 2015. |
| **Data-types** | ES5 supports primitive data types that are **string, number, boolean, null,** and **undefined**. | In ES6, there are some additions to JavaScript data types. It introduced a new primitive data type **'symbol'** for supporting unique values. |
| **Defining Variables** | In ES5, we could only define the variables by using the **var** keyword. | In ES6, there are two new ways to define variables that are **let** and **const**. |
| **Performance** | As ES5 is prior to ES6, there is a non-presence of some features, so it has a lower performance than ES6. | Because of new features and the shorthand storage implementation ES6 has a higher performance than ES5. |
| **Support** | A wide range of communities supports it. | It also has a lot of community support, but it is lesser than ES5. |
| **Object Manipulation** | ES5 is time-consuming than ES6. | Due to destructuring and speed operators, object manipulation can be processed more smoothly in ES6. |
| **Arrow Functions** | In ES5, both **function** and **return** keywords are used to define a function. | An arrow function is a new feature introduced in ES6 by which we don't require the **function** keyword to define the function. |
| **Loops** | In ES5, there is a use of **for** loop to iterate over elements. | ES6 introduced the concept of **for...of** loop to perform an iteration over the values of the iterable objects. |

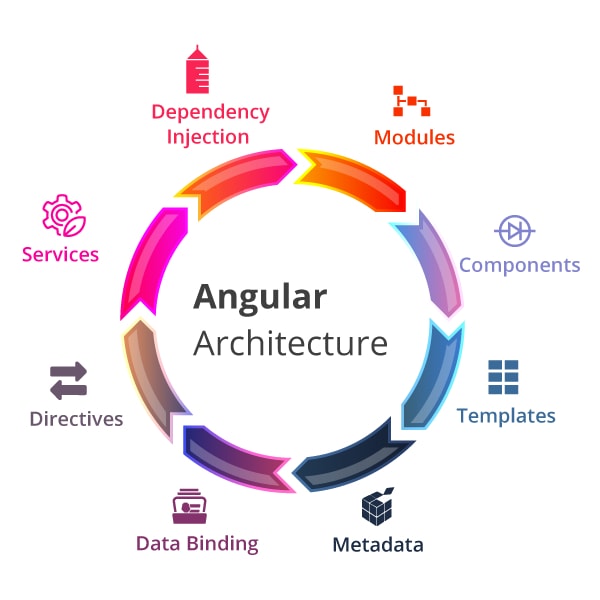
# **What is diff between ES6 ,TypeScript, Javascript ?**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **TypeScript** | **ES6** | |
| **Definition** | TypeScript is a free and open-source pure object-oriented programming language. It is developed and maintained by Microsoft. | ES6 is a version of ECMAScript (ES), which is a scripting language specification standardized by ECMA international. |
| **Explanation** | Typescript is to eradicate the development errors. | ES6 is comparatively more flexible in development time. |
| **Data-Types** | TypeScript supports all primitive data types. | ES6 does not support all data types. |
| **Features** | TypeScript contains features such as generics and type annotations, Inference, Enums, and Interfaces. | ES6 does not support these features. |
| **Scope** | Typescript has three scopes.   1. Global Scope 2. Class Scope 3. Local Scope | ES6 has two scopes.   1. Global Scope 2. Local Scope |
| **Decision-Making** | 1. if Statement 2. if-else Statement 3. else...if and nested if statements 4. switch Statement | 1. if Statement 2. if-else Statement 3. The else- if ladder/nested if statements. 4. switch?case Statement |
| **Modules** | TypeScript Modules are of two types:   1. Internal 2. External modules | We can classify the ES6 modules in two ways:   1. Importing a module 2. Exporting a module |
| **Loop** | Typescript and ES6 both are having same loops.   1. Definite 2. Indefinite | Typescript and ES6 both are having same loops.   1. Definite 2. Indefinite |
| **Why choose** | The developers choose TypeScript:   * Typesafe * JavaScript superset * Powerful type system, including generics & JS features. * Aligned with ES development for compatibility. * Structural, rather than nominal, subtyping. * Compile-time errors. * Starts and ends with JavaScript. | The developers choose ES6:   * ES6 code is shorter than traditional JS * Module System Standardized * Extremely compact * Destructuring Assignment |
| **Company using** | The list of companies which uses TypeScript are:   * Slack * Asana * CircleCI * Intuit * Swat.io * Avocode | The list of companies which uses ES6 are:   * Slack * StackShare * eBay * Asana * Intuit * Swat.io |

# **What are the differences between Angular and AngularJS**

| **Parameters** | **AngularJS** | **Angular** |
| --- | --- | --- |
| Script | Based on JavaScript | Based on TypeScript |
| To use a property or event | you must remember the correct ng directive | uses () for event binding and [] for property binding . |
| Mobile features | Not focused on this | Supports mobile features |
| Syntax for routing | Uses $routeProvider.when()to configure routing | Uses @RouteConfig{(...}). |
| Performance | Drawing speed of the view is moderate | Drawing speed of the view is 3 to 5 times faster than Angular JS. Template pre-compilation and view caching are used to reduce memory usage and CPU load. |
| Template engine | Has many directives and every developer can also specify new custom directive. | Angular also has standard directives, but they are used in a bit different way. E.g.: ng-model in AngularJS means that you want to create two-way binding. If you want to create one-way binding, you should use ng-bind. |
| Features that affect supporting by various browsers | Have controllers | Controllers are replaced by components. |
| One-way data binding | ng-bind is used to link data in one direction (one-way data binding) | Replaced by one-way data binding.[property] |
| Two-way data binding links | ng-model | replaced by [(ngModel)] |

# **How does an Angular application work?**



Every Angular app consists of a file named angular.json. This file will contain all the configurations of the app.

While building the app, the builder looks at this file to find the entry point of the application.

Following is an image of the angular.json file:

"build": {

"builder": "@angular-devkit/build-angular:browser",

"options": {

"outputPath": "dist/angular-starter",

"index": "src/index.html",

"main": "src/main.ts",

"polyfills": "src/polyfills.ts",

"tsConfig": "tsconfig.app.json",

"aot": false,

"assets": [

"src/favicon.ico",

"src/assets"

],

"styles": [

"./node\_modules/@angular/material/prebuilt-themes/deeppurple-amber.css",

"src/style.css"

]

}

}

Inside the build section, the main property of the options object defines the entry point of the application which in this case is **main.ts**.  
The main.ts file creates a browser environment for the application to run, and, along with this, it also calls a function called **bootstrapModule**, which bootstraps the application. These two steps are performed in the following order inside the main.ts file:

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

platformBrowserDynamic().bootstrapModule(AppModule)

In the above line of code, **AppModule** is getting bootstrapped.  
The AppModule is declared in the app.module.ts file. This module contains declarations of all the components.  
Below is an example of app.module.ts file:

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

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

import { AppComponent } from './app.component';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule

],

providers: [],

entryComponents: [],

bootstrap: [AppComponent]

})

export class AppModule { }

As one can see in the above file, **AppComponent** is getting bootstrapped.  
This component is defined in **app.component.ts** file. This file interacts with the webpage and serves data to it.  
Below is an example of app.component.ts file:

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

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

title = 'angular';

}

Each component is declared with three properties:  
1. **Selector** - used for accessing the component  
2. **Template/TemplateURL** - contains HTML of the component  
3. **StylesURL** - contains component-specific stylesheets  
  
After this, Angular calls the **index.html** file. This file consequently calls the root component that is **app-root**. The root component is defined in **app.component.ts**.  
This is how the index.html file looks:

**<!doctype html>**

**<html lang="en">**

**<head>**

**<meta charset="utf-8">**

**<title>Angular</title>**

**<base href="/">**

**<meta name="viewport" content="width=device-width, initial-scale=1">**

**</head>**

**<body>**

**<app-root></app-root>**

**</body>**

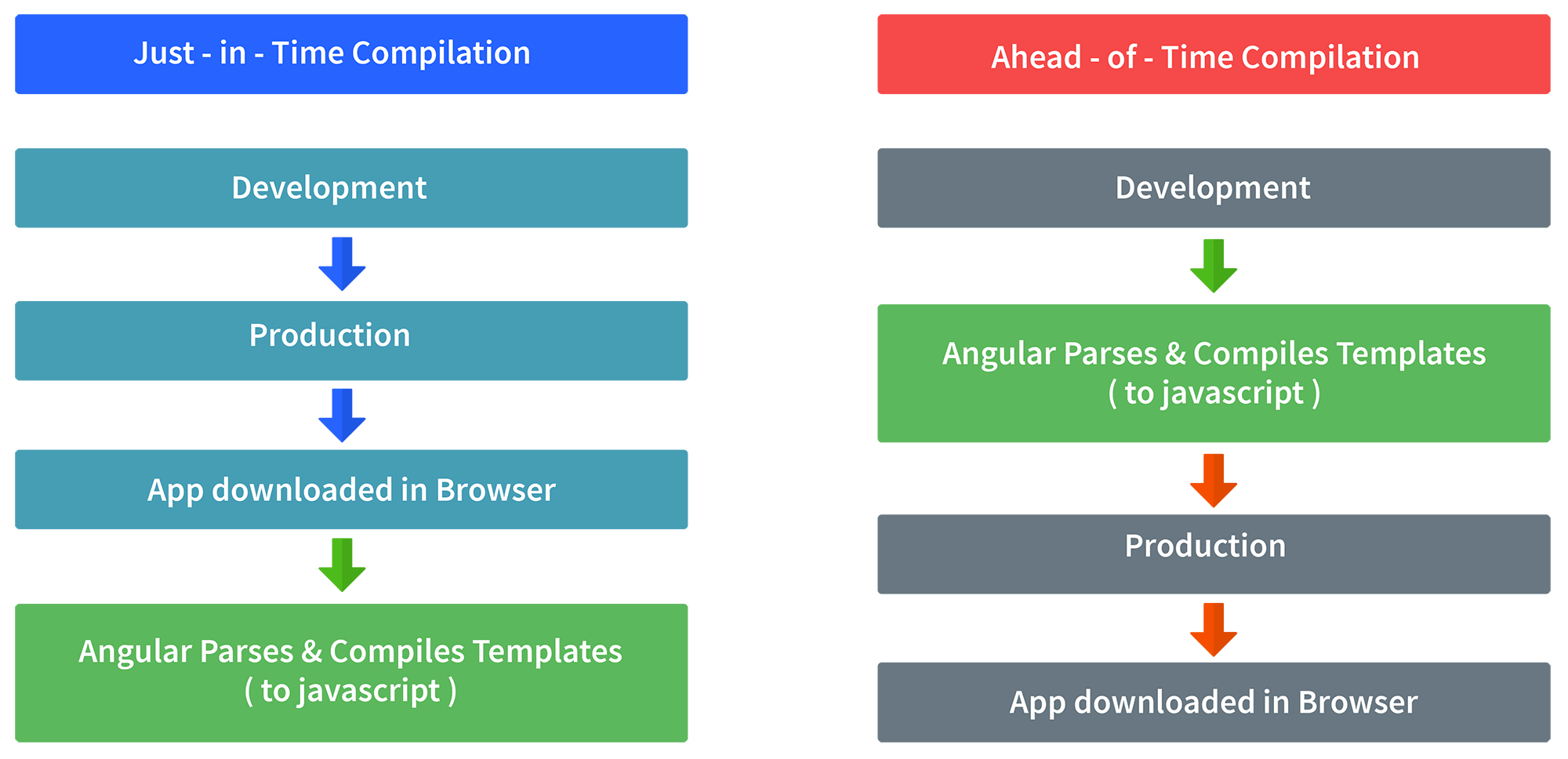
**</html>**

The HTML template of the root component is displayed inside the <app-root> tags.  
This is how every angular application works.

# **What is AOT compilation? What are the advantages of AOT?**

Every Angular application consists of components and templates which the browser cannot understand. Therefore, all the Angular applications need to be compiled first before running inside the browser.  
Angular provides two types of compilation:

* JIT(Just-in-Time) compilation
* AOT(Ahead-of-Time) compilation



In JIT compilation, the application compiles inside the browser during runtime.  
Whereas in the AOT compilation, the application compiles during the build time.  
  
The advantages of using AOT compilation are:

 Since the application compiles before running inside the browser, the browser loads the executable code and renders the application immediately, which leads to **faster rendering**.

 In AOT compilation, the compiler sends the external HTML and CSS files along with the application, eliminating separate AJAX requests for those source files, which leads to **fewer ajax requests**.

 Developers can detect and handle errors during the building phase, which helps in **minimizing errors**.

 The AOT compiler adds HTML and templates into the JS files before they run inside the browser. Due to this, there are no extra HTML files to be read, which provide **better security** to the application.

By default, angular builds and serves the application using JIT compiler:

ng build  
ng serve

For using AOT compiler following changes should be made:

ng build --aot  
ng serve –aot

# **Explain Components, Modules and Services in Angular ?**

create an Angular application by running the following inside the command terminal:

For better understanding, I would like you to create an Angular application by running the following inside the command terminal:

ng new angularApp

The above command will create an angular application in the directory.  
Next, let's move on to understand Components, Modules, and Services.  
  
**Components**  
In Angular, components are the basic building blocks, which control a part of the UI for any application.  
A component is defined using the **@Component** decorator. Every component consists of three parts, the template which loads the view for the component, a stylesheet which defines the look and feel for the component, and a class that contains the business logic for the component.  
For creating a component, inside the command terminal, navigate to the directory of the application created, and run the following command:

ng generate component test

Or

ng g c test

One can see the generated component inside src/app/test folder. The component will be defined inside test.component.ts and this is how it looks:

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

@Component({

selector: 'app-test',

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

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

})

export lass TestComponent implements OnInit {

constructor() {}

ngOnInit() {

}

}

As we can see in the above image, our component is defined with **@Component** decorator.

**Modules**  
A module is a place where we can group components, directives, services, and pipes. Module decides whether the components, directives, etc can be used by other modules, by exporting or hiding these elements. Every module is defined with a @NgModule decorator.  
By default, modules are of two types:

 Root Module

 Feature Module

Every application can have only one root module whereas, it can have one or more feature modules.  
A root module imports **BrowserModule**, whereas a feature module imports **CommonModule**.  
In the application that we created before, one can see that the root module is defined inside **app.module.ts** and this is how it looks:

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

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

import { AppComponent } from './app.component';

import { TestComponent } from './test/text.component';

@NgModule({

declarations: [

AppComponent,

TestComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

We can see in the above image that the component we created earlier is already imported in the declarations array.  
  
To create a feature module, run the following command:

ng g m test-module

The module is created inside the src/app/test-module/test-module.module.ts file:

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

import { CommonModule } from '@angular/common';

@NgModule({

declarations: [],

imports: [

CommonModule

]

})

export class TestModuleModule { }

As one can see, **CommonModule** is imported since this is a feature module.

**Services** Services are objects which get instantiated only once during the lifetime of an application. The main objective of a service is to share data, functions with different components of an Angular application.  
A service is defined using a **@Injectable** decorator. A function defined inside a service can be invoked from any component or directive.  
  
To create a service, run the following command:

ng g s test-service

The service will be created inside src/app/test-service.service.ts:

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

@Injectable({

providedIn: 'root'

})

export class TestServiceService {

constructor() { }

}

Any method/function defined inside the TestServiceService class can be directly used inside any component by just importing the service.

**How do you provide a service?**

You must register at least one provider of any service you are going to use. A service can be provided for specific modules or components or it can be made available everywhere in your application.

Provide at root level

@Injectable({

providedIn: 'root',

})

Angular creates a single, shared instance if a service is provided at root level. This shared instance is injected into any class that asks for it. By using the @Injectable() metadata, Angular can remove the service from the compiled app if it isn’t used.

**Provide with a specific NgModule**

Registering a provider with a specific NgModule will return the same instance of a service to all components in that NgModule if they ask for it.

@NgModule({

providers: [

BackendService,

Logger

], ...

})

Provide at component level

A new instance of a service is generated for each new instance of the component if you register the provider at component level.

@Component({

selector: 'app-hero-list',

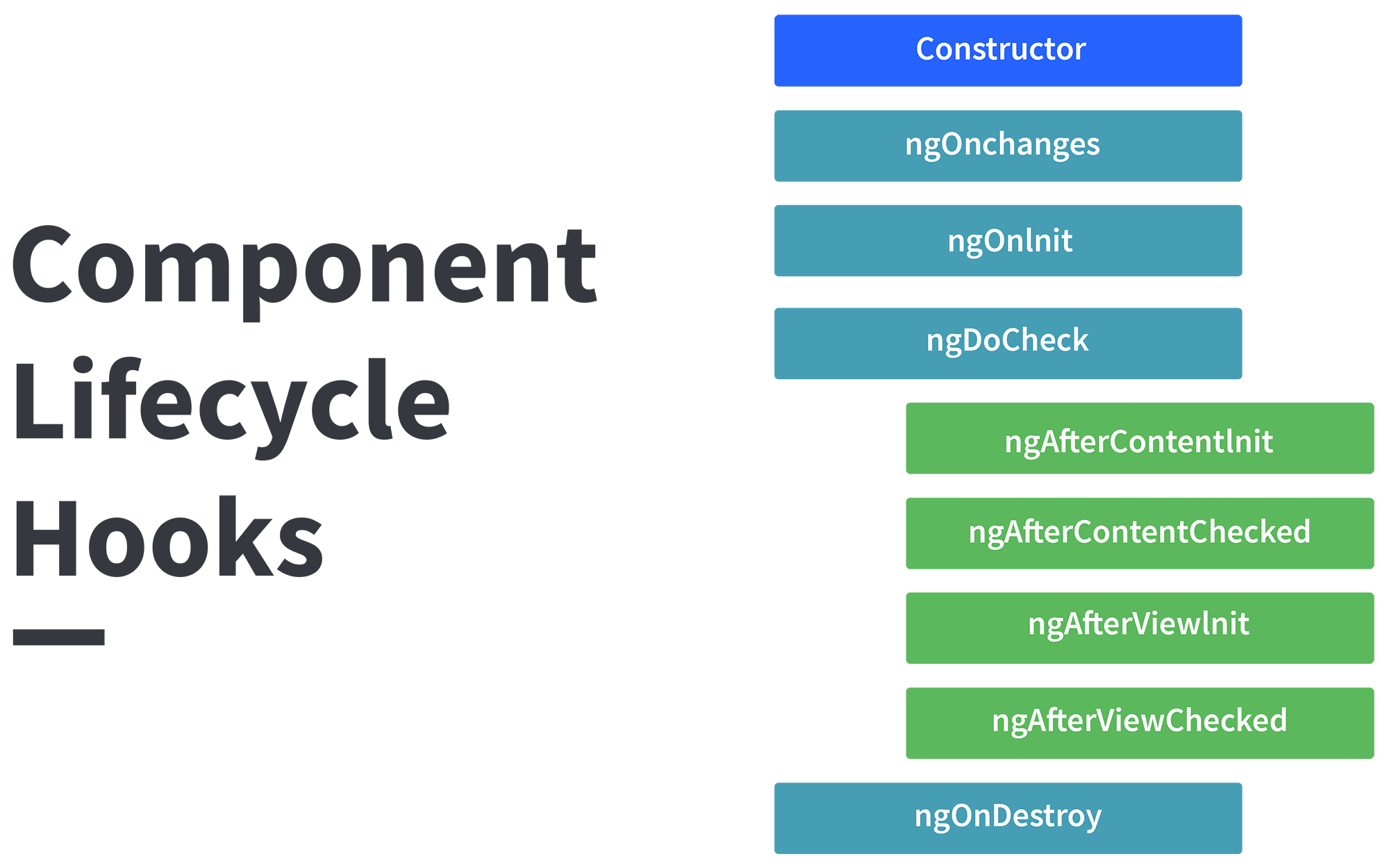
templateUrl: './hero-list.component.html',

providers: [ HeroService ]

})

# **What are lifecycle hooks in Angular? Explain a few lifecycle hooks.**

Every component in Angular has a lifecycle, different phases it goes through from the time of creation to the time it's destroyed. Angular provides **hooks** to tap into these phases and trigger changes at specific phases in a lifecycle.



**ngOnChanges( )** This hook/method is called before **ngOnInit** and whenever one or more input properties of the component changes. This method/hook receives a SimpleChanges object which contains the previous and current values of the property.  
  
**ngOnInit( )** This hook gets called once, after the **ngOnChanges** hook. It initializes the component and sets the input properties of the component.  
  
**ngDoCheck( )** It gets called after **ngOnChanges** and **ngOnInit** and is used to detect and act on changes that cannot be detected by Angular.We can implement our change detection algorithm in this hook. **ngAfterContentInit( )** It gets called after the first **ngDoCheck** hook. This hook responds after the content gets projected inside the component.  
  
**ngAfterContentChecked( )** It gets called after **ngAfterContentInit** and every subsequent **ngDoCheck**. It responds after the projected content is checked.  
  
**ngAfterViewInit( )** It responds after a component's view, or a child component's view is initialized.  
  
**ngAfterViewChecked( )** It gets called after **ngAfterViewInit**, and it responds after the component's view, or the child component's view is checked.  
  
**ngOnDestroy( )** It gets called just before Angular destroys the component. This hook can be used to clean up the code and detach event handlers.  
  
Let’s understand how to use **ngOnInit** hook, since it’s the most oftenly used hook. If one has to process lot of data during component creation, it’s better to do it inside **ngOnInit** hook rather than the constructor:

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

@Component({

selector: 'app-test',

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

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

})

export class TestComponent implements OnInit {

constructor() { }

ngOnInit() {

this.processData();

}

processData(){

// Do something..

}

}

As you can see we have imported OnInit but we have used **ngOnInit** function. This principle should be used with the rest of the hooks as well

# **Explain string interpolation and property binding in Angular?**

String interpolation and property binding are parts of **data-binding** in Angular.  
Data-binding is a feature in angular, which provides a way to communicate between the component(Model) and its view(HTML template).

Data-binding can be done in two ways, **one-way** binding and **two-way** binding.  
In Angular, data from the component can be inserted inside the HTML template. In one-way binding, any changes in the component will directly reflect inside the HTML template but, vice-versa is not possible. Whereas, it is possible in two-way binding.  
  
String interpolation and property binding allow only one-way data binding.  
String interpolation uses the double curly braces **{{ }}** to display data from the component. Angular automatically runs the expression written inside the curly braces, for example, {{ 2 + 2 }} will be evaluated by Angular and the output 4, will be displayed inside the HTML template. Using property binding, we can bind the DOM properties of an HTML element to a component's property. Property binding uses the square brackets **[ ]** syntax.

# **How are observables different from promises?**

The first difference is that an Observable is **lazy** whereas a Promise is **eager**.

|  |  |
| --- | --- |
| Promise | Observable |
| Emits a single value | Emits multiple values over a period of time |
| Not Lazy | Lazy. An observable is not called until we subscribe to the observable |
| Cannot be cancelled | Can be cancelled by using the unsubscribe() method |
|  | Observable provides operators like map, forEach, filter, reduce, retry, retryWhen etc. |

Consider the following Observable:

const observable = rxjs.Observable.create(observer => {

console.log('Text inside an observable');

observer.next('Hello world!');

observer.complete();

});

console.log('Before subscribing an Observable');

observable.subscribe((message)=> console.log(message));

When you run the above Observable, you can see messages being displayed in the following order:

Before subscribing an Observable  
Text inside an observable  
Hello world!

As you can see, observables are lazy. Observable runs only when someone subscribes to them hence, the message “Before subscribing…” is displayed ahead of the message inside the observable.  
  
Now let’s consider a Promise:

const promise = new Promise((resolve, reject) => {

console.log('Text inside promise');

resolve('Hello world!');

});

console.log('Before calling then method on Promise');

greetingPoster.then(message => console.log(message));

Running the above promise, the messages will be displayed in the following order:

Text inside promise  
Before calling then method on Promise  
Hello world!

As you can see the message inside Promise is displayed first. This means that a promise runs before the **then** method is called. Therefore, promises are **eager**.  
  
The next difference is that Promises are always **asynchronous**. Even when the promise is immediately resolved. Whereas an Observable, can be both **synchronous** and **asynchronous**.  
  
The above example of an observable is the case to show that an observable is synchronous. Let’s see the case where an observable can be asynchronous:

const observable = rxjs.Observable.create(observer => {

setTimeout(()=>{

observer.next('Hello world');

observer.complete();

},3000)

});

console.log('Before calling subscribe on an Observable');

observable.subscribe((data)=> console.log(data));

console.log('After calling subscribe on an Observable');

The messages will be displayed in the following order:

Before calling subscribe on an Observable  
After calling subscribe on an Observable  
Hello world!

You can see in this case, observable runs asynchronously.  
  
The next difference is that Observables can emit **multiple** values whereas Promises can emit only one value.  
  
The biggest feature of using observables is the use of **operators**. We can use multiple operators on an observable whereas, there is no such feature in a promise.

# **What are directives in Angular?**

A directive is a class in Angular that is declared with a @Directive decorator. Every directive has its own behavior and can be imported into various components of an application.

When to use a directive?

Consider an application, where multiple components need to have similar functionalities.

The norm thing to do is by adding this functionality individually to every component but,

this task is tedious to perform. In such a situation, one can create a directive having the

required functionality and then, import the directive to components which require this functionality.

**Types of directives**

**Component directives**

These form the main class in directives. Instead of @Directive decorator we use @Component decorator to declare these directives. These directives have a view, a stylesheet and a selector property.

**Structural directives**

These directives are generally used to manipulate DOM elements.

Every structural directive has a ‘ \* ’ sign before them.We can apply these directives to any DOM element.

In the above example, we can \*ngIf and \*ngFor directives being used.

\*ngIf is used to check a boolean value and if it’s truthy,the div element will be displayed.

\*ngFor is used to iterate over a list and display each item of the list.

**Attribute Directives :**

These directives are used to change the look and behaviour of a DOM element.

# **How to create a custom directive?**

We’re going to create an attribute directive:

In the command terminal, navigate to the directory of the angular app and type the following command to generate a directive:

ng g directive blueBackground

The following directive will be generated. Manipulate the directive to look like this:

import { Directive, ElementRef } from '@angular/core';

@Directive({

selector: '[appBlueBackground]'

})

export class BlueBackgroundDirective {

constructor(el:ElementRef) {

el.nativeElement.style.backgroundColor = "blue";

}

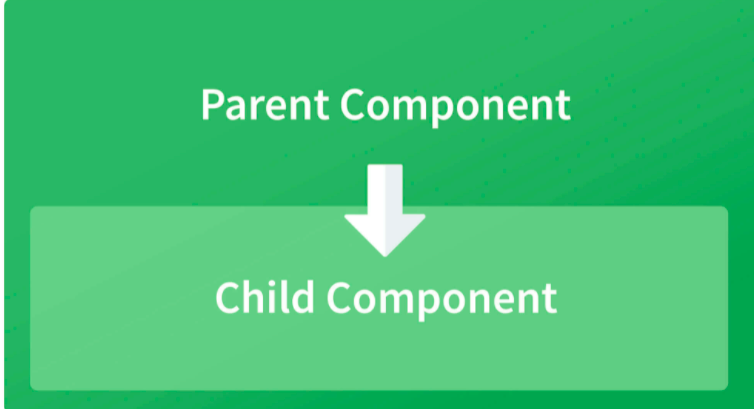
}

Now we can apply the above directive to any DOM element:

<p appBlueBackground>Hello World!</p>

# **How does one share data between components in Angular?**

Following are the commonly used methods by which one can pass data between components in angular:



**Parent to child using @Input decorator**  
  
Consider the following parent component:

@Component({

selector: 'app-parent',

template: `

<app-child [data]=data></app-child>

` ,

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

})

export class ParentComponent{

data:string = "Message from parent";

constructor() { }

}

In the above parent component, we are passing “data” property to the following child component:

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

@Component({

selector: 'app-child',

template:`

<p>{{data}}</p>

`,

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

})

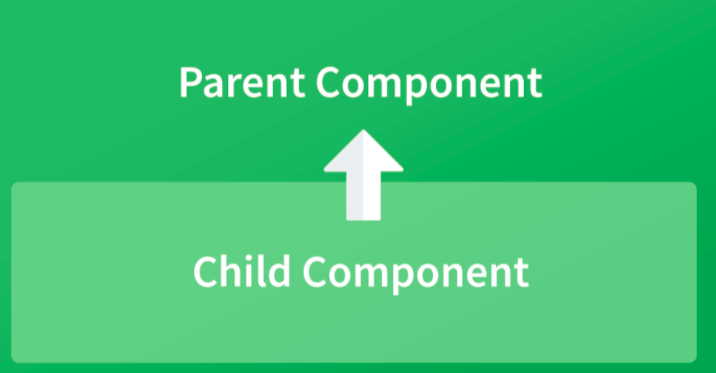
export class ChildComponent {

@Input() data:string

constructor() { }

}

In the child component, we are using @Input decorator to capture data coming from a parent component and using it inside the child component’s template.



**Child to parent using @ViewChild decorator**  
  
Child component:

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

@Component({

selector: 'app-child',

template:`

<p>{{data}}</p>

`,

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

})

export class ChildComponent {

data:string = "Message from child to parent";

constructor() { }

}

Parent Component

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

import { ChildComponent } from './../child/child.component';

@Component({

selector: 'app-parent',

template: `

<p>{{dataFromChild}}</p>

` ,

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

})

export class ParentComponent implements AfterViewInit {

dataFromChild: string;

@ViewChild(ChildComponent,{static:false}) child;

ngAfterViewInit(){

this.dataFromChild = this.child.data;

}

constructor() { }

}

In the above example, a property named “data” is passed from the child component to the parent component.  
**@ViewChild** decorator is used to reference the child component as “child” property.  
Using the **ngAfterViewInit** hook, we assign the child’s data property to the messageFromChild property and use it in the parent component’s template.  
  
**Child to parent using @Output and EventEmitter**  
  
In this method, we bind a DOM element inside the child component, to an event ( **click** event for example ) and using this event we emit data that will captured by the parent component:  
  
Child Component:

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

@Component({

selector: 'app-child',

template:`

<button (click)="emitData()">Click to emit data</button>

`,

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

})

export class ChildComponent {

data:string = "Message from child to parent";

@Output() dataEvent = new EventEmitter<string>();

constructor() { }

emitData(){

this.dataEvent.emit(this.data);

}

}

As you can see in the child component, we have used **@Output** property to bind an **EventEmitter**. This event emitter emits data when the button in the template is clicked.  
  
In the parent component’s template we can capture the emitted data like this:

<app-child (dataEvent)="receiveData($event)"></app-child>

Then inside the receiveData function we can handle the emitted data:

receiveData($event){

this.dataFromChild = $event;

}

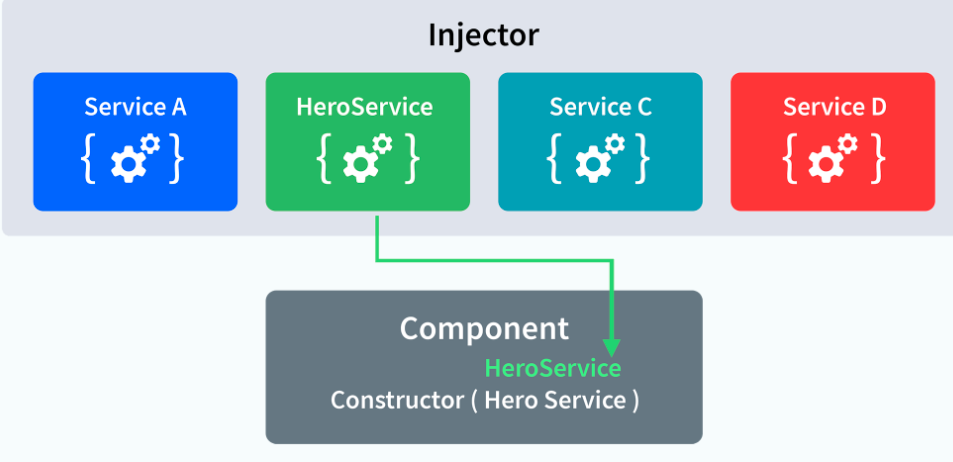
# **Explain the concept of Dependency Injection?**

Dependency injection is an application design pattern which is implemented by Angular.

It also forms one of the core concepts of Angular.

So what is dependency injection in simple terms?

Let’s break it down, dependencies in angular are nothing but services which have a functionality. Functionality of a service, can be needed by various components and directives in an application. Angular provides a smooth mechanism by which we can inject these dependencies in our components and directives.So basically, we are just making dependencies which are injectable across all components of an application.



Let’s understand how DI (Dependency Injection) works:  
  
Consider the following service, which can be generated using:

ng g service test

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

@Injectable({

providedIn: 'root'

})

export class TestService {

importantValue:number = 42;

constructor() { }

returnImportantValue(){

return this.importantValue;

}

}

As one can notice, we can create injectable dependencies by adding the **@Injectable** decorator to a class.  
  
We inject the above dependency inside the following component:

import { TestService } from './../test.service';

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

@Component({

selector: 'app-test',

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

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

})

export class TestComponent implements OnInit {

value:number;

constructor(private testService:TestService) { }

ngOnInit() {

this.value = this.testService.returnImportantValue();

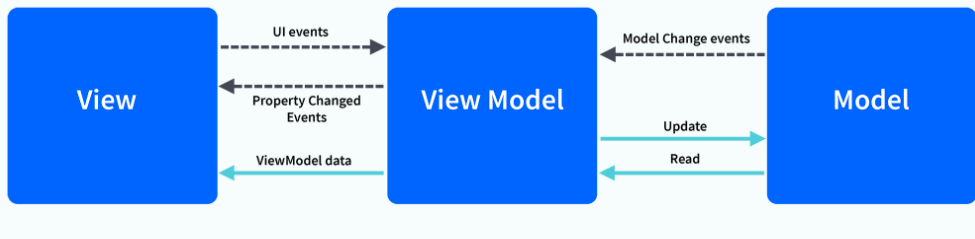
}

}

One can see we have imported our TestService at the top of the page. Then, we have created an instance inside the constructor of the component and implemented the **returnImportantValue** function of the service.

# **Explain MVVM architecture**

MVVM architecture consists of three parts:  
  
1. Model  
2. View  
3. ViewModel



Model contains the structure of an entity. In simple terms it contains data of an object.  
View is the visual layer of the application. It displays the data contained inside the Model. In angular terms, this will be the HTML template of a component.  
  
ViewModel is an abstract layer of the application. A viewmodel handles the logic of the application. It manages the data of a model and displays it in the view.  
View and ViewModel are connected with data-binding (two-way data-binding in this case). Any change in the view, the viewmodel takes a note and changes the appropriate data inside the model.

# **How to make module as lazy Loading?**

By default, NgModules are eagerly loaded, which means that as soon as the app loads, so do all the NgModules, whether or not they are immediately necessary. For large apps with lots of routes, consider lazy loading—a design pattern that loads NgModules as needed. Lazy loading helps keep initial bundle sizes smaller, which in turn helps decrease load times.

# **How to make shared module ?**

# **What is Differance between CanActivate and CanDeactivate ? And when to use them?**

Controlling Access to or from a Route:

To control whether the user can navigate to or away from a given route, use route guards.

For example, we may want some routes to only be accessible once the user has logged in or accepted Terms & Conditions. We can use route guards to check these conditions and control access to routes.

Route guards can also control whether a user can leave a certain route. For example, say the user has typed information into a form on the page, but has not submitted the form. If they were to leave the page, they would lose the information. We may want to prompt the user if the user attempts to leave the route without submitting or saving the information

canActivate and canDeactivate are guards, when your app multiple roles in application, like user, admin etc,

you use guards to protece them, also you use canActivate for pages which are accessible only for loggedin users and vice versa for canDeactivate

Registering the Route Guards with Routes:

In order to use route guards, we must register them with the specific routes we want them to run for.For example, say we have an accounts route that only users that are logged in can navigate to. This page also has forms and we want to make sure the user has submitted unsaved changes before leaving the accounts page. In our route config we can add our guards to that route:

# **What is resolver in angular ? why it used?**

# **What is Differance between resolver and CanActivate and canDeactivate ?**

there are a lot of guards in Angular that we can use to protect routes and provide the right page for the right user.

Guards are processed in the following order:

* canDeactivate
* canLoad
* canActivateChild
* canActivate
* resolve

1. **canActivate (Guard navigation to a route):**

The canActivate is called when the URL changes to the route and much the route with the Guard. This type of guards is commonly used to:

Limit route access to specific users

Ensure prerequisites are met

If you use Angular CLI, you can generate the guard you want with the following command:

'ng g g article/auth'

The first g is for generate and the second is for guard. After launching this command on the terminal, the Angular CLI will ask you which guard do you want to implement except canDeactivate and canLoad, I’ll talk about them later. After choosing the guard type, the CLI generates the guard with its testing file.

1. **canActivateChild (Guard navigation to a child route)**

This guard type is similar to the canActivate guard except that it is called when activating a route child and not the route itself. This guard job is to check the criteria before activating a child route. This type of guards is commonly used when:

Limit access to child route

Ensure prerequisites for the child route are met

1. **canDeactivate (Guard navigation away from a route):**

The canDeactivate guard’s job is to check criteria before leaving this time a route. Some common use cases for this type

of guard are:

Check for unsaved changes

Confirm leaving an incomplete operation

Alert the user

1. **canLoad (Prevent asynchronous routing):**

The canLoad guard is implemented to decide if children can be loaded or not.

You might be wondering what’s the difference between the canLoadand the canActivate?

Well, there is a difference, the canActivate exists to prevent unauthorized users from accessing a route, while canLoad is used to prevent the application from loading an entire module or component in a lazy way (lazy loading) if the user is not authorized.

So the main advantage of this type of guard is to optimize the app and have better performance.

1. **resolve (Prefetch data before activating a route):**

The last type I’m going to talk about is the resolve guard. This guard allows us data before we navigate to a route.

You may say, wait but we can retrieve data in the ngOnInit() life cycle hook, right? I agree but this approach will lead us to see an empty component at the beginning. If your client is okay with an empty component at the opening or a spinner while loading data, that’s okay but if not there is a solution for that which is the resolve guard. This allows you to render the component along with data.

# **What is Interceptor in angular ? why it used?**

Interceptors are one of the built-in tools for specifically handling HTTP requests at a global application level.

Table of contents

* [HTTP Header Interceptor](https://ultimatecourses.com/blog/intro-to-angular-http-interceptors#http-header-interceptor)
* [Formatting JSON Responses](https://ultimatecourses.com/blog/intro-to-angular-http-interceptors#formatting-json-responses)
* [Error Handling](https://ultimatecourses.com/blog/intro-to-angular-http-interceptors#error-handling)

Often we want to enforce or apply behavior when receiving or sending HTTP requests within our application. Interceptors are a unique type of Angular Service that we can implement. Interceptors allow us to intercept incoming or outgoing HTTP requests using the HttpClient. By intercepting the HTTP request, we can modify or change the value of the request.

In this post, we cover three different Interceptor implementations:

* Handling HTTP Headers
* HTTP Response Formatting
* HTTP Error Handling

This post assumes some basic knowledge of the Angular HTTP Client and RxJS Observables. Let’s take a look at the basic API implementation.

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

import { HttpInterceptor, HttpEvent, HttpResponse, HttpRequest, HttpHandler } from '@angular/common/http';

import { Observable } from 'rxjs';

@Injectable()

export class MyInterceptor implements HttpInterceptor {

intercept(httpRequest: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

return next.handle(httpRequest);

}

}

To create an Interceptor, we need to implement the HttpInterceptor interface from @angular/common/http package. Every time our application makes an HTTP request using the HttpClient service, the Interceptor calls the intercept() method.

When the intercept() method is called Angular passes a reference to the httpRequest object. With this request, we can inspect it and modify it as necessary. Once our logic is complete, we call next.handle and return the updated request onto the application.

Once our Interceptor is created, we need to register it as a multi-provider since there can be multiple interceptors running within an application. Important note, you must register the provider to the app.module for it to properly apply to all application HTTP requests. Interceptors will only intercept requests that are made using the HttpClient service.

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

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

import { HttpClientModule, HTTP\_INTERCEPTORS } from '@angular/common/http';

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

import { MyInterceptor } from './my.interceptor';

import { AppComponent } from './app.component';

@NgModule({

imports: [BrowserModule, HttpClientModule],

declarations: [AppComponent],

bootstrap: [AppComponent],

providers: [

{ provide: HTTP\_INTERCEPTORS, useClass: MyInterceptor, multi: true }

]

})

export class AppModule { }

Next, let’s take a look at our first Interceptor implementation by creating an Interceptor that can modify request headers.

HTTP Header Interceptor

Often we need to return an API key to an authenticated API endpoint via a request Header. Using Interceptors, we can simplify our application code to handle this automatically. Let’s make a simple use case of attaching an API header key to each request.

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

import { HttpInterceptor, HttpEvent, HttpResponse, HttpRequest, HttpHandler } from '@angular/common/http';

import { Observable } from 'rxjs';

import { map, filter } from 'rxjs/operators';

@Injectable()

export class HeaderInterceptor implements HttpInterceptor {

intercept(httpRequest: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

const API\_KEY = '123456';

return next.handle(httpRequest.clone({ setHeaders: { API\_KEY } }));

}

}

On the httpRequest object, we can call the clone method to modify the request object and return a new copy. In this example we are attaching the API\_KEY value as a header to every HTTP request httpRequest.clone({ setHeaders: { API\_KEY } }).

Now let’s use the HttpClient service to make a HTTP get request.

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

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

import { Observable } from 'rxjs';

import { tap } from 'rxjs/operators';

@Component({

selector: 'app-header',

template: `

<h2>Header Example</h2>

<pre>{{ data | json }}</pre>

`

})

export class HeaderComponent implements OnInit {

data: {};

constructor(private httpClient: HttpClient) { }

ngOnInit() {

this.httpClient.get('/assets/header.json').subscribe(data => this.data = data);

}

}

If we look at the dev tools in the browser, we can see the network request containing our new header API\_KEY with the corresponding value.

Request Headers

Now with each request, we automatically send our API key without having to duplicate the logic throughout our application.

Important! For security reasons ensure your Interceptor only sends your API key to the APIs that require it by checking the request URL.

Formatting JSON Responses

Often we want to modify the request value that we get back from an API. Sometimes we work with APIs that have formatted data that can make it challenging to work within our application. Using Interceptors, we can format the data and clean it up before it gets to our application logic. Let’s take a look at an example API response.

{

"id": "123",

"metadata": "blah",

"data": {

"users": {

"count": 4,

"list": [

"bob",

"john",

"doe"

]

}

}

}

In this example, our data we want to render to our component is nested deeply within the response object. The other data is just noise for our app. Using a Interceptor, we can clean up the data.

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

import { HttpInterceptor, HttpEvent, HttpResponse, HttpRequest, HttpHandler } from '@angular/common/http';

import { Observable } from 'rxjs';

import { map, filter } from 'rxjs/operators';

@Injectable()

export class FormatInterceptor implements HttpInterceptor {

intercept(httpRequest: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

return next.handle(httpRequest).pipe(

filter(event => event instanceof HttpResponse && httpRequest.url.includes('format')),

map((event: HttpResponse<any>) => event.clone({ body: event.body.data.users.list }))

);

}

}

With the httpRequest object, we can inspect the URL of the request and determine if it is a request we want to ignore or modify. If the request is to the format API endpoint, then we continue and update the response. We also only want to modify the request if the request was a response coming back from our API.

filter(event => event instanceof HttpResponse && httpRequest.url.includes('format')),

Now that we filter out only the request we care about we can update the response body to be a simple array of the users we want to display.

return next.handle(httpRequest).pipe(

filter(event => event instanceof HttpResponse && httpRequest.url.includes('format')),

map((event: HttpResponse<any>) => event.clone({ body: event.body.data.users.list }))

);

Now in our component, we can subscribe to our data without having to dig into the response object, our API returned.

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

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

import { Observable } from 'rxjs';

@Component({

selector: 'app-format',

template: `

<h2>Formated JSON</h2>

<pre>{{ data | json }}</pre>

`

})

export class FormatComponent implements OnInit {

data: {};

constructor(private httpClient: HttpClient) { }

ngOnInit() {

this.httpClient.get('/assets/format.json').subscribe(data => this.data = data);

}

}

Error Handling

We can leverage Interceptors to also handle HTTP errors. We have a few options on how to handle these HTTP errors. We could log errors through the Interceptors or show UI notifications when something has gone wrong. In this example, however, we will add logic that will retry failed API requests.

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

import { HttpInterceptor, HttpEvent, HttpResponse, HttpRequest, HttpHandler, HttpErrorResponse } from '@angular/common/http';

import { Observable } from 'rxjs';

import { retry } from 'rxjs/operators';

@Injectable()

export class RetryInterceptor implements HttpInterceptor {

intercept(httpRequest: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

return next.handle(httpRequest).pipe(retry(2));

}

}

In our request handler, we can use the RxJS retry(), operator. The retry operator allows us to retry failed Observable streams that have thrown errors. Angular’s HTTP service uses Observables which allow us to re-request our HTTP call. The retry operator takes a parameter of the number of retries we would like. In our example, we use a parameter of 2, which totals to three attempts, the first attempt plus two additional retries. If none of the requests succeed, then, the Observable throws an error to the subscriber of the HTTP request Observable.

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

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

import { Observable, of } from 'rxjs';

import { catchError } from 'rxjs/operators';

@Component({

selector: 'app-retry',

template: `<pre>{{ data | json }}</pre>`

})

export class RetryComponent implements OnInit {

data: {};

constructor(private httpClient: HttpClient) { }

ngOnInit() {

this.httpClient.get('https://example.com/404').pipe(

catchError(err => of('there was an error')) // return a Observable with a error message to display

).subscribe(data => this.data = data);

}

}

In our component, when we make a bad request, we still can catch the error using the catchError operator. This error handler will only be called after the final attempt in our Interceptor has failed.

# **How to push data on Post method REST API ?**

Whenever we see the explanation of the REST API POST method on different sites or in books, we find two major points about when to use POST methods.

1. POST Method is not idempotent.
2. We should use POST to create the resources.

But is this the right explanation of using POST methods, or have we misunderstood the contract of the POST method and now it has been so ingrained in our brains that it has started looking right?

Let's dive deeper into this topic.

## When Can We Use POST Method to Update Resources?

As the name suggests, the POST method is used to post information to the server. We do not have any control on how the server will process and save that record (PUT method also sends information to the server, but in addition to that, PUT method asks the server to put (save) that information in a particular place).

We say that POST method requests are not idempotent and have to be used only for creating resources because we do not have control on how the server is going to process that information or where it will save the information. The point here is that not having control on the server side does not mean that the POST method request has to be non-idempotent. Actually, it may or may not be idempotent.

So, we can rewrite the contract for the POST method as:

**Use the POST method when:**

* you do not have a unique identifier of the resource that you are going to send
* you are trying to send a bunch of resources in one API call, where the state of few/all resources is already persisted on the server.

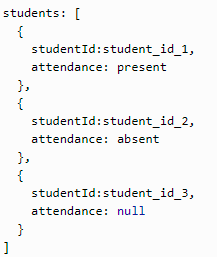
Still not clear with the concept? Let's look at an example.

## Example of Using the POST Method to Create or Update Records

There are many scenarios where we do not know whether we need to create or update the resource. In other words, it may be possible that we do not have an ID (unique location) of where we need to create/update a record.

Let's take an example of marking the attendance of students in a session. We need a GET API /sessions/{session\_id}/students, which can give the list of students for that session, and a POST/PUT API, which we want to save attendance.

Consider that the GET API response is below:



In the above response, we can see that for the first two records attendance has been marked already, and for the third record, attendance is not already marked.

Now to mark attendance, we have two ways.

### 1. Send Student Attendance Information in Bulk

If we want to send the list of student attendance details in one API call, then we have to use POST API /sessions/{session\_id}/students because few records are for creating a new entry and few are to update the existing entry, and there is no unique identifier that can represent all records present in the request. So for such bulk API calls, the POST method is the only option.

### 2. Send Each Student's Attendance Information Using a Separate API Call

You can say that we should send each student record using a separate API call (which would cause unnecessary network calls, but let's ignore that for now). POST /sessions/{session\_id}/students/{student\_id}for creating attendance and PUT /sessions/{session\_id}/students/{student\_id}/attendances/{attendance\_id} for updating attendance. But have you observed the GET API response object (mentioned above)? We do not have any attendance\_id -related information in that, so we don't have any way to identify the resource uniquely.

You can argue that attendance\_id should be present in the GET API, but there would be many cases where would not have control of the GET API contract. Your job would be to parse the data that you are already getting and use that to send information to the server.

Thank you for reading. Please share your thoughts in the comments.

# **How to Get data on Get method REST API ?**

### Use HttpClient to make HTTP calls

In the newly created DataAccessService, import HttpClient from @angular/common/http.

Look at the following code snippet. Notice the service is decorated with injectable. It injects the HttpClient service. The HttpClient object is used to make HTTP GET call.

Observe the function getTravellers:

|  |
| --- |
| --- data-access.service.ts ---  import { HttpClient } from '@angular/common/http';    @Injectable()  export class DataAccessService {  constructor(private client: HttpClient) { }  --- data-access.service.ts ---  getTravellers(){  this.client.get(`${DATA\_ACCESS\_PREFIX}/traveller.json`)  .subscribe( (data) => console.log(JSON.stringify(data)));  } |

Notice the get function returns an observable. The subscribe function receives and prints JSON response from a server-side API. Figure-1 shows the console output

The response includes the travellers array each containing the traveller object. Imagine parsing this information to create traveller objects.

|  |
| --- |
| --- data-access.service.ts ---  this.client.get(`${DATA\_ACCESS\_PREFIX}/traveller.json`)  .subscribe( (data) => {  let travellers: Array<any> = data && data["travellers"];  console.log(travellers && travellers.map((traveller) => ({  'firstName': traveller['firstName'] || "",  'lastName': traveller['lastName'] || "",  'city': traveller['city'] || "",  'country': traveller['country'] || "",  'age': traveller['age'] || -1  })));  }); |

- We create an array object named travellers, from the response structure.

- We perform null checks.

- Transform each object to the expected structure.

- Notice traveller object’s field names are not available (while coding in the IDE). That’s because it’s of type Object. Hence we dynamically provide the expected field to be returned.

### Explicitly type response object

Considering we are using TypeScript, the above code can be simplified.

Create an interface for Travellers object expected to be returned from the service.

|  |
| --- |
| --- traveller.ts ---  export interface Traveller{  firstName: string;  lastName: string;  city: string;  country: string;  age: number;  }  export interface Travellers{  travellers: Array<Traveller>  } |

On calling get function, specify the type as a generics parameter. Refer to the highlighted code in the following snippet.

|  |
| --- |
| --- data-access.service.ts ---  this.client.get<font style="background-color: #4f81bd"><Travellers></font>(`${DATA\_ACCESS\_PREFIX}/traveller.json`)  .subscribe( (data) => {  data.travellers.map(traveller => console.log(`Hello ${traveller.lastName}, ${traveller.firstName}`));  }); |

Notice the subscribe function can access travellers and other objects on the data directly.

# **How to check any input is String or Object or Array ?**

var booleanValue = true;

var numericalValue = 354;

var stringValue = "This is a String";

var stringObject = new String( "This is a String Object" );

alert(typeof booleanValue) // displays "boolean"

alert(typeof numericalValue) // displays "number"

alert(typeof stringValue) // displays "string"

alert(typeof stringObject) // displays "object"

# **How to create MAP ,Set and List in angular?**

Map is a new data structure introduced in ES6 which lets you map keys to values without the drawbacks of using Objects.

### [**Creating, Getting and Setting**](https://codecraft.tv/courses/angular/es6-typescript/mapset/#_creating_getting_and_setting)

We create a map using the new keyword, like so

TypeScript

Copylet map = new Map();

We can then add entries by using the set method, like so:

TypeScript

Copylet map = new Map();

map.set("A",1);

map.set("B",2);

map.set("C",3);

The set method is also chainable, like so:

TypeScript

Copylet map = new Map()

.set("A",1)

.set("B",2)

.set("C",3);

Or we could initialise the Map with a an array of key–value pairs, like so:

TypeScript

Copylet map = new Map([

[ "A", 1 ],

[ "B", 2 ],

[ "C", 3 ]

]);

We can extract a value by using the get method:

TypeScript

Copymap.get("A");

// 1

We can check to see if a key is present using the has method:

TypeScript

Copymap.has("A");

// true

We can delete entries using the delete method:

TypeScript

Copymap.delete("A");

// true

We can check for the size (number of entries) using the size property:

TypeScript

Copymap.size

// 2

We can empty an entire Map by using the clear method:

TypeScript

Copymap.clear()

map.size

// 0

# **Add/ Retrieve/ Delete Entries from Map**

1. map.set() – method to add entries in Map
2. map.get() – to retrieve an entry from Map
3. map.has() – to existence of an entry in the Map
4. map.delete() – deletes an entry from the Map
5. map.size – ‘size’ property will return size of Map
6. map.clear()-to clear the Map data

### [**Looping over a Map**](https://codecraft.tv/courses/angular/es6-typescript/mapset/#_looping_over_a_map)

We use the for-of looping operator to loop over entries in a Map.

There are a couple of different method we can employ, we’ll go over each one using the below map as the example:

TypeScript

Copylet map = new Map([

[ "APPLE", 1 ],

[ "ORANGE", 2 ],

[ "MANGO", 3 ]

]);

#### [**Using keys()**](https://codecraft.tv/courses/angular/es6-typescript/mapset/#_using_keys)

The keys method returns the keys in the map as an array which we can loop over using for-of like so:

TypeScript

Copyfor (let key of map.keys()) {

console.log(key);

}

// APPLE

// ORANGE

// MANGO

#### [**Using values()**](https://codecraft.tv/courses/angular/es6-typescript/mapset/#_using_values)

The values method returns the values in the map as an array which we can loop over using for-of like so:

TypeScript

Copyfor (let value of map.values()) {

console.log(value);

}

// 1:

// 2

// 3

#### [**Using entries()**](https://codecraft.tv/courses/angular/es6-typescript/mapset/#_using_entries)

The entries method returns the [key,value] pairs in the map as an array which we can loop over using for-of like so:

TypeScript

Copyfor (let entry of map.entries()) {

console.log(entry[0], entry[1]);

}

// "APPLE" 1

// "ORANGE" 2

// "MANGO" 3

Using destructuring we can access the keys and values directly, like so:

TypeScript

Copyfor (let [key, value] of map.entries()) {

console.log(key, value);

}

// "APPLE" 1

// "ORANGE" 2

// "MANGO" 3

Looping over key-value pairs via entries is so common that this is the default for a Map.

Therefore we don’t even need to call entries() on a map instance, like so:

TypeScript

Copyfor (let [key, value] of map) {

console.log(key, value);

}

// "APPLE" 1

// "ORANGE" 2

// "MANGO" 3

#### **Important**

A distinction between Object and Map is that Maps record the order in which elements are inserted. It then replays that order when looping over keys, values or entries.

## [**Set**](https://codecraft.tv/courses/angular/es6-typescript/mapset/#_set)

Sets are a bit like maps but they only store keys not key–value pairs.

They are common in other programming languages but are a new addition to JavaScript in ES6.

### [**Creating, Getting and Setting**](https://codecraft.tv/courses/angular/es6-typescript/mapset/#_creating_getting_and_setting_2)

We create a Set using the new keyword, like so

TypeScript

Copylet set = new Set();

We can then add entries by using the add method, like so:

TypeScript

Copylet set = new Set();

set.add('APPLE');

set.add('ORANGE');

set.add('MANGO');

The add method is chainable, like so:

TypeScript

Copylet set = new Set()

.add('APPLE')

.add('ORANGE')

.add('MANGO');

Or we can initialise the Set with an array, like so:

TypeScript

Copylet set = new Set(['APPLE', 'ORANGE', 'MANGO']);

We can check to see if a value is in a set like so:

TypeScript

Copyset.has('APPLE')

// true

We can delete a value from the set:

TypeScript

Copyset.delete('APPLE')

We can count the number of entries in the set like so:

TypeScript

Copyset.size

// 2

We can empty the entire set with the clear method:

TypeScript

Copyset.clear();

set.size

// 0

Sets can only store unique values, so adding a value a second time has no effect:

TypeScript

Copylet set = new Set();

set.add('Moo');

set.size

// 1

set.add('Moo');

set.size

// 1

### [**Looping over a Set**](https://codecraft.tv/courses/angular/es6-typescript/mapset/#_looping_over_a_set)

We can use the for-of loop to loop over items in our set, like so:

TypeScript

Copylet set = new Set(['APPLE', 'ORANGE', 'MANGO']);

for (let entry of set) {

console.log(entry);

}

// APPLE

// ORANGE

// MANGO

#### **Important**

Similar to Maps, Sets also record the order in which elements are inserted, it then replays that order when looping.

# **How to create select and button input in angular?**

# **What is pipe in angular ? why it used?**

Pipes are a useful feature in Angular. They are a simple way to transform values in an Angular template. There are [some built in pipes](https://alligator.io/angular/built-in-pipes-angular/), but you can also build your own pipes.

A pipe takes in a value or values and then returns a value. This is great for simple transformations on data but it can also be used in other unique ways. This post will highlight a few useful or unique use cases that we found for pipes.

The ratings on each of the points indicate how much I believe this particular use case abuses the pipe framework.

## **1. Return Default Values**

The functionality behind a Default Pipe is pretty self explanatory—if a value is falsy, use a default value instead. The implementation is also very simple.

@Pipe({name: 'default', pure: true})

export class DefaultPipe {

transform(value: any, defaultValue: any): any {

return value || defaultValue;

}

}

The default pipe uses Angular’s ability to pass multiple values into the pipe to get both the value and the default value.

An example of using the default pipe in Angular would be:

<lucid-icon [name]="folder.icon | default:'Folder'"></lucid-icon>

In the example above, either the folder has an icon that is used or the Folder icon is used as a default. You can play with the example code using the Default Pipe in the Plunker below.

## **2. Debounce Input**

A Debounce Pipe is much more technically interesting than the Default Pipe. The basic premise of the idea is that the passed-in value can change frequently, but the actual value returned from the pipe will not change until the value has remained changed for a certain period of time. This can be very useful when listening to user input but not wanting to update the user interface until after the user has finished typing.

You can see the implementation below:

@Pipe({name: 'debounce', pure: false})

export class DebouncePipe {

private currentValue: any = null;

private transformValue: any = null;

private timeoutHandle: number = -1;

constructor(

private changeDetector: ChangeDetectorRef,

private zone: NgZone,

) {

}

transform(value: any, debounceTime?: number): any {

if (this.currentValue == null) {

this.currentValue = value;

return value;

}

if (this.currentValue === value) {

// there is no value that needs debouncing at this point

clearTimeout(this.timeoutHandle);

return value;

}

if (this.transformValue !== value) {

// there is a new value that needs to be debounced

this.transformValue = value;

clearTimeout(this.timeoutHandle);

this.timeoutHandle = setTimeout(() => {

this.zone.run(() => {

this.currentValue = this.transformValue;

this.transformValue = null;

this.changeDetector.markForCheck();

});

}, typeof debounceTime == 'number' ? debounceTime : 500);

}

return this.currentValue;

}

}

The Debounce Pipe takes in a value, and then if the value has changed since the last time the debounce completed (or it is null), it will wait either the time passed in as the second value or 500 ms if a time wasn’t provided, and then apply the new value.

An example of using it in Angular would be:

<div

\*ngIf="hasInputError(contentOption) | debounce"

class="error-message"

>

{{errorMessage(contentOption)}}

</div>

You can play with the example code using the Debounce Pipe in the Plunker below.

## **3. Get the Position of an Element**

Angular has a handy feature that lets you assign an element or component to a variable and then reference that variable within that template. The Element Position Pipe takes advantage of that feature and allows you to pass an element to the pipe and have it return the position. This can be a useful feature for deciding where to position a pop-up or some other element.

Here is the implementation:

@Pipe({name: 'elementPosition', pure: true})

export class ElementPosition {

transform(value: HTMLElement, xLerp: number, yLerp: number): Point|null {

if (value != null) {

const boundingRect = value.getBoundingClientRect();

return {

x: boundingRect.left + xLerp \* boundingRect.width,

y: boundingRect.top + yLerp \* boundingRect.height,

};

} else {

return null;

}

}

}

The two numbers that are passed in with the element are used to decide where on the element the position should be. The first value is used in the x position and is multiplied by the width. So if you want to get the position of the right side of the element, you would pass in 1 for the first parameter, which would mean x + 1 \* width. The second parameter is the same but for the y value and the height.

An example of using the Element Position Pipe in Angular would be:

<div

#titleElement

(click)="expanded = !expanded"

>

{{title}}

</div>

<ng-container popup [visible]="expanded">

<lucid-menu

\*popupContent

[items]="menuOptions"

[position]="titleElement | elementPosition:0:1"

></lucid-menu>

</ng-container>

In this simplified snippet, the menu is getting the bottom-left position of the `titleElement` passed into its `[position]` input.

## **4. Feign Natural Typing**

A flashy feature that I have seen throughout the web is animating text as if it was being typed by a user. An intuitive way to implement such a feature in Angular would be using pipes.

Below is a simple implementation of a Natural Typing Pipe:

@Pipe({name: 'naturalType', pure: false})

export class NaturalType {

private typed: string = '';

private target: string = '';

private currentIndex: number = -1;

private timeoutHandle: number = -1;

constructor(

private changeDetector: ChangeDetectorRef,

private zone: NgZone,

) {

}

transform(value: string, mintypingSpeed: number = 30): any {

if (this.target !== value) {

clearTimeout(this.timeoutHandle);

this.typed = '';

this.currentIndex = -1;

this.target = value;

this.typeNextCharacter(mintypingSpeed);

}

return this.typed;

}

private typeNextCharacter(mintypingSpeed: number) {

this.currentIndex++;

this.typed = this.target.substr(0, this.currentIndex);

this.changeDetector.markForCheck();

if (this.typed !== this.target) {

const time = Math.round(Math.random() \* 70) + mintypingSpeed;

this.timeoutHandle = setTimeout(()=> {

this.zone.run(() => this.typeNextCharacter(mintypingSpeed));

},time);

}

}

}

Usage of the pipe is pleasantly simple—simply pass a string through the pipe.

{{value | naturalType}}

You can play with the example code using the Natural Typing Pipe in the Plunker below

## **5. Track User Input**

Analytics and recently used lists are two features that come to mind when I think of tracking user input. It seems possible to do that with a pipe. So why not?

First, we will need a tracking service.

@Injectable()

export class TrackingService {

private wordsUsed: Set = new Set();

public addWordUsed(word: string) {

this.wordsUsed.add(word);

}

public getWords(): string[] {

return Array.from(this.wordsUsed);

}

}

Then we will need a pipe to feed the service.

@Pipe({name: 'track', pure: true})

export class TrackingPipe {

constructor(

private trackingService: TrackingService,

) {

}

transform(value: string): string {

this.trackingService.addWordUsed(value);

return value;

}

}

Consuming the feed is then simple with Angular.

@Component({

...

template: `

...

<div \*ngFor="let word of getWords()">{{word}}</div>

`,

})

export class AppComponent {

...

constructor(private trackingService: TrackingService) {

}

public getWords(): string[] {

return this.trackingService.getWords();

}

}

After running, it doesn’t render the list immediately. Checking the console we have this error: ERROR Error: ExpressionChangedAfterItHasBeenCheckedError: Expression has changed after it was checked. This error makes sense. When the change detector runs, it transforms the value using the tracking pipe which modifies the list. Since the value changed during change detection, the change detection is then out of date and has to run again. If it changes during change detection … ∞. Since we were smart ?, we put the debounce pipe that we built earlier in front of the tracker pipe ({{inputValue | debounce:200 | track}}), so we know that the value isn’t going to actually change every change detection and cause an infinite loop. So we can just make our logging run after an async callback, and then everything should work.

@Pipe({name: 'track', pure: true})

export class TrackingPipe {

constructor(

private trackingService: TrackingService,

private changeDetector: ChangeDetectorRef,

private zone: NgZone,

) {

}

transform(value: string): string {

Promise.resolve().then(() => {

this.trackingService.addWordUsed(value);

this.zone.run(() => this.changeDetector.markForCheck());

});

return value;

}

}

If we wanted to, we could even log the values over the network and then we wouldn’t have to fake the async! ?

You can play with the example code using the Tracking Pipe in the Plunker below.

# **How is popup in angular ?**

# **What is Differance between ng-bootstrap and primeng ?**

# **What is amCharts in angular ?**

amCharts used for responsive graph -Map, XY graph,Pie,Venn Diagram

https://www.amcharts.com/docs/v4/getting-started/

**What is primeng in angular ?**

PrimeNG :PrimeNG is a rich set of open source native Angular UI components.

PrimeNG Components used for responsive -widget like Button.grid,table,menu,Popup etc.

More:https://primefaces.org/primeng/showcase/#/setup

**Can you explain various ways of component communication in Angular?**

1. Data sharing between parent and one or more child components using the @Input() and @Output() directives.
2. Data sharing using an Angular service
3. Using state management, like [NgRx](https://ngrx.io/)
4. Read and write data to local storage
5. Pass data via URL parameters

**How can you bind data to templates?**

**Property binding**: Property binding in Angular helps you set values for properties of HTML elements or directives

<img [src]="itemImageUrl">

**Event binding:** Event binding allows you to listen for and respond to user actions such as keystrokes, mouse movements, clicks, and touches.

<button (click)="onSave()">Save</button>

**Two-way binding:** Two-way binding gives components in your application a way to share data. Use two-way binding binding to listen for events and update values simultaneously between parent and child components.

<app-sizer [(size)]="fontSizePx"></app-sizer>

**What is the difference between ViewChild and ContentChild?**

ViewChild and ContentChild are used for component communication in Angular, for example, if a parent component wants access to one or multiple child components.

A ViewChild is any component, directive, or element which is part of a template.

A ContentChild is any component or element which is projected in the template.

In Angular exist two different DOMs:

Content DOM which has only knowledge of the template provided by the component at hand or content injected via <ng-content>.

View DOM which has only knowledge of the encapsulated and the descending components.

**What are @HostBinding and @HostListener?**

@HostListener() function decorator allows you to handle events of the host element in the directive class. For example, it can be used to change the color of the host element if you hover over the host element with the mouse.

@HostBinding() function decorator allows you to set the properties of the host element from the directive class. In this directive class, we can change any style property like height, width, color, margin, border, etc.

**What is Angular? What is the difference between Angular and Vue.js / React?**

Angular is an application design framework and development platform for creating efficient and sophisticated single-page apps. Angular is built entirely in TypeScript and uses it as a primary language. As it is a framework it has many useful built-in features like routing, forms, HTTP client, Internationalization (i18n), animations, and many more.

Vue.js and React are no application frameworks but JavaScript libraries to build user interfaces. Vue.js describe itself as an incrementally adoptable ecosystem that scales between a library and a full-featured framework and React as a JavaScript library for building user interfaces.