Index.html is served by the sevrer - this is the page which is served as a single page

Index.html - app-root = root component which ties together the application

All the files which have "component" in there name relate to the root component

App.component.ts -> selector = app-route. Information Angular needed to replace in the index.html.

Script tags which are injected automatically (webpack)

Main.ts

platformBrowserDynamic().bootstrapModule(AppModule)

Main.ts gets passed appModule which is the root module

App module has the bootstrap array which is the component list which should be known to Angular when it starts

Main.ts gets passed appModule as an argument in appModule it lists the components which should be there on bootup and app.component gets passed in as the first thing.

App-component = root component

Nest other componentns (inside HTML)

Each template has own HTML code and own styling + own business logic

Able to reuse business logic elsewhere

App component - root component

Listed in app.module in the bootstrap array

New component selectors get added to the app-component html file = root component of the app

All app related content goes into the folder which is under the app foldre

Angular - components are just classes typescript class

Angular - instantiate it so create objects based off the blue print based in the component file.

Decorators are used to enhance classes.

Angular ships with a number of packages which group functionalities

Core

Pas JS object to the decorator to configure it

**App.moduel**

Angualr uses components to build webpages and uses modules to bundle diff pieices into packages

Only in different projects would you not use App.module and use something else.

It is a bundle of functionalties of app gives Angualr the information which features does my app use?

Bootstrap - which components should you be aware of when the appication starts?

Any new components need to be registered in the module so that Angualr knwos that it exisits and by deafult Angualr will not scan all files so need to tell it all the componentent. Register the compoentn in the NgModules

Always need template or templateURL as a paramter in the @component decorator.

Data binding = communication between typescript (business logic) and HTML

Output data from TS -> HTML = string interpoleration {{data}} or property binding [property]="data"

Any expression which resolves to a string what ever is between the {{}} needs to return a string. Could call a method which returns a string. However, a number can easily be convtered into a string.

Click button in HTML and do something in TS

Event binding - bind to a click event when ever it occurs

Two way binding = both ways of getting data into TS and getting stuff out of TS . You can bind to a HTML element

Output text to the application = string interlopation {{}}

Change some property - property binding []

Constructor - method at the point of time compoent is created by Angualr

Property binding - dynamically bind to a property on a HTML element

Event binding = () atttach that to the HTML element (click)= "methodName" or code in the quotation marks

$event - variable name which you can use when using event binding - gives us acccess to the event data #

Two way bindings

@[Component](https://angular.io/api/core/Component) is a decorator function that specifies the Angular metadata for the component.

ngOnInit() for two main reasons:

1. To perform complex initializations shortly after construction.
2. To set up the component after Angular sets the input properties.

<input [([ngModel](https://angular.io/api/forms/NgModel))]="hero.name" placeholder="name">

[(ngModel)] is Angular's two-way data binding syntax.

Here it binds the hero.name property to the HTML textbox so that data can flow *in both directions:* from the hero.name property to the textbox, and from the textbox back to the hero.name.

Directives are instructions in the DOM

Once we place the selector in

Components are a type of

Compoenents are ditectives just with a templare

Directives are components without a template

**Structual Directive**

**ngIf** directive - star at the start showing that its a structual directive chnaging the DOM Structual directive, changes the structure of the DOM

Either adds or it doesn't.

Either needs to be true

Placing the ngIf directive on a component, or element, will in fact hide or show that element based on the expression you pass it to be evaluated.

Ng-template = Directive which comes with Angualr, is used as a marker in the DOM.

ngFor - Adds a new attribute for each element in the array.

let loopingVr of buttonClickArr

You can use loopingVar in other directives and can also use string interlopation with i.

ngFor let loopingVar of Arr; let i = index. I === the index of the arra.

  \*ngFor = "let recipe of recipes; let i = index"

**Attribute Directives**

Look like HTML attribute

**[ngStyle]** - bind property on the directive - allows us to dynamically assign a style

**[ngClass]** - allows us to dynamically add/remove CSS classes. nOnly works when using property binding. Takes a JS Objective where the keys are the CSS class names and the values are the conditions whether to apply or not?

**Model** = TS File

Model is a blueprint for objects that we will create

Which can be done though defining a class

Adding Public as an instantiate means that you can access it from anywhere.

Constructor is a built in function every class class has and will be executed once you create a new instance of the class

Source Maps - allow JS code to go back to TS code

[element] = "server"

**Going from Parent -> Child**

We can bind to our own custom events

@Input() means that the property which you define in another component can be acccessed globally.

Parent component

<app-child

[message] = "message1">

</app-child>

[Message] = like property binding in that it takes in an input

message1 = the variable which contains the information which should be displayed

message1 = "coming from parent"

Child Component

That message input is what is in the [] in the parent component

TS

@Input () message: string;

HTML

<h2>Parent -> Child: {{message}}</h2>

That is where we are using the message which is getting passed from the parent

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Going from Child -> Parent**

Parent Component Yhe

HTML

<app-parent

(messageEvent)="receiveMessage($event)">

</app-parent>

message:string;

receiveMessage($event) {

this.message = $event

}

(messageEvent) - Event Binding Syntax binding to the event we emitted from the child

receiveMessage - function call which takes in the $event argument

This.message = what we defined in the child (Hola!)

Child Component

**HTML**

<button (click) = "sendMessage()"> Send Message </button>

**TS**

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

sendMessage() {

this.messageEvent.emit(this.message)

}

message: string = "Hola Mundo!"

sendMessage- function which emits a message to messageEvent

messageEvent - The thing which gets sent using Output

EventEmitter<string>() - Defiing the type of thing which we are emititing

Event emitter is an object in the Angualr framework which allows you to emit your own events

Emit event

Event Emitter is a event emiiter which allows you to emmit your own events

**Consuming Component**

Needs to require in the componenent in the HTML and bind to the event which is being outputted on the broadcasting component

**Broadcasting Component**

Needs to broadcast an event using the Output command

@Output ()blueprintCreated =new EventEmitter <{serverName:string, serverContent:string}>();

We cabn emit our own events

@input - bindable from outside

@output - Get the parent compoennt listening to the events emmited from the outside

App component CSS file = paragraph meaning only applied to the app component as Angualr adds in attributes for each element < p\_ejo-2> <p/p>

Adding style encapsulation for the components

Shadow DOM - each element has its own DOM

@Component ({

encapsulation : ViewEncapsulation.

})

Emulated - Default

None - Does not but on the attributes on the elements and the CS selectors do not get alterd at run time

Native - Shadow DOM technology - Same as emulatd but only in browsewrs which support it.

Normally choose Enulated.

Local references buy using the # keyword in the HTML template.

This will only be used in the HTML and will not be exposed to the TS Code.

<input

type="text"

class="form-control"

#serverNameInput>

Access #serverNameInput in the HTML or pass to TS as an argument for a function.

Or use

n

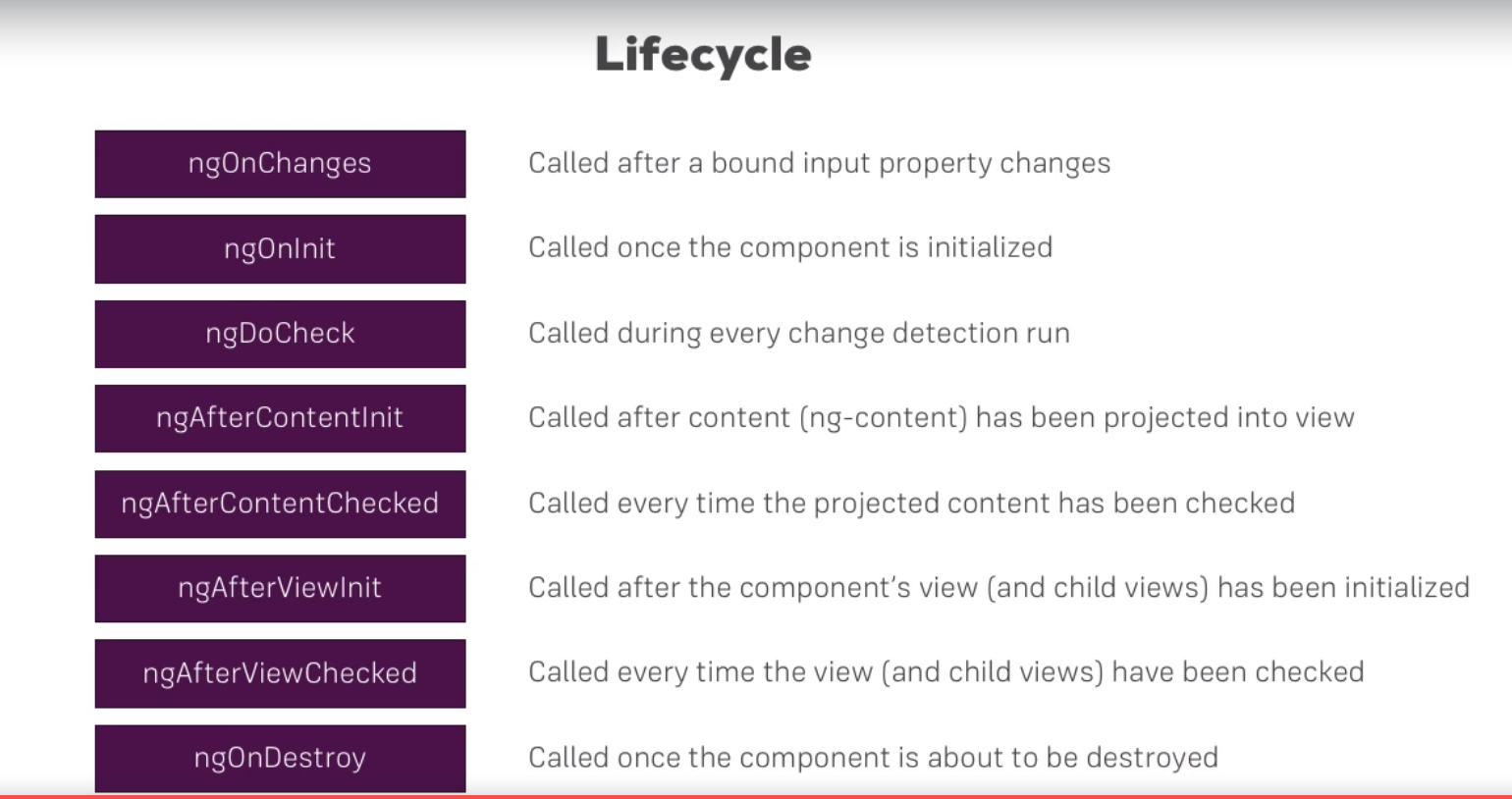
And use ".nativeElement.value" when accessing the value inutted as that is what is in the DOM.

Abstracting out complexity from components

Pass complex HTML into a component from outside

Everything you put in-between your opening and closing tags for own components are lost. However, using ng-content is a way to inject the content which you have defined in the component where you are using the child component. Ng-content is a hook as it pulls in HTML into template. Could use property binding but that might get complex.

ngOnInit - Lifecycle hook



Everytime the component is created (server is createed) constuctor and NgOnInit gets called as they are hooks

**ngOnChanges**

Using the ngOnChanges means that you have to use it in the implements section of the code to speficiy which interfaces you are using ect and import it from @Angualr Core

ngOnChanges only hook which takes in an argument (changes: SimpleChanges type)

SimpleChanges - import from @AngularCore

Element = bound property and gives current value +

 Whenever parent component changes the value of properties used in child component decorated with @Input() then the method ngOnChanges() created in child component runs automatically.

This hook would be useful if you wanted to react to any changes and do something with the old value ect.

**DoCheck**

Method which is ran on every change detection run

When ever Angular checks for any chnages doCHeck gets called

AfterContentInit

A lifecycle hook that is called after Angular has fully initialized all content of a directive. Define an ngAfterContentInit() method to handle any additional initialization tasks.

Content is 2retrived through ng-content#

AfterViewInit()

Called when the view hasinaltised and we can access the templates elements

Before its reached - you cannot do that - as the DOM hasn't been rendered yet

***ngOnDestory()***

***Called before you rmeove something from DOM***

**@ContentChild**('contentParagraph') paragraph:ElementRef

Access content stored in another component but is passed on by ng-content as a local reference on a template which

Recipe item -> recpie -> recipe detail

*Cannot have more than one structual directive on an element*

**Attribute Directive**

Directive does not have a template and cannot have one.

this.elememtRef.nativeElement.style.backgroundColor - accessing elements like this is not a good idea as Angular can render templates with no DOM which would mean that the attributes would not be available.

Use the Renderer to access the DOM.

Renderer

The Renderer2 class is an abstraction provided by Angular in the form of a service that allows to manipulate elements of your app without having to touch the DOM directly.

This is the recommended approach because it then makes it easier to develop apps that can be rendered in environments that don’t have DOM access, like on the server, in a web worker or on native mobile.

Putting host listener means that it will listen to wherever this is hosted.

And you can access the DOM @HostListener('mouseenter') - event supoported by the DOM this element sits on. All the events you have with event binding. So it is listening on the host which it is placed upon.

@HostBinding() - pass a string defining to which property of the hosting element we want to bind. Style.backgroundColor so @HostBinding('style.backdground.color) variableName: string - where variableName is the placeholder for the style.backgroundcolor.

When you attatch a directive onto a property these are evaluated first and then it will use property binding.

These are used on the HTML like:

<li class="dropdown" appDropdown>

Where appDropDown is the directive.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

export *class* BetterHighlightDirective implements *OnInit* {

@Input () defaultColor: *string* = 'transparent';

@Input () highlightColor: *string* = 'blue';

// binding to the background colour

@HostBinding('style.backgroundColor') backgroundColor: *string*;

*constructor*(private *elref*: ElementRef, private *renderer*: Renderer2) { }

ngOnInit() {

this.backgroundColor = this.defaultColor;

// this.renderer.setStyle(this.elref.nativeElement, 'background-color', 'blue');

}

// Putting host listener means that it will listen to wherever this is hosted.

@HostListener('mouseenter') mouseOver(*eventData*: Event) {

this.backgroundColor = this.highlightColor;

// this.renderer.setStyle(this.elref.nativeElement, 'background-color', 'blue');

}

HTML

<p appBetterHighlight defaultColor="yellow" highlightColor="red" >Style me with basic directive </p>

**Structural Directives**

Structural directives with the \* are used by Angualr which transforms them into something else. Which is <ng-template [ngIf] = "true/false"> using property binding

To make our own we can strutural directive by (same as NGIF)

1. Creating a new directive
2. Injecting TemplateRef and ViewContainerRef (into the constructor)
3. Creating an input which which accepets in a setter method (which means the method gets called everytime the value changes)
4. On this setter method set the viewContainer to createEmbeddedView with the template ref as an argument
5. Else set the viewContainer to .clear()

Ng switch

When you want to display a certain attribute based on a condition

<div [ngSwitch] = "value">

<p \*ngSwitchCase = "5">Value is 5</p>

<p \*ngSwitchCase = "10">Value is 10</p>

<p \*ngSwitchCase = "100">Value is 100</p>

<p \*ngSwitchCaseDefault >Value is Default</p>

</div>

@Directive({

selector: '[appUnless]'

})

export *class* UnlessDirective {

@Input() *set* appUnless(*condition*: *boolean*) {

if (!condition) {

this.vcRef.createEmbeddedView(this.templateRef);

} else {

this.vcRef.clear();

}

}

*constructor*(private *templateRef*: TemplateRef<*any*>, private *vcRef*: ViewContainerRef) { }

}

HTML=

<div \*appUnless="onlyOdd">

Using the \* as this is a structual directive.

**Services**

Another pice in App, central repo/business unit/ store centralise code in.

Another class

No @ decorator.

Do not import the service normally and initialise manually as that is incorrect. Get Angualr to do this through using the constructor on the component which is needing to use the service.

*constructor* (private *loggingService*:LoggingService){}

Stay within the Angualr eco syrtem

To specify array type you write

accounts: {name:*string*, status:*string*}[] = []

Angualr Dependency Injector = hireactical injector

Provide a service in one component - Angualr creates an instance of this sservice for this component and all its child componentn. Will recieve the SAME instance of this service

App Module - Provide a service here in the Providers array the same instance of the class in whole application in other directives/services

App Component - all componentns but not other services. Do not progate up.

Single Component - Same service on this level. This would override if you provided service on a higher level.

The providers array specifies if the component is implementing this service alone or is coming from anywhere else.

Do not remove from the child component construsctor

To inject a service into something, that something has to have some metadata around it .

Component has meta data which is the @Component.

Directve has meta data as that has @Directive

Service has no meta data.

Using a service in a service, use @Injectable to inject the servive. Something can be injected into this service. Gets injecte into the reciving service.

Emiiting an event from the service.

Setting the eventEmmitter in the service and emitting the event in the componnt which we want to emit it in

When you are setting a service to be addded into another service it needs to be injected into the Provideres array in the app module.

**Routing**

Declare in the app module

Path- entered in URL after domain

*const* appRoutes: Routes = [

{ path: '', component: HomeComponent},

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

{ path: 'servers', component: ServerComponent}

]

Means that localhost:4200/users is the route which is allowed.

Also need to add RouteModule.forRoot(variable you have defined in the app module file) in the imports array:

imports: [

BrowserModule,

FormsModule,

HttpModule,

RouterModule.forRoot(appRoutes)

],

And adding this directive into the app-component:

<router-outlet></router-outlet>

Using <a href = “/users”>Users</a>

Means that the app gets refreshed for every refresh and is not a good implemtation

You should use the following?:

<li role="presentation"><a routerLink ="/servers">Servers</a></li>

<li role="presentation"><a [routerLink] ="'/users'">Users</a></li>

Where we can either add the property of routerLink onto the attribute and give it the direct path or use property binding [routerLink] and pass it the String of the path, we can also pass an array into the property binding and use the elements in the array to build up the URL so using the property binding is a good idea.

You need to pass in the / when specifying routes as if you don’t then you get relative paths being generated.

Adding <a routerLink= “servers”></a> into the users component will navigate to “localhost:4200/users/servers”. If users was the base domain you were on

You can use relative paths when at root level, when you are in the router component.

/servers = Absoulte paths always get appended to the route domain

Servers = Relative paths = added to the currently loaded path

../../servers = relative go back two paths and append servers to the URL.

<li role="presentation"

routerLinkActive ="active"

[routerLinkActiveOptions] ="{exact: true}"

>

routerLinkActive – You can give a css class when the routerLink becomes active

For routes which are just / - these will always be active as the **routerLinkActive** looks if the route is correct. To match the exact path, we need to add on **routerLinkActiveOptions** = {exact: true}.

**Triggering Route Programmatically**

Inject the Router class into the component and call:

this.router.navigate(['/servers'])

Where the [‘/servers’] – is the route defined. This is an absolute path i.e will naviate from the base URL to whats in the string.

Relative Path

Unlike the router link the router.navigate does not know where it is/ which route your currently on, to tell the router.navigate where we currently are we need to pass in an object

import { Router, ActivatedRoute } from '@angular/router';

private *route*: ActivatedRoute

this.router.navigate(['servers'], {relativeTo : this.route })

We need to import the ActivatedRoute from router and set it to a local variable. This will nvaite

**Adding Parameters to Routes**

This is for using URL Params

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

Observables

Feature added by a third party package heavily used by Angular to use async task, parameters of currently loaded route, might change

Observable – subscribe to code but not have to wait for it now.

*constructor*(private *route*: ActivatedRoute) { }

ngOnInit() {

this.user = {

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

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

}

this.route.params.subscribe(

(*params*:Params)*=>*{

this.user.id = params['id'];

this.user.name = params['name'];

}

)

}

Here we are setting the route up to look in the URL and get the params and use it in the code.

We are also setting up the subscription to the observable in the ngOnInit too so when the params change (from initiating something in the component) then the callback gets fired.

If you know that the component will only be loaded from outside the component (so it will get iniatlised) then you do not need to subscribe to the observable as there is no way for it to get updated.

Angular cleans up subscription when component is destroyed otherwise we would have to unsubscribe from the observable.

<a

[routerLink] ="['/servers',5, 'edit']"

[queryParams] = "{allowEdit: '1'}"

[fragment] ="'loading'"

href="#"

class="list-group-item"

\*ngFor="let server of servers">

{{ server.name }}

</a>

Above example of loading query params and fragement harcoded ^ and dynamically Below.

onLoadServers(*param*:*number*){

this.router.navigate(['/servers', param, 'edit'], {queryParams:{allowEdit:param}, fragment: 'loading' })

}

*console*.log(this.route.snapshot.queryParams);

To retrieve the parameters you set in the URL you can do this on ngOnIt accessing the activatedRoute.snapshot.queryParams

You can also subscribe to the observerable on the activatedRoute so

this.route.fragment.subscribe(()*=>*{

})

To load routes within routes you specify this in the app module like:

{ path: 'servers', component: ServersComponent, children: [

{ path: ':id', component: ServerComponent},

And in the toplevel component you specify a

<router-outlet></router-outlet>

As that is where the component is going to be renderd from.

Using this pattern is the reason you need to subscribe to the events which clicking the buttons throws off as we are not re-loading the component just sending it new params.

onEdit(){

this.router.navigate(['edit'], {relativeTo : this.route, queryParamsHandling: 'preserve' })

}

When switching from one component to another we normally loose the query params but using the queryParams: preserve, this maintains them when switching components.

**Accounting for redirection of pages**

In the app.module put this in the last thing in the routes array.

{ path:'not-found', component: PageNotFoundComponent},

{ path:'\*\*', redirectTo: '/not-found'}

When you export a module you specify what you want to be accessible from the importing module in the exports array.

**Protecting Routes**

Protect routes using canActivate Guard. – with guard

canActivate

either returns async stuff with an observavle or a promise or just a Boolean (so synchrnossly)

This protects the top level route and all of its children.

canActivateChild(*route*: ActivatedRouteSnapshot, *state*: RouterStateSnapshot) : Observable <*boolean*> | Promise<*boolean*> | *boolean* {

return this.canActivate(route, state);

}

The can activate child route when implemented protects the child routes only. We are returning the can activate method but due to implementing this under the interface of canActivateChild means that it only looks at the children routes.

Export Interface = contract which gets imported by other class which forces the importing class to provide some logic.

**CanDeactivate**

This is more complex than the can canActivate or canActivateChild component as it takes in the component which it is implanted on. To use this guard, I created a new service canDeactiveGuardService and implemented the CanDeactivate inferace on that class. By defining an interface we force a class to implement a method which takes parameters of a specific type and returns a value of a specific type.

I then defined the method CanDeactivate which takes in a lot of information and returns a Boolean, this will be called by the Angualr router when we try and leave a route

export *class* canDeactivateGuard implements *CanDeactivate*<CanComponentDeactivate>{

canDeactivate(*component*:EditServerComponent,

*currentRoute*: ActivatedRouteSnapshot,

*currentState*: RouterStateSnapshot,

*nextState*: RouterStateSnapshot) : Observable<*boolean*> | Promise<*boolean*> | *boolean*

{

return component.canDeactivate();

}

}

We then set the route up so that on editserver we are using this guard

{ path: ':id/edit',component: EditServerComponent, canDeactivate: [canDeactivateGuard]}

Then in editserver we are performing some checks which return true (so the user can deactivate the component)

canDeactivate(): Observable<*boolean*> | Promise<*boolean*> | *boolean*{

*console*.log('got here 1')

// if allow edit is true then it can deactivte

if(!this.allowEdit){

return true

}

if((this.serverName !== this.server.name || this.serverStatus !== this.server.status) && !this.changesSaved){

*console*.log('got here 2')

return confirm('Do you want to discard the changes?');

}else {

return true;

}

}

**Passing data with routes**

Pass the data object which takes any key/value pair.

{ path:'not-found', component: ErrorPageComponent, data:{message: 'Page not found'}},

Here we are subscribing to the data property (incase it changes) and setting it to the local var of this.errorMessage.

ngOnInit() {

this.route.data.subscribe((*data*:Data)*=>*{this.errorMessage = data['message']})

}

The reason we do this is because even though we are hardcoding the data in the route, the errorPageComponent might be re-used later on with different config.

**Resolver**

Being able to run some code before a component is actually initialised.

1. Set up the resolver service

export *class* ServerResolver implements *Resolve*<Server>{

*constructor*(private *serverService*: ServersService) {}

resolve(*route*: ActivatedRouteSnapshot, *state*: RouterStateSnapshot): Observable<Server> | Promise <Server> | Server {

return this.serverService.getServer(parseInt(route.params['id']))

}

}

In the <> is what the service actually resolves in the end so it will give us back an object with ID, Name and Status – as that is what the sever interface is.

1. Set the routing to use the resolver and bind to a property
2. { path: ':id', component: ServerComponent, resolve: {server: ServerResolver}},

In this case were binding to the sever name, this gets put in the Activated Route Data bucket which we access using an observerable

3. In the component which we are using the component (Server Component) we need to subscribe to the server property which is found under the data observable.

ngOnInit() {

this.route.data.subscribe((*data*: Data)*=>*{

this.server = data['server'];

})

When we are serving content from a web sever, as Angualr is a SPA so will always serve index.html However, when using a web sever, and going to the route /servers/id – this would return a 404 as it doesn’t technically exisit from an Angular perpective. To enable dynamic routing, you would have to put the following config in app.module:

RouterModule.forRoot(appRoutes, {useHash: true })

Which will tell the server to only care about the things after the hash.

{path: '', redirectTo: '/recipes',pathMatch: 'full' },

pathMatch: 'full' means, that the whole URL path needs to match and is consumed by the route matching algorithm.

pathMatch: 'prefix' means, the first route where the path matches the start of the URL is chosen, but then the route matching algorithm is continuing searching for matching child routes where the rest of the URL matches.

**Observable**

Observable = data source

Object we import from 3rd party = rxjs

Observale and an Observer and in between we have a stream/timeline, can have multiple events emitted by the observable depending on the dat source of the observable.

Observer = subscribe function.

Handle Data, Handle error/ handle completion – your code gets executed in that block.

Some observables never complete such as the button but the HTTP response will complete.

Async will close

Normal application code

Obserbanle handle async – trigger HTTP/ user events as you don’t know how long they take/ when they will happen, you don’t want to wait for the programme as this would be a blocking code/ block logic.

Using callbacks/promises are another way of handling async.

Angualr does not re-render the whole component when the paramer of an observable changes. It uses obserables to react to the change.

**routerLink -> URL Param Change -> Subscription to an Observable**

Angualr sends the data- it regognises that we clicked a route

ngOnInit() {

this.route.params

.subscribe(

(*params*: Params) *=>* {

this.id = +params['id'];

}, () *=>*{

// error

},() *=>*{

// complete

}

);

}

Subscription to an observable – you can have three call backs, the expected response, the error and the completion of an observable.

import { Observable } from 'rxjs/Observable';

import 'rxjs/rx';

*const* myNumbers = Observable.interval(1000);

Setting up observable (will not fail nor complete).

Data source is a utility function avaibale in the RXJS package.

*const* myObservable = Observable.create((*observer*: Observer<*string*>)*=>*{

setTimeout(()*=>*{

observer.next('first package')

}, 2000)

setTimeout(()*=>*{

observer.next('second package')

}, 4000)

setTimeout(()*=>*{

observer.error('Error')

}, 5000)

// takes fucntion as argument.

//

})

Creating an observable, next is emits a data package, pushes the next data package.

Error – this is called when the erro

myObservable.subscribe((*data*: *string*)*=>*{

*console*.log(data)

}, (*error*: *string*)*=>*{

*console*.log(error);

}, ()*=>*{

*console*.log('completed')

})

This is what you get when you subscribe to an observable.

*const* myObservable = Observable.create((*observer*: Observer<*string*>)*=>*{

setTimeout(()*=>*{

observer.next('first package')

}, 2000)

setTimeout(()*=>*{

observer.next('second package')

}, 4000)

setTimeout(()*=>*{

// observer.error('Error')

observer.complete()

}, 5000)

setTimeout(()*=>*{

// observer.error('Error')

observer.next('third package')

}, 6000)

// takes fucntion as argument.

//

We never see third package as we have completed the third package.

If you do not complete your observables and keep them going in the background then you are creating a memory leak, as you have an active subscription which you don’t care about.

To prevent this, declare a variable in the global scope which is of type Subscription and subscribe to the variable

this.customObservableSubsciption = myObservable.subscribe((*data*: *string*)*=>*{

*console*.log(data)

}, (*error*: *string*)*=>*{

*console*.log(error);

}, ()*=>*{

*console*.log('completed')

})

Then implement the lifecycle hook of OnDestory and unsubscribe to it

ngOnDestroy(){

this.customObservableSubsciption.unsubscribe();

this.numbersObservableSubsciption.unsubscribe()

}

Subject- like an observable and allows you to conviently to push it to emit new code.

You can both push data and subscribe to a subject so when you want to do cro

Nextpushes new value – complete or error

Subscribe consumes it

Like event emitter -

Don’t forget to unsubscribe

export *class* UsersService {

userActivated = new Subject();

}

onActivaten() {

// pushing data package with ID

this.usersService.userActivated.next(this.id);

}

onActivatee() {

// pushing data package with ID

this.usersService.userActivated.error(this.id);

}

onActivatec() {

// pushing data package with ID

// this.usersService.userActivated.complete(this.id);

this.usersService.userActivated.complete();

}

this.usersService.userActivated.subscribe(

(*id*: *number*) *=>* {

if (id === 1) {

this.user1Activated = true;

} else if (id === 2) {

this.user2Activated = true;

}

}, ()*=>*{

*console*.log("error!")

}, ()*=>*{

*console*.log("complete!")

}

);

Map – maps the data you get back into a new observable with any transformation of your choice.

**Forms** – Angular gives a JS object view of the form data.

Template Driven

Template driven approach – set up form in template in HTML and Angular will infer the inputs.

Reactive

You define the structure of the form in TS and you set up the HTML and then do a manual connection.

FormsModule imported into app.module

Once Angualr detects the <form> element in HTML it will create a JS representation of the form. – Dirtective to create the object.

Still need to register controls manually – tell Angular, within that form element what should be a control of my form?

Template driven – pick the input you want and attatch the <name> =”name” and the ngModel.

If you put a button in a HTML form element, the default behaviour of HTML will happen. The Default behaviour of HTML will override.

On the <form> tag use

<form (ngSubmit)="onSubmit()">

Which will capture and override the default behaviour.

To get access to the value which are stored on the form you have to make a local reference to the form:

<form (ngSubmit)="onSubmit(f)" #f="ngForm">

import { NgForm } from '@angular/forms';

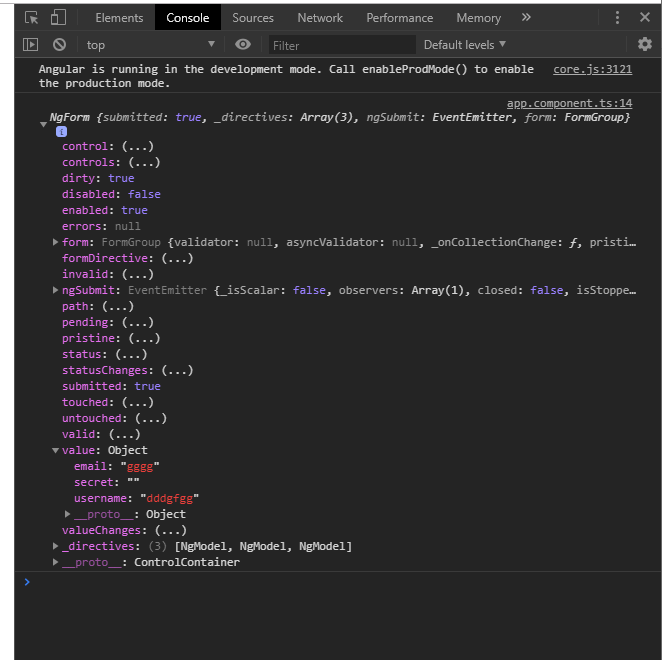
and

onSubmit(*form*:NgForm){

*console*.log(form)

}

This logs out all information about the form at time you clicked submit.



Under value is where we have kept our values

Dirty – has the form been used?.

Another way we can get the form value is by using view child and referncing the local ref you have set up, giving it the type of NgForm.

@ViewChild('f') signUpForm: NgForm;

Angular tracks the form state and adds or removes classes accordingly:

form-control ng-dirty ng-touched ng-invalid

<form (ngSubmit)="onSubmit(f)" #f="ngForm">

@ViewChild('f') signUpForm: NgForm;

this.signUpForm.setValue({

userData:{

username: suggestedName,

email: ''

},

secret: 'pet',

questionAnswer: '',

gender: 'male'

})

Sets a value in the form - not the best approach as it overrides stuff

To stop the overriding the whole form use the patch value

this.signUpForm.form.patchValue({userData:{

username: suggestedName

}});

Disable the whole form:

[disabled] = "!f.valid"

Property binding on disabled, checking if the overall form is valid

@ViewChild('f') signUpForm: NgForm;

this.signUpForm.reset();

This clears the form and resets the state of the form

**Template**

Angualr infers form from DOM

**Reactive**

Form created programmatically and synchronized with the DOM

Create in typescript and

When rendering the form, ensure to initialize it before the view is rendered, so on ngInIt.

Vaidation on form

ngOnInit(){

this.signupForm = new FormGroup({

'username' : new FormControl(null, Validators.required),

'email': new FormControl(null, [Validators.required, Validators.email]),

'gender': new FormControl('male')

});

}

Get helper method in the view.

\*ngIf="!signupForm.get('username').valid && signupForm.get('username').touched"

Validator is a function which get executed automarically when it checks the validity of the form control. This gets checked when you change the control.

Valifdator needs to return something for Angualr to handle the return value correctly

forbiddenNames(control: FormControl): {[s: string]: boolean }{

}

Expected return type os a JS object with a key of a string and a Boolean as a value

IsTrue: False

**Own form Validator**

'username' : new FormControl(null, [Validators.required, this.forbiddenNames.bind(this)]),

forbiddenNames(control: FormControl): {[s: string]: boolean }{

if(this.forbiddenUserNames.indexOf(control.value) !== -1){

return {'nameIsForbidden': true};

}

return null

}

This function is saying if the return value of indexOf is not -1 then return the first return type. If the return value is minus one (so its not found) then return null.

**Async Validators**

Used when calling out to a websever for validation

'email': new FormControl(null, [Validators.required, Validators.email], this.forbiddenEmails)}),

Adding Async into the FormControl

forbiddenEmails (control: FormControl): Promise <any> | Observable <any> {

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

setTimeout(()=>{

if(control.value === 'test@test.com') {

resolve ({'emailIsForbidden': true})

}else{

resolve(null);

}

}, 1500);

});

return promise

}

**Reactive approach**

Accessing values on a form

Declaring form value

recipeForm:FormGroup;

Creating values in the form

let receipeName = '';

let imageURL ='';

let receipeDescription='';

let recipeIngreidents = new FormArray([])

this.recipeForm = new FormGroup({

'name': new FormControl(receipeName, Validators.required),

'imagePath' : new FormControl(imageURL,Validators.required),

'description': new FormControl(receipeDescription,Validators.required),

'Ingredients' : recipeIngreidents

})

Getting information from the form:

console.log(this.recipeForm);

Method to remove element at the index

(<FormArray>this.recipeForm.get('Ingredients')).removeAt(index)

As of **Angular 8**, there's a new way of **clearing all items**in a FormArray.

1. (<FormArray>this.recipeForm.get('ingredients')).clear();

The clear() method automatically loops through all registered FormControls (or FormGroups) in the FormArray and removes them.

It's like manually creating a loop and calling removeAt() for every item.

**Pipes**

Used to transform output

Username = ‘Max’ (string)

Output into a template = string interperlation {{username}}

All upper case when outputting it

<p> {{ username | uppercase }} </p>

MAX – would be the output

Pipes transforming values.

Only place you use a Pipe is the template.

Give a pipe a parameter by giving it **:**

Pipes get applied left to right,

{{ server.started | date: 'fullDate' | uppercase }}

So date gets applied to server started, then uppercase gets applied to date.

**Creating your own pipe**

Creating your own pipe in the app folder

Create new pipe:

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

@Pipe({

name: 'shorten'

})

export class ShortenPipe implements PipeTransform{

transform(value:any){

if(value.length > 10){

return value.substr(0,10) + '...'

}

return value

}

}

And put the class name in

declarations: [

AppComponent,

ShortenPipe

],

When running a pipe, it does not run when the data is changed. This is to enhance performance in the Angular application.

@Pipe({

name: 'filter',

pure: false

})

Re-calculate the pipe given the data changes, if you set Pure to false, there will be a performance hit.

ASync pipe for data which is going to come later

{{appStatus| async }}

**Angular and Databases**

Do not connect DB creds into Angular Application as everyone can inspect the code there. Cannot store creds in there.

Sends HTTP response to a server. REST/GraphQL API. Use server side languages such as Node/PHP to communicate with API’s.

Uploding files/Analytics and back end server interaction.

**URL** – end point.

**HTTP Verb** – what do you want to do with the API?

**Headers** – Optional

**Body –**Core data, POST/PUT PATCH.

Firebase is a complete backend server

Requests are only sent when we subscribe to the method.

When doing a HTTP POST, the browser sends 2 POSTS, one for the options to check that it is allowed to do the post, the other is the actual data.

Pipe – do transformation on observable and return observable

Map function takes another function which gets response and return transformed response data.

.pipe(map(response =>{

const postsArr = []

for(const key in response){

if(response.hasOwnProperty(key)){

postsArr.push({ ...response[key], id: key} )

}

}

return postsArr;

}))

Observable methods

On HTTP methods you can set the return type of the data you are getting back from HTTP.

@Injectable({providedIn: 'root'})

This is a way of making the service which you are using available at root so it is the same as importing it into app.module providers array.

If you want to catch errors and log them you can use:

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

import { Subject, throwError } from 'rxjs';

catchError(errorRes =>{

//send to analytics server behind the scenes, Log it etc.

console.log(`coming from the posts service ${errorRes.message}`)

return throwError(errorRes);

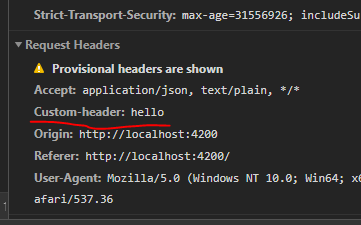
if you want to do some backend parsing or logging of the error.

Adding headers, set a config object and then set some headers

this.http.get<{[key:string] : Post}>('https://test-project-b8fa5.firebaseio.com/posts.json', {

headers: new HttpHeaders({"Custom-header": "hello"})

})



Query params

let searchParams = new HttpParams()

searchParams= searchParams.append('print', 'pretty')

searchParams= searchParams.append('mark', 'nelson')

return this.http.get<{[key:string] : Post}>('https://test-project-b8fa5.firebaseio.com/posts.json', {

headers: new HttpHeaders({"Custom-header": "hello"}),

params: searchParams

})

deletePosts(){

return this.http.delete('https://test-project-b8fa5.firebaseio.com/posts.json',

{

observe: 'events'

}).pipe(tap(event=>{

console.log(event)

if(event.type === HttpEventType.Sent){

console.log(the event has been sent)

}

if(event.type === HttpEventType.Response){

console.log(we have got a respone from the sever);

}

}))

}

Here we are tapping into the event that comes back which is an RXJS method allowing you to tap into what you have. You can use the HTTPEventType enum to find out what type of event is getting sent back.

Interceptors are where you need to set something on every HTTP request. Ie headers etc.

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

export class AuthInterceptorService implements HttpInterceptor{

intercept(req:HttpRequest<any>, next: HttpHandler){

const modifiedRequest = req.clone({headers: req.headers.append('Auth', 'xhdh')})

console.log(`request is on its way and its ${req.body}`)

return next.handle(modifiedRequest);

}

}

Sample Interceptor

Cannot modify the actual request as the object is immutable

Add the following to the app.module

@NgModule({

declarations: [AppComponent],

imports: [BrowserModule, FormsModule, HttpClientModule],

providers: [{

provide: HTTP\_INTERCEPTORS,

useClass:AuthInterceptorService,

multi:true

},

{

provide: HTTP\_INTERCEPTORS,

useClass:LoggingInterceptor,

multi:true

}

],

bootstrap: [AppComponent]

})

The order of the objects which are in the array matters, the one which is first will be executrd first

**Authentication**

Sessions aren’t used as it’s a single paged application so no sessions needed

Client sends username and password

Server sends back JWT which is encoded not encrypted with session algorithm

Can be unpacked and read by the client

This is stored in the local storage of the browser and every request the client does, it sends with the token which only the server knows. Atttach as a query param or a URL param. The token is validated with a private key.

