***What is Angular:***

Angular is a JavaScript Framework which allows you to create reactive Single-Page-Applications (SPA’s)

Versions :

Angular 1 : started in 2009 (3 years After jQuery)

Angular 2 : 2015 (Complete Rewrite)

***Angular Folder Architecture:***

<https://mdbootstrap.com/angular/angular-project-structure/>

Go through this link for further more information….

***What are Domain Models ?***

A good practice when writing Angular code is to try to isolate the data structures you are using from the component code.

Understanding: It’s just a plain class which we will use to store data and functions.

***What is NPM:***

Understanding: As we all know npm is node package manager.

NPM is two things:

It’s an Online Platform and

It’s a Command Line Tool

Online Platform: Here everyone can publish and share tools written in JavaScript.

The tools can be used in Browser (FRONT END), Server (BACK END), Command Line.

Command Line Tool: Install and Uninstall packages (A package is a tool someone created and uploaded to the npm online platform. These are like bui006Cding blocks you can add them to get a new package and can develop a new cool application)

Version Management: Every package has a version as the package changes, the package version changes npm makes it easy to keep packages up to date. Even lets you switch to any version, whenever you want.

Dependency Management: With one NPM Command, you can install a package along with all of its dependencies.

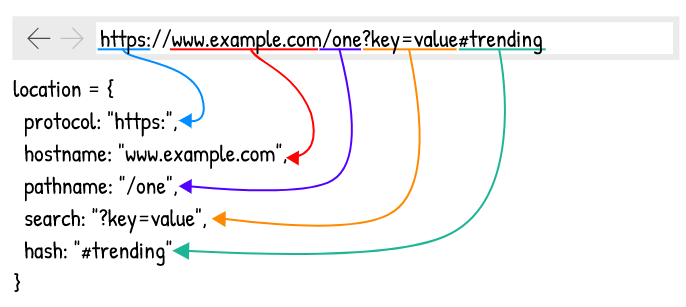
***SPA and MPA:***

**Single Page Application:**

A single-page application (SPA) is a website that re-renders its content in response to navigation actions (e.g. clicking a link) without making a request to the server to fetch new HTML.

While single-page application implementations vary, most rely on the same browser behavior and native APIs to enable the core functionality. Understanding these is key to knowing how single-page applications work.

SPA’s use window.location. This allows you to interact with the different parts of the URL without having to parse it yourself.



Only three of the location object’s properties are important for an SPA:

Pathname, hash and search (query string)

Pathname is typically the most important of these three properties because it is the one used for determining what content to render.

While the hash and search provides additional data. For example, in the URL */images?of=mountains*  , the /images pathname would specify that an images page should be rendered, while the ?of=mountains search specifies the type of images that should be rendered within the images page.

Note: You can prevent native behavior of the browser by using **event.preventDefault()**

History API: The History API has three core functions: **pushState(), replaceState(), and go().**

These are accessed via **window.history**.

**Multi-Page Application:**

Multiple-page applications work in a “traditional” way. Every change e.g. Display the data or submit data back to server requests rendering a new page from the server in the browser.

These applications are large, bigger than SPA’s because they need to be. Due to the amount of content, these applications have many levels of UI. Luckily, it’s not a problem anymore.

To overcome this, we have Ajax. We don’t have to worry about the big and complex applications have to transfer a lot of data between server and browser.

The solution improves and it allows refreshing only particular parts of the application.

On the other hand, it adds more complexity and it is more difficult to develop than a single-page application.

***Package.json***

The CLI ng new command creates a default package.json file for your project. This package.json specifies a starter set of packages that work well together and jointly support many common application scenarios.

You will add packages to package.json as your application evolves. You may even remove some.

**dependencies and devDependencies:**

The package.json includes two set of packages, dependencies and Devdependencies.

The dependencies are essential to running the application. The devDependencies are only necessary to develop the application

**Dependencies:**

The dependencies section of package.json contains:

* Angular packages : Angular core and optional modules; their package names begin @angular/
* Support packages: 3rd party libraries that must be present for Angular apps to run.
* Polyfill packages: Polyfills plug gaps in a browser’s JavaScript implementation.

**Polyfill packages:**

Many browsers lack native support for some features in the latest HTML standards, features that Angular requires. “ Polyfills ” can emulate the missing features. The browser support guide explains which browsers need polyfills and how can you add them.

Here package.json installs the core-js package which polyfills missing features for several popular browser.

**DevDependencies:**

The packages listed in the devDependencies section of package.json help you develop the application on your local machine.

It’s not necessary to deploy them with the production application, although there is no harm in doing so.

***Directive:***

Directive allows you to make a class as an Angular directive and provide additional metadata that determines how the directive should be processed, instantiated and used at runtime.

Directives allow you to attach behavior to elements in the DOM.

A directive must belong to an NgModule in order for it to be usable by another directive, component or application. To specify that a directive is a member of an NgModule, you should list in the declarations field of the NgModule.

Three types of Directives :

**Components Directives**

**Structural Directives**

Structural Directives are used to add or remove the HTML elements.

Starts with \* , for Example, \*ngIf and \*ngFor

**\*ngFor:**

NgForOf is a repeater directive – a way to present a list of items.

We can use the Angular \*ngFor directive in the template to display each item in the list.

***<li \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes"> {{ hero }} </li>***

NgFor with index:

The index property of the NgForOf directive context returns the zero-based index of the item in each iteration. You can capture the index in a template input variables and use it in the template.

<li \*ngFor = “let hero of heroes; let i=index”>{{i+1}} – {{hero.name}}</li>

**Attribute Directives**

Are only used to change the elements they were placed on . We can say they used to modify the DOM elements.

**Understanding : Directives are instructions to the DOM.**

***Component:***

Components are the most basic UI building block of an Angular app. Angular app contains a tree of Angular components.

Angular components are a subset of directives, always associated with a template. Unlike other directives, only one component can be instantiated per an element in a template.

**Understanding**: Components are basically classes that interact with the .html file of the component, which gets displayed on the browser.

***Modules:***

Module in Angular refers to a place where you can group the components, directives, pipes and services, which are related to the application.

To define module, we can use the NgModule.

***Selectors:***

Selectors are like CSS selectors. They can be attribute selectors, tag selectors, class selectors, Id selectors and combinations of these.

We use them to render our component to the end user.

Declare as one of the following:

***element-name: Select by element name.***

***.class: Select by class name.***

***[attribute]: Select by attribute name.***

***[attribute=value]: Select by attribute name and value.***

***:not(sub\_selector): Select only if the element does not match the sub\_selector.***

***selector1, selector2: Select if either selector1 or selector2 matches.***

***Decorators :***

Decorators are functions that are invoked with a prefixed @ symbol, and immediately followed by class , parameter or method , and the decorator functions returns something in its place, or manipulates its target in some way.

There are four main types:

* Class decorators e.g. @Component and @NgModule
* Property decorators for properties inside classes. e.g. @Input and @Output
* Method decorators for methods inside classes, e.g. @HostListener
* Parameter decorators for parameters inside class constructors, e.g. @Inject

**Class Decorators:**

Angular offers us class decorators. These are the top level decorators that we use to express intent (Objective) for classes. They allow us to tell Angular that a particular class is a component, or module, for example. And the decorator allows us to define this intent without having to actually put any code inside the class.

A @Componet and @NgModule decorator example with classes:

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

***@Component({***

***selector: 'example-component',***

***template: '<div>Woo a component!</div>',***

***})***

***export class ExampleComponent {***

***constructor() {***

***console.log('Hey I am a component!');***

***}***

***}***

***@NgModule({***

***imports: [],***

***declarations: [],***

***})***

***export class ExampleModule {***

***constructor() {***

***console.log('Hey I am a module!');***

***}***

***}***

Notice how both classes by themselves are effectively the same. No code is needed within the class to tell Angular that it is a component or a module. All we need to do is decorate it, and Angular will do the rest.

**Property Decorators:**

These are probably the second most common decorators that you’ll come across. They allow us to decorate specific properties within our classes – an extremely powerful mechanism. For example we have @Input and @Output (which are used input and output binding)

**Method Decorators:**

Method Decorators are very similar to property decorators but are used for methods instead. This lets us decorate specific methods within our class with functionality.

A good example of this is @HostListener. This allows us to tell Angular that when an event on our host happens, we want the decorated method to be called with the event.

**Parameter Decorators:**

Parameter decorators are quite interesting. You may have come across these when injecting primitives into a constructor, where you need to manually tell Angular to inject a particular provider.

Parameter decorators allow us to decorate parameters in our class constructors. An example of this is @Inject that lets us tell Angular what we want that parameter to be initiated with.

***Data-Binding:***

Link to understand: <https://blog.cloudboost.io/basic-data-binding-with-angular-67680fc149e6>

Data binding is a mechanism for coordination what users see, with application data values. While you could push values to and pulls values from HTML.

Understanding: Data Binding coordinated (correlate, fit together) the communication between a component class and the template that it’s associated with.

We have 4 different types of Data Binding in Angular:

* Interpolation Binding / String Interpolation.
* Property Binding
* Event Binding
* Two-way Binding

**Interpolation Binding:**

Interpolation Binding is more than likely the first type of binding people will come across.

You use Interpolation Binding to take expressions and change them into text which can be used within HTML element tags and attribute values.

All we have to do is place the expression in the double-curly braces as shown here.

<p>The person's name is {{person.name}}.</p>

You cannot use multi line expression, in string Interpolation.

We cannot use any conditional statements such as if, while etc.

Though we can use ternary operations here.

**Property Binding:**

Property Binding allows us to set the value of a property on an HTML element to the value of a template expression.

Here I’m setting the src value on an image element to the imageUrl attribute on a project object and then setting the title value from the same product object.

<img [src]='product.imageUrl' [title]='product.productName'/>

**Event Binding:**

Event Binding allows our component to listen to events triggered by user actions in view.

The event is enclosed in the parenthesis followed by the method that need to be called when that event is triggered.

<button (click)='updateProduct()'>  
 Update  
</button>

**Two-Way Binding:**

Two way binding is where the value of a property on the component is displayed in say an input element and the change to the element updates the property on the component.

<input type="text" [(ngModel)]='firstName' />

The square brackets indicate Property Binding and the parenthesis indicate Event Binding to send a notification of the user entered data back to the property.

***Interpolation:***

Interpolation is a technique that allows the user to bind a value to a UI element.

Interpolation binds the data one-way. This means that when value of the field bound using interpolation changes, it is updated in the page as well. It cannot change the value of the field. An object of the component class is used as data context for the template of the component. So the value to be bound on the view has to be assigned to a field in the component class.

Simply place the component value inside the {{}} (mustache symbols or curly braces)

***Parent to Child:***

Create a receivable value in **child.component.ts**

**@Input () receivable;** //Make sure you import the value from the **@angular/core**

Then this receivable has to get the data from the parent , so parent has some data to send which will be the **parent.component.ts**

To get this data, you need to do property binding in the child selector at the **parent.component.html.**

**<app-child [receivable]=’ParentData’></app-child>**

Now you’ll have the access to the **parentData** , but you need to use the **receivable** variable in your child source to use it.

***Child to Parent:***

Child to parent is little complicated.

You need to create a variable again in **child.component.ts**

**@Output() output = new EventEmitter**; //Make sure you import **output and EventEmitter** from **@angular/core**

To receive the data in the parent create a variable so that you can use the child data in the parent .

**Parent.Component.ts**

**Receivable;**

Then this output has to send the data from the child to the parent, so child will have some data to send which will be in **child.component.ts**

To get this data, you need to event bind the child selector at the **parent.component.html.**

Before that you need to write a function to trigger the event, which will emit the value. In our case for example:

**Send(){**

**this.output.emit(“Needed Data”)**

**}**

**Child.component.html**

**<button (click)=’send()’></button>**

**Parent.component.html**

**<app-child [receivable]=’ParentData’ (output)=” Receivable = $ ”></app-child>**

Because $event will carry the payload of information with which it is used.

Now you can use the variable Receivable in anywhere in the **parent.component.html.**

***Custom directives:***

To create a custom directive all you need to do is to write a new file with the respected name.

Then you will create it similar to the Component but you will write as follows.

import { Directive,ElementRef, Renderer, HostListener, HostBinding } from "@angular/core";

@Directive({

selector:'[AppHover]'

})

export class AppDirective{

@HostBinding('class.card-outline-primary') private isHiding:boolean = false;

constructor(private eleref: ElementRef , private renderer : Renderer){

// renderer.setElementStyle(eleref.nativeElement,'backgroundColor', 'lightgray');

}

@HostListener('mouseover') onMouseOver(){

console.log("Mouse In");

this.isHiding=true;

}

@HostListener('mouseout') onMouseOut(){

console.log("Mouse out");

this.isHiding=false;

}

}

It is not recommended to use ElementRef as it says it makes your application vulnerable to XSS attack.

So it is good way to add renderer which works the same as the ElementRef, but protects you from XSS attack.

@HostBinding : Allows you to access the properties of the element that it is attached to.

Renderer.setElementStyle(eleref.nativeElement,’backgroundColor’,’White’)

**How to create a custom directive :**

*We annotate a class with the @Directive decorator.*

**How would you define a selector which matches to a directive to an element by the class name foo ?**

*We will write out selector: “.foo”.*

**How does a directive get a reference to the DOM element it’s attached to ?**

*We inject into directive class constructor an instance of ElementRef.*

**How can we make a custom directive react to events emitted on the host element ?**

*We annotate a function with the @HostListner decorator, passing to the @HostListner the name of the event we want to listen for.*

*@HostListner (‘mouseover’) func(){} //listen to the event and if it happens run this function.*

**How can we bind a host components property to a property of our custom directive ?**

*We annotate a property on our custom directive with the @HostBinding decorator, passing to @HostBinding the path to the property we want to bind to in the host element.*

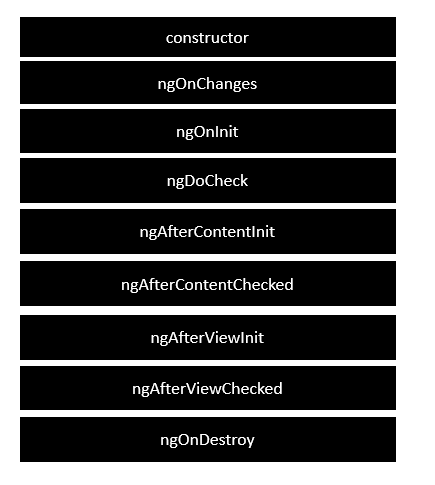
*@HostBinding (‘class. Thing-class’) localvar : boolena;*

**How do we make a custom directive configurable ?**

*We simply add an @Input to the directives class and bind to that input in the template.*

*@Input() data: Object;*

***Angular LifeCycle Hooks:***



<https://angular.io/guide/lifecycle-hooks>

***Angular forms and Forms Module:***

Two types of Forms one is Template Driven Form and Other in Reactive Form

We create a simple HTML form. We created a form model on our component using the FormGroup and FormControl classes.

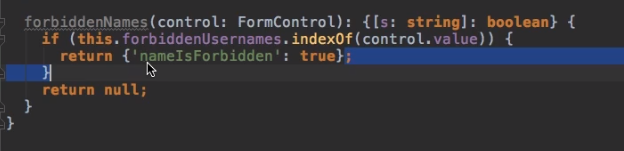
Then by using directives such as formGroup, FormControlName and formGroupName. We linked our HTML form to our form model.

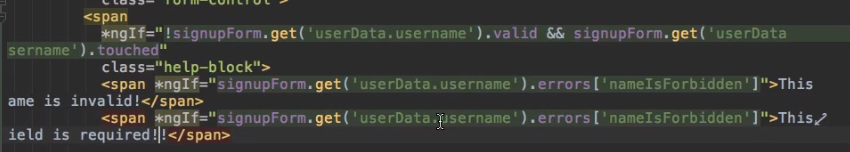
Bind [formGroup] = ‘myform’ // the name you created in component .ts for myform : FormGroup;

***You can add custom form validators using the AbstractControl which is the parent of the formgroup. (Template driven form)***

In a Reactive Form you can just add the validator as a function , which is actually a validate which takes ***FormControl*** and return ***keyValue pairs : Boolean***. ***(Reactive Control form)***

***The returned key value pairs which is in the end is the key-> ErrorName and value-> Boolean***

******



This is how you will get the error name for more specific error representation.

***Async-Validators:***

These Async Validators are similar to the custom validators which also need to be the instance of the FormControl and must return a promise or Observable



***You can also use the validators to listen to valuechanges and StatusChanges this will fire for every key input.***

***Status Changes returns whether the form is valid or invalid.***

***Template Driven Forms Features***

* Easy to use
* Suitable for simple scenarios and fails for complex scenarios
* Similar to AngularJS
* Two way data binding(using [(NgModel)] syntax)
* Minimal component code
* Automatic track of the form and its data(handled by Angular)
* Unit testing is another challenge

***Reactive Forms Features***

* More flexible, but needs a lot of practice
* Handles any complex scenarios
* No data binding is done (immutable data model preferred by most developers)
* More component code and less HTML markup
* Reactive transformations can be made possible such as
* Handling a event based on a debounce time
* Handling events when the components are distinct until changed
* Adding elements dynamically
* Easier unit testing

***Reactive: Setting and Patching Values (FORMS)***

We also have two more properties , which are used to set some values to the form(default).

1. this.signUpForm.setValue({ //Complete form has to set

username: “Default Value”,

email: “Some Default Value”

})

1. this.signUpForm.patchValue({ with the same above template}) // part of the form can be set, hence name patch.

We can use this.signUpForm.reset() to reset the form .

***View Encapsulation***

Shadow DOM refers to the ability of the browser to include a sub tree of DOM elements into the rendering of a document, but not into the main document DOM tree.(encapsulated Behavior)

View encapsulation defines whether the template and styles defined within the component can affect the whole application or vice versa. Angular provides three encapsulation strategies

They are three view encapsulation strategies:

***Emulated (default)*** : Styles from main HTML propagate to the component . Styles defined the Component’s ***@Component*** decorator are scoped to this component only.

***Native:*** Styles from main HTML do not propagate to the component. Styles defined in the Component’s @Component decorator are scoped to this component.(Native === Local )

***None:*** Styles from the component propagate back to the main HTML and therefore are visible to all components on the page. (It works as main css and apply to all the elements that are defined with that tag or class , and the other components still have their encapsulated behaviour)

Be careful with apps that have ***None*** and Native Components in the Application. All components with ***None*** encapsulation will have their styles duplicated in all components with Native encapsulation.

Nghost...

@Component({

selector: 'app-server-element',

templateUrl: './server-element.component.html',

styleUrls: ['./server-element.component.css'],

encapsulation:ViewEncapsulation.Emulated

})

***Ng-template***

The <ng-template> is an Angular element for rendering HTML. It is never displayed directly. In fact, before rendering the view , Angular replaces the <ng-template> and its contents with a comment.

If there is no structural directive and you merely wrap some element in a <ng-tempalte > those elements disappear.

***<p>Hip!</p>***

***<ng-template> <p>Hip!</p> </ng-template>***

***<p>Hooray!</p>***

The middle tags defined inside the ng-template would be hided as comment inside the HTML Page.

***<ng-container>***

The Angular <ng-container> is a grouping element that doesn’t interfere with styles or layout because Angular doesn’t put it in the DOM.

It’s basically a logical container and is used to group nodes together and support structural directives.

**Content Projection**

<https://medium.com/@ssokurenko/understanding-content-projection-in-angular-3ee45e4c041c>

projected using <ng-content></ng-content>

***Services and Dependency Injection***

These are Centralized Business Logic

Reusable code that can be accessed from multiple Components.

Services are functions that allows you to access its defined properties and methods and also helps to keep your coding organized.

To access a service inside another service you need to have @injectable in the service you want to write the service.

To create its instance you need make that instance in the constructors inside the arguments. Making the component to know that it is the type service(Mandatory).

***We use services for Cross-Component Communication***

**Dependency Injection:**

When a module A in an application needs module B to run, then module B is a dependency of module A. When an instance has been created in Module A to run this means we are injecting module A property or method inside module B. Thus we called it as Dependency Injection.

**Hierarchical Injector:**

When you have a service to get the service throughout the complete Application we will write the service in the AppModule under providers. While if you want to use in a particular component then it has to be under the respective component under providers.

If you create a instance in the parent component all of its child components will receive this service (components under its tree components)



Services you can also use by using EventEmitter which will emit some value in the function or some other value which can be received in the constructor by subscribing inside function every time a change in the value the subscribe will be called and it will assign the value to the set value.

***Router***

A router

<router-outlet></router-outlet>

routerLink = ‘/’ //specify the path in which it has to be

routerLinkActive = “active” // add css class inside it to know which router is active.

[routerLinkActiveOptions]=”{exact : true }” // this is to match the exact path true

You mention it in the app root component using RouterModule.forRoot() mention with which path is attached to which component.

To trigger programmatically we use by creating the instance of the router here and then by calling it in the required function by using this.instancename.navigate([‘PageName’])

This also takes an second argument as object, which is **relativeTo:** which helps to direct the page from the providedPagename // pageName/relativeName

Accessing the routing parameters:

App-Module.ts

const appRoutes: Routes = [

{path: '', component: HomeComponent},

{path: 'users', component: UsersComponent},

{path: 'users/:id/:name', component: UserComponent},

{path: 'servers', component: ServersComponent}

];

User.component.ts //the component in which you want to access the files.

export class UserComponent implements OnInit {

user: {id: number, name: string};

constructor(private route:ActivatedRoute) { }

ngOnInit() {

this.user = {

id : this.route.snapshot.params['id'],

name : this.route.snapshot.params['name']

}

}

}

**Absolute path:**

This path will always append to the root path;

<a router-link = ‘/servers’>reload<a>

This will always added to the domain path, that means if your domain name is [www.domain.com/servers](http://www.domain.com/servers) the servers will append to this domain path that is ***www.domain.com/servers***

**Relative path:**

This path will always append to the present path it is in;

<a router-link = ‘servers’>reload<a>

This always add to the existing path, that is if your current path is [www.domain.com/servers](http://www.domain.com/servers) the servers will append to this path that is to

[***www.domain.com/servers/servers***](http://www.domain.com/servers/servers)

You can programmatically change the path you are in that is clicking on something or doing some DOM events. This has to be done in ***Component.ts*** where you will add the instance of Router in the component and with the variable of that instance we will use the ***.navigate([])*** method which takes the input as a string.

<div class="p-4">

<h1>Home is on process...</h1>

<p>We use some complex calculations </p>

<button class="btn btn-primary btn-lg" (click)="server()">Load blog</button>

</div>

Import { Router } from ‘@angular/router’

@Component({

selector: 'app-home',

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

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

})

export class HomeComponent implements OnInit {

constructor(private route:Router) { }

ngOnInit() {

}

server(){

this.route.navigate(['/blog'])

}

}

To get the Data from the URL angular has a library which is ***ActivateRouter*** available in ***Router***  imported from “@Angular/router” and then we create a instance of it in the constructor and then we use the instanced variable to access the URL metadata, it is only possible if you declared in the RouterModule that the URL of specified component has some MetaData.

You use ***variableName.snapshot.params[],*** snapshot is a method inside the ActivateRouter gives you access the information to be precise the metadata to use .

We also use ***queryParams*** to have some search queries and ***fragments*** to get the extra URL metadata.

<a [routerLink] = "['/work']" [queryParams] = "{ allowEdit: 1 }"

[fragment] = "'somemore'" >Some Link</a>

We can get this queryparams and fragment data by using the ActivatedRoute similar to params to get this whole data from the URL.

constructor(private route:Router,

private aroute: ActivatedRoute) { }

ngOnInit() {

console.log(this.aroute.snapshot.queryParams);

console.log(this.aroute.snapshot.fragment)

}

To be have a proper use of this whole thing you need to subscribe to this params, queryParams and fragments.

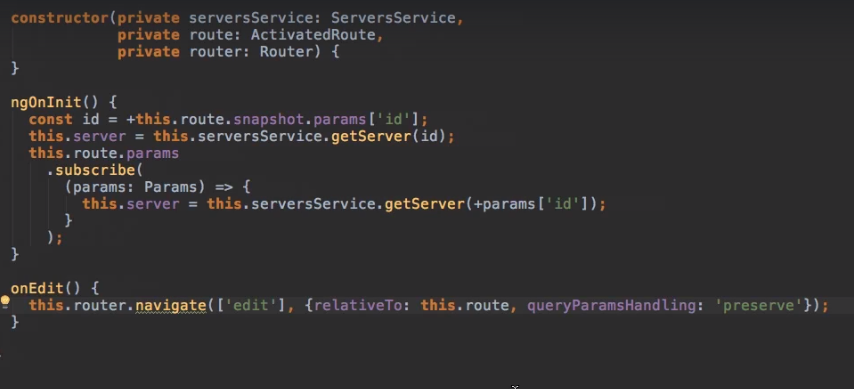
this.aroute.queryParams.subscribe((params : Params)=>{

this.aroute.Prams = this.params;

})

CanActivate,CanActivateChild, PageNotFoud using wildcard “\*\*”.

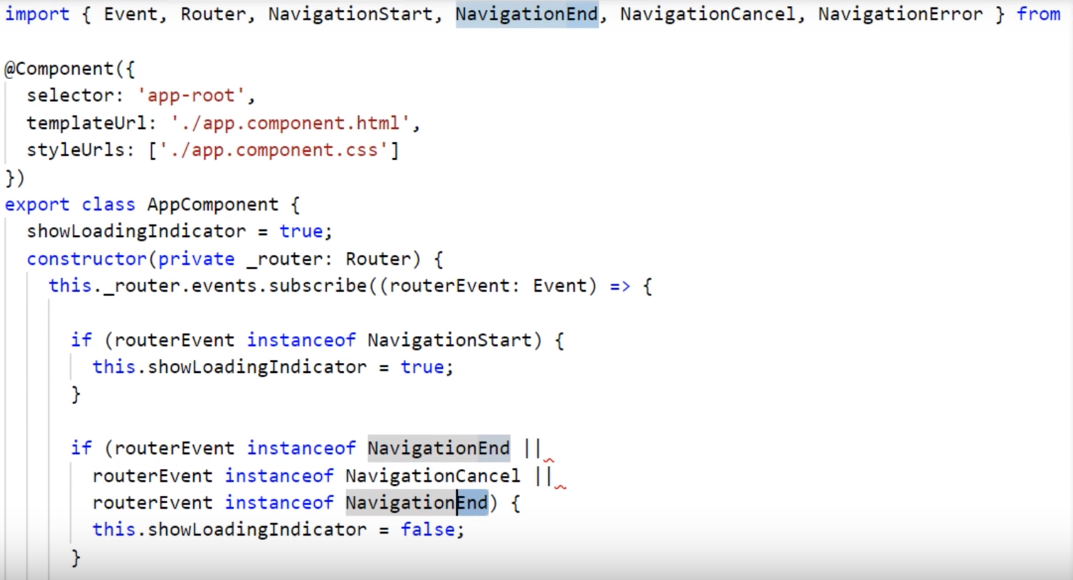
Super Important: We use queryParamsHandling: ‘preserve’ to keep the last visited url saved to use it in the further pages when you navigate away from the current pages



We also have **Router Event’s** in Router package which will fire whenever a event has been take place in the Router, you have some useful properties there for these ***Router Events*** and we need to subscribe to the event to use the following properties. Such as..

1. NavigationStart
2. NavigationEnd
3. NavigationCancel
4. NavigationError

From the Angular Routers.



Link: https://www.youtube.com/watch?v=V\_64FqedqW0

***Guard Types:***

There are four different guard types we can use to protect our routes:

* ***CanActivate –*** Decides if a route can be activated
* ***CanActivateChild-*** Decides if children routes of a route can be activated
* ***CanDeactivate-*** Decides if a route can be deactivated
* ***CanLoad-*** Decides if a module can be loaded lazily

<https://blog.thoughtram.io/angular/2016/07/18/guards-in-angular-2.html>

Depending on what we want to do, we might need to implement one or the other guard. In some cases, we even needed to implement all of them.

***Guards:***

Guards can be implemented in different ways, but after all it really boils down to a function that returns either Observable<boolean>, Promise<boolean> or boolean. In addition, guards are registered using providers, so they can be injected by Angular when needed.

Guards are great. They enable us to protect certain routes or even protect the user from losing data. In addition, we can have multiple guards protecting a single route which helps us implementing sophisticated use cases, where a chain of different checks is needed.

CanActivate receives two arguments

1. ActivateRouteSnapshot
2. RuterStateSnapshot

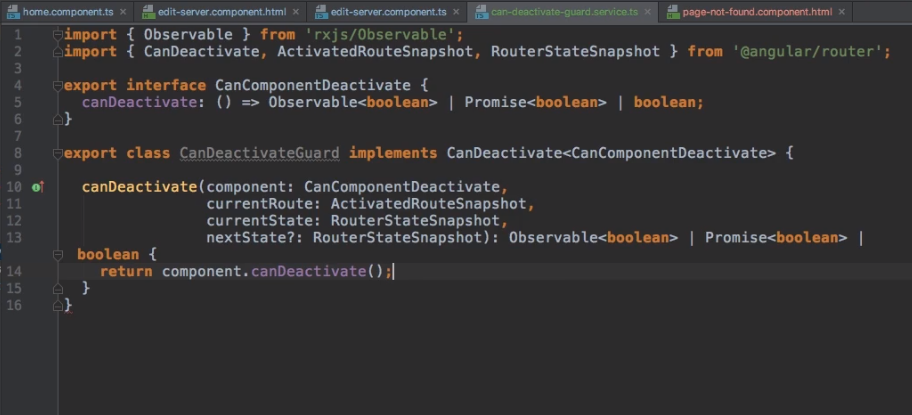
And returns a Observable<boolean> | Promise<boolean> | Boolean.



CanDeactivate receives three arguments

1. ComponentInterface we create in service
2. ActivatedRouteSnapshot
3. RouterStateSnapshot
4. RouterStateSnapshot : NextState()

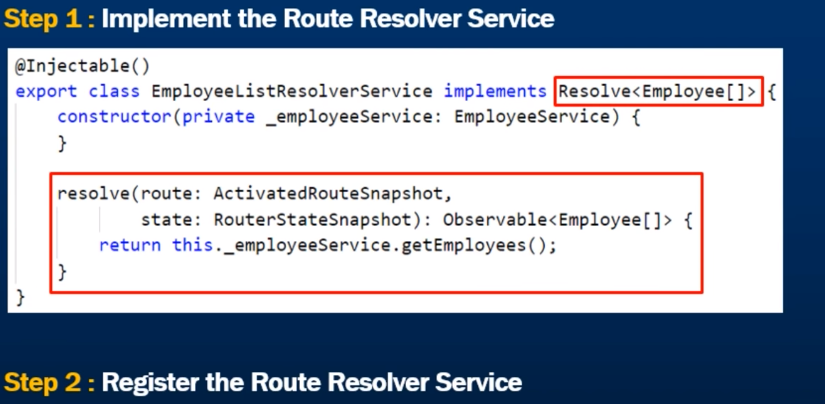
And returns a Observable<boolean>, Promise<boolean>, Boolean

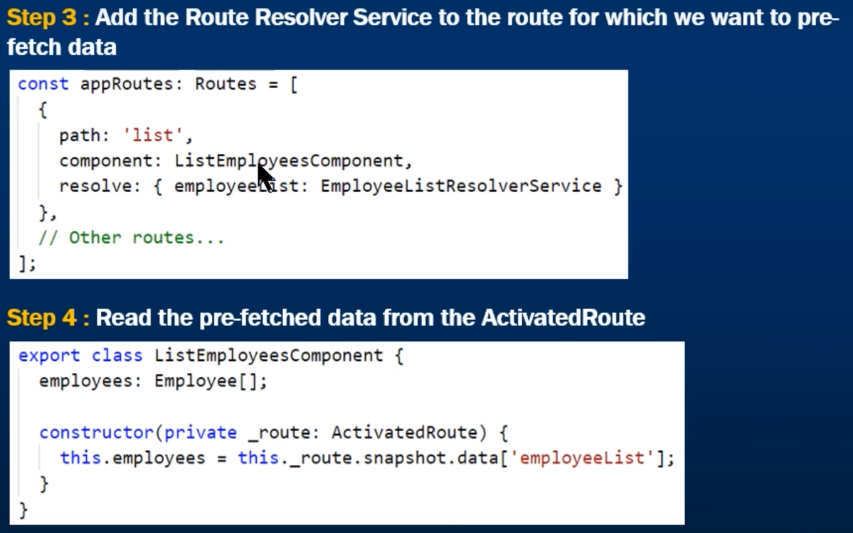


CanDeactivate is used to check whether the use really want to keep the changes are not or else whether he really want to move to the other component without making any savings in the current component

***Resolvers:***

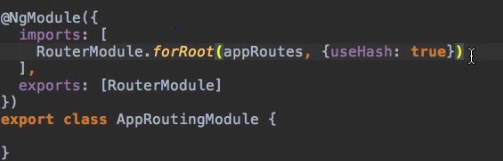
We use resolvers to collect the data before the component actually load or initialized so that we can have necessary operations to make the data available for our component.



****

UserHash:

Which will be appended in the RouterModule.forRoot as the second parameter which will used to handled the 404 error which some servers are unable to handle so that by keeping true we make sure it handles this error.



***Observables***

Subscribe has three arguments

* Handle Data
* Handle Error
* Handle Completion

We can also use Subject from Rxjs which is used to communicate between two components where we can use subscribe to get the values from the server using subject.

1. Create a service with a variable of new Subject ();

2. Use the .next () method which will be available to the variable subject (), it acts like a emit so that you can use it in the function and whenever that function emits a value you can listen to this any component.

3. Inside the constructor using the service we create inject it in the component where we required using this. Then subscribe to this variable because we know what we are sending and inside the observable we will communicate with the component local variable i.e., passing data to the them.

***Note: You can also use the observables as making a variable as OBSERVABLE<T>***

***and then subscribing it anywhere.***

***Rxjs***

It is a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code.

***Ways to Communicate***

<https://www.udemy.com/the-complete-guide-to-angular-2/learn/v4/questions/1711500> link helps to get all the ways to communicate

***Pipes***

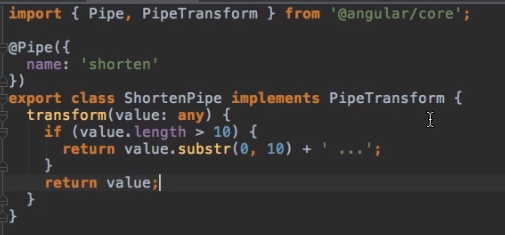
A pipes takes an input value and transforms it to a desired output.

[***https://angular.io/api?type=pipe***](https://angular.io/api?type=pipe)

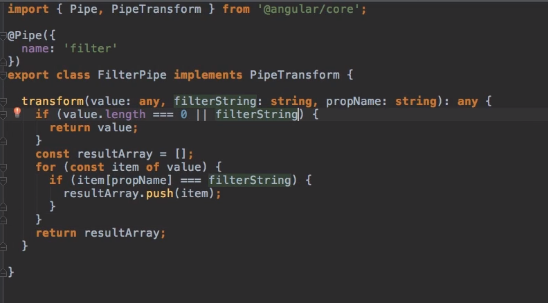
Here you can find all the built in types of PIPES, some of them are:

Currency Pipe, Decimal Pipe, LowerCase, UpperCase, Slice , PercentPipe, AsyncPipe etcc

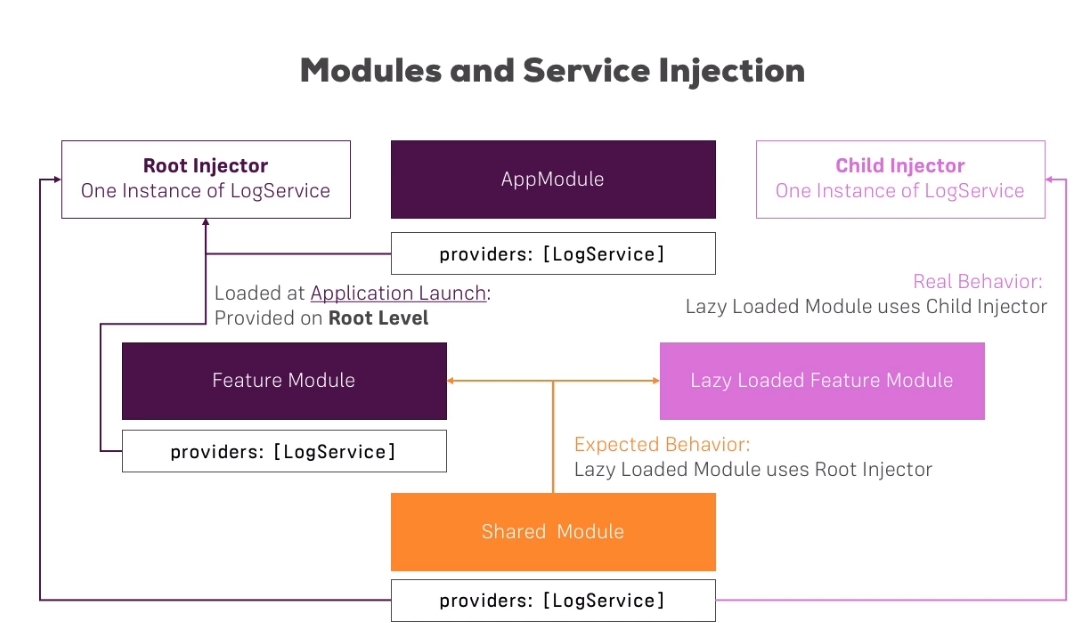
We also have Custom pipes which is shown below, You need to write a class for that which can be used to transform the the value. Here we are using it to make the string short up to 10 characters.



We can also create some complicated pipes which can be used on for loop’s , as pipes is something which generally takes the input and outputs to the end user. Here is some example which is used on for loops..

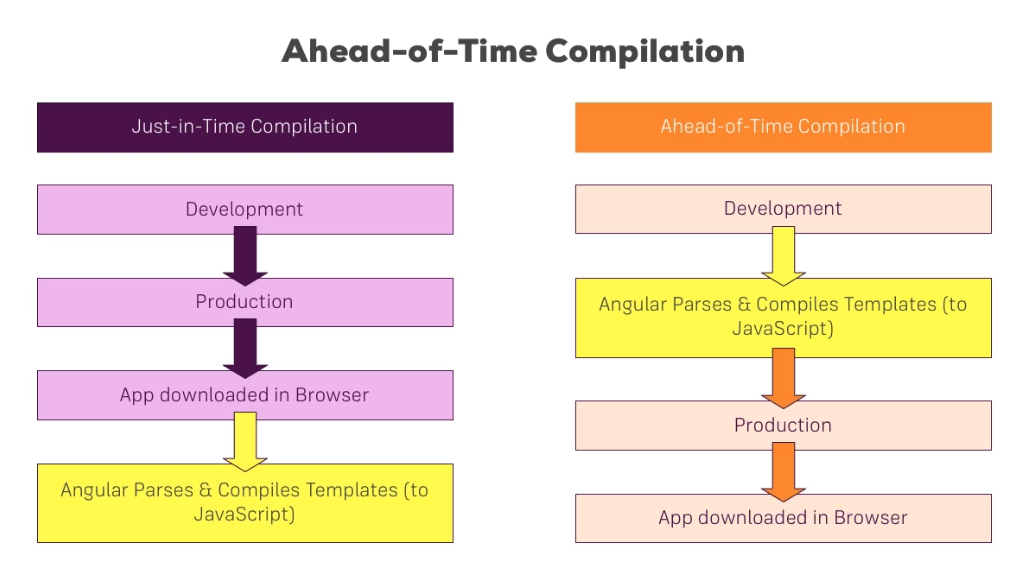


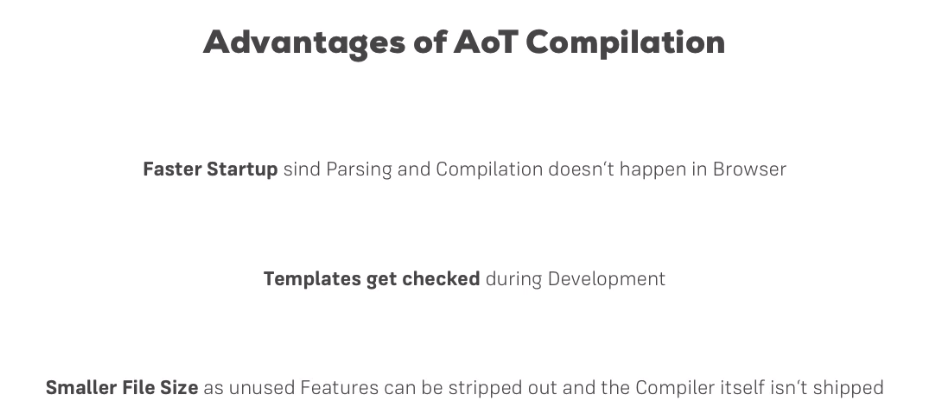
***Lazy Loading / Featured Modules / Modules and Service Injection***



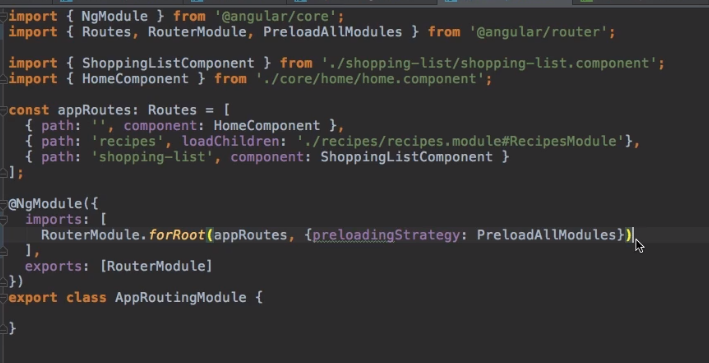
**Never ever ever use providers in Shared Module**

***Just-in-Time & Ahead-of-Time Compilation:***





***Preloading Lazy Loaded Routes***



By using the second parameter for out App.forRoot method we can easily preload the Lazy Modules after our AppModule has been completely loaded, which helps the user to actually see the fast performance even when he access the Lazy Modules as it would be already preloaded and readily available to the browser…

***Form Builder***

***FormBuilder creates reactive form with minimum code*** using **FormGroup**, **FormControl** and **FormArray**. **FormBuilder** has following methods.   
***group():*** Creates FormGroup.   
***control():*** Creates FormControl.   
***array():*** Creates FormArray.

For Example:

***this.teamForm = this.formBuilder.group({***

***teamName: ['', Validators.required ],***

***teamManager: '',***

***teamDept: this.formBuilder.group(new Department()),***

***employees: this.formBuilder.array([])***

***});***

***What is the difference between Angular Js and Angular 2+ ?***

The Major difference between AngularJs and Angular 2

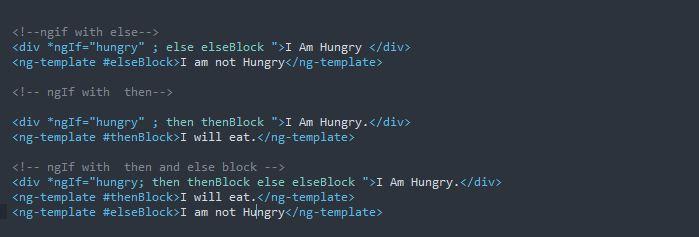
1. The Architecture of AngularJs in MVC(Model View Controller) where as the architecture of Anuglar 2 is based on service/controller.
2. Angular Js use JavaScript to build the application while from Angular v2.0 angular team introduced the typescript to write the application. TypeScript is a superset of JavaScript which helps to build robust and structured code. Dart can be used by developers along with TypeScript in version 2.0.

**DART (Data Acquisition and Reporting Tool) is a software package that gathers data and synthesizes runtime information from Biomek log files to capture each manipulation of the sample during the course of the method**

1. The Controller Concept which was present in AngularV1.0 is eliminated in Angular V2.0. Angular 2 has changed to component based UI. This helps a developer to divide the applications in terms of components with desired features and enable to call required UI.
2. With Angular v1.0 developing the search engine friendly single page applications was the major difficulty. But this bottleneck was eliminated in Angular 2

Angular 2 vs 4:

1. The upgrade of the version from 2.0 from 4.0 has reduced it’s bundled file size by 60%. The code generated is reduced and has accelerated the application development. Here the developed code can be used for prod mode and debug.
2. Router ParamMap
3. Animation Package : Up until AngularJS the code required for the animation part was always included in the application in spite of the fact that animation in actually used or not. In Angular 4, the animation in part of a separate package. This has eliminated unnecessary bundles with large sized files.
4. ngIf with a new else statement:



***Why skipped Angular 3 ?***

  
  
The angular team already updated @angular/router with version 3.X before Angular 3 would have been released that's because of major development on router packages, like router-preload. So to avoid such confusion between Angular packages with angular version. They decided to skip the Angular version 3 and directly released angular 4. So that dependency in the MonoRepo will be right on track.

***Angular Security:***

1. ***Prevent Cross-Site-Scripting:***

*Sanitization and Security Contexts:*

When a value is inserted into the DOM from a template, via property, attribute, style, class binding, or interpolation, angular sanitizes and escapes un trusted values.’

***export enum SecurityContext { NONE, HTML, STYLE, SCRIPT, URL, RESOURCE\_URL }***

***export abstract class DomSanitizer implements Sanitizer {***

***abstract sanitize(context: SecurityContext, value: SafeValue|string|null): string|null;***

***abstract bypassSecurityTrustHtml(value: string): SafeHtml;***

***abstract bypassSecurityTrustStyle(value: string): SafeStyle;***

***abstract bypassSecurityTrustScript(value: string): SafeScript;***

***abstract bypassSecurityTrustUrl(value: string): SafeUrl;***

***abstract bypassSecurityTrustResourceUrl(value: string): SafeResourceUrl;***

***}***

There are 2 kinds of patterns here, which are sanitizing methods and bypassSecurityTrustX.

***THE SANITIZE METHOD:***

If a value is trusted for the context, this sanitize method will (in case of a SafeValue ) unwrap the contained safe value and use it directly. Otherwise, the value will be sanitized to be safe according to the security context.

We have some helper functions for sanitizing the values.

The sanitizeHtml,sanitizeStyle and sanitizeUrl…

In specific situations, it might be necessary to disable sanitization. Users can bypass security by constructing a value with one of the bypassSecurityTrustX Methods

***Codelyzer***

All enterprise applications follows a set of coding conventions and guidelines to maintain code in better way. Codelyzer is an open source tool to run and check whether the pre-defined coding guidelines has been followed or not. Codelyzer does only static code analysis for angular and typescript project.

Angular links

<https://www.greycampus.com/blog/programming/top-30-interview-questions-and-answers-on-angular-5>