## Angular

## Getting Started

Practice:

<https://jcoop.io/angular-practice-exercises/>

Git hub:

<https://github.com/jmcooper/angular-fundamentals-files>

### Angular JS vs Angular

**Angular js**

* MVC Framework

**Angular**

### Start with angular

Install angular CLI

$ sudo npm install –g @angular/cli@7.1.2

Create a project using CLI

$ ng new ng-fundamentals

You may can copy existing project package.json and package-lock.json contents.

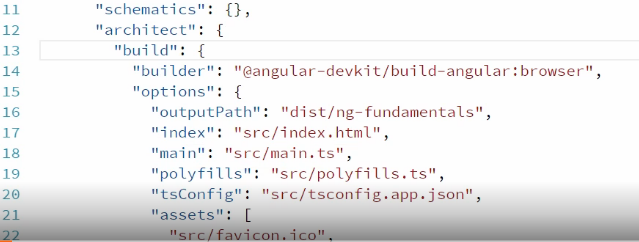
Delete the “node\_modules” and then npm install

$ npm install

To start

$ npm start

If we access the app (default port 4200), it will launch the file **src/main.ts** This mapping can be found under package.json **build** mapping section.



The **main.ts** file will be loaded by web pack i.e. **angular.json** file.

Then the main.ts file will load the **app.module** and that makes angular aware about our app component, which will load the index.html

## Creating and communicating between components

### Creating a component

We need to create a ts file as below

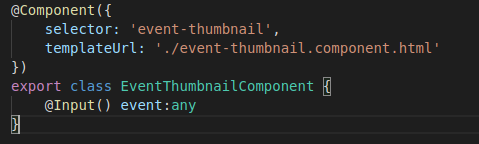


Register the component to root module file (app.module.ts) in **declarations[]**

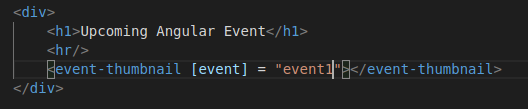
### Passing data to child component

We need to pass through **@Input()** decorator

Child component:



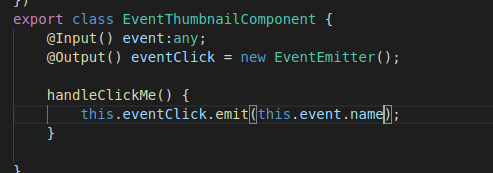
Parent component (especially template file):



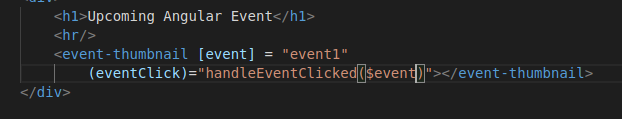
### Passing the data from child to parent

We will do that with **@Output()** decorator, especially we will pass back events generated in child component to parent component.

Capture event in child component:



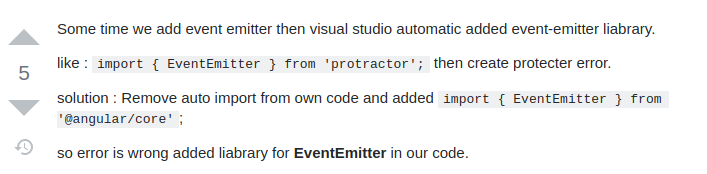
Capture the event emitted in parent template:



Define a method in parent component.ts

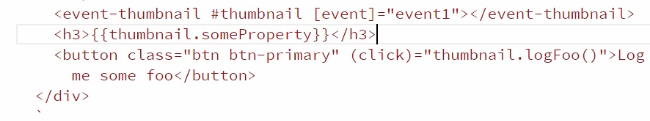


A Issue faced with **EventEmitter** import, many compilation packages were failing:



### Template Reference Variable

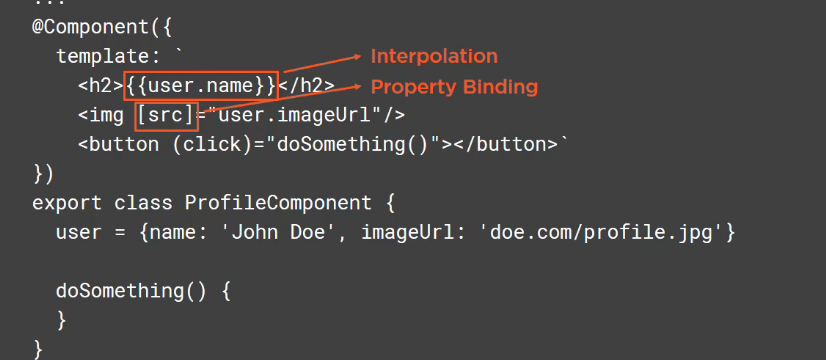
Template reference variable is a simple and straight forward way to use methods and variables of child component in parent component



In above **#thumbnail** is the template reference variable.

## Angular Template Syntax

### Interpolation and Property binding



**Interpolation** is used to bind the data from component to template.

**Property binding** is used to bind the property of a **DOM** element. (In above case src of image)

We can use expressing inside interpolation ex. {{ 2 + 2 }}, but there are restrictions as below

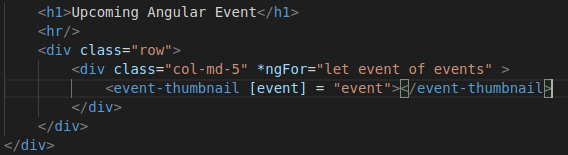
* Can’t use assignments (=, +=, ++, etc)
* Can’t use new keyword
* Can’t chain expression with ;
* Can’t use global name space as “console”

### Event binding and statements

<button (click) = “doSomething()”>Click Me</button>

### Repeating data with \*ngFor

\*ngFor is a structural directive which change the DOM. In angular structural directives starts with **\*.**

****

\*ngFor=”let event of events”

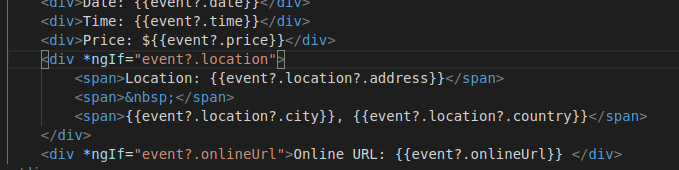
### Handling NULL values

When binding the data through interpolation if the object is **undefined** then we will fall into issue. (can’t read property of undefined)

To avoid it we can use **safe navigation** operator **“?”**, where we are using object

### Hiding and showing elements with \*ngIf

**\*ngIf** is a structural directive.



\*ngIf will just not hide the element but comment (completely removes the element ) from DOM if expression evaluates to false.

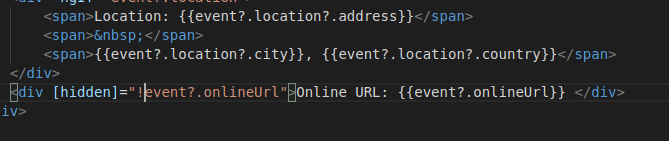
Which is a performance saver, but in some case also we just need to hide the element but not remove from DOM.

### Hiding content with [hidden]

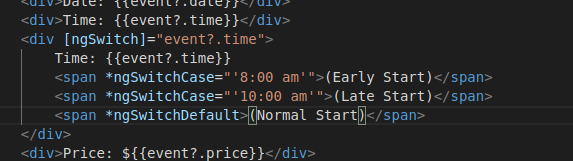
If we need something to show and hide frequently based on button click or something. It’s better to hide the element rather removing and adding to DOM every time (on every click).

Hiding here will improve the performance.

In HTML we have **hidden** attribute, using angular we can bind to a property as **[hidden]**

****

Hiding and Showing content with [ngSwitch]



### Styling component with ngClass

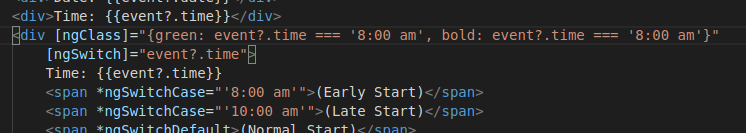
There is a special type of binding is called class binding as

<div **[class.green]**=”event?.time === ‘8:00 am’” …

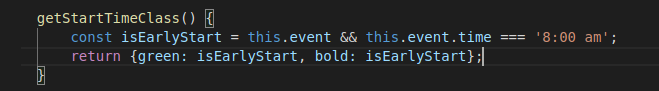
Here **green** is a css class we can create

.green {color: #003300 !important;}

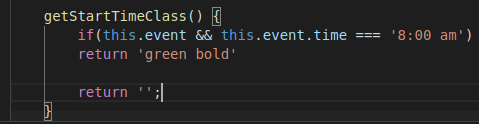
But if we have many class to bind we have to use **[ngClass]** binding. This binding expects a object representing key as class name and value is an boolean expression.



Even we can call a function and that function returns object as below



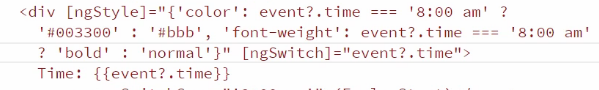
Or also



We even can use this approach if the html attribute already have a class attribute, in that case both class binding and ngClass will append to the existing class.

### Styling component with ngStyle

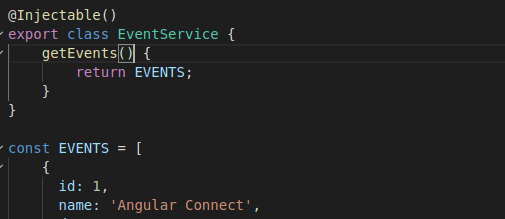
Just like class binding we can attach single style to tag using style binding. To use multiple style we need to use **ngStyle**

****

## Creating reusable Services

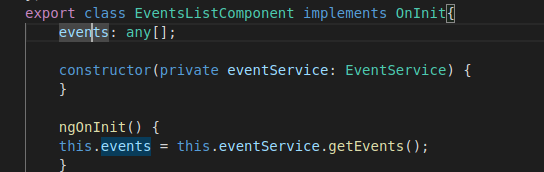
### Creating service

Create and export a class



Map this class under **providers: []** in app.module.ts

Inject this class in the component where to be used

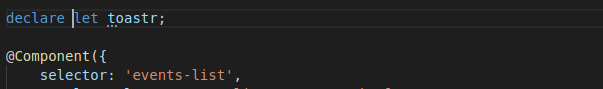


### Wrapping third party library into our app

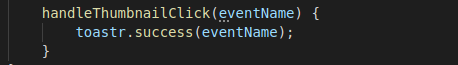
Here we will download and use a third part library toastr(create pop up notification message).

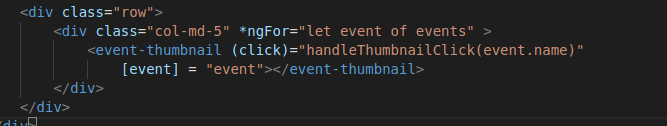
1. $ npm install toastr –save
2. Map the toastr css and js file under “styles” and “scripts” in angular.json. This will make toastr available **globally.** Now on the top of the class where we will be using we can declare variable as **declare let toastr;** This will let type script know the variable is in scope already.





Inside the class we can have method bind to an event

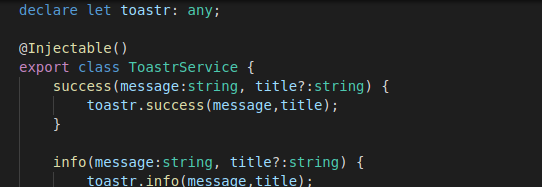




Problem with above approach

* Making global reference is not a good idea
* As making global, we can’t test it as we are not injecting we can’t mock it.

To solve these we will create a “ToastrService” and wrap the methods we are interested in



Then inject this service to required component and use the method.

## Routing and Navigation

### Why Routing is necessary

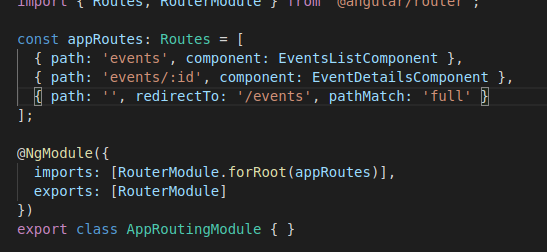
Before SPA(Single Page Application) every time user request the server will render a page based on user’s query. Here each file is different from other and every file will be loaded from the server and entire previous page will be replaced even 60% of the page is same.

Modern application loads a single page into memory, typically index.html and all other pages(not really pages) will be loaded by javascript. Initial index.html was the only page load and others will be the portion of it which will be loaded by JS. That portion will be replaced as we navigate from page to page in site.

### Adding Route

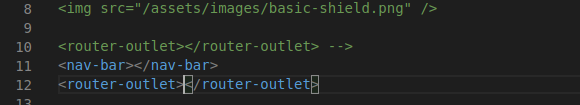
Create a component (event-details.component), when a user clicks on a specific event the detail will be shown. This is not a new page but rather that section will be brought to the page.

Create a module “app-routing.module” under “app” and configure the application routes as below



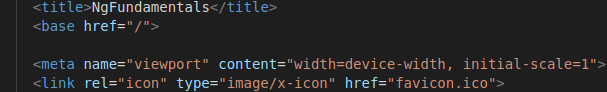
Configure the module under root “app.module.ts” in **imports[]**

Now in the root html (starting page of the app) we need to remove all the independent component mapping and add a tag **<router-outlet>**



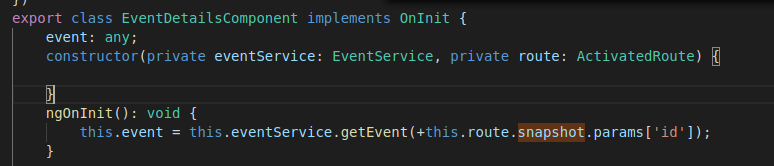
Now we can **remove** the **selector** from every component as we will be accessing through path and component in **appRoutes.**

**Most importantly** we need to let our application know the path from where all these routes are relative to and that we need to configure through **<base href=”/”>** or **<base href=”/puma/”>** in our root file i.e. **index.html**

****

### Accessing Route Parameters

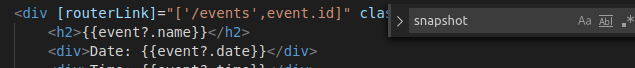
So we have **ActivatedRoute** class/package. We need to inject to the component to which we are accessing with a path parameter and then can use as below.



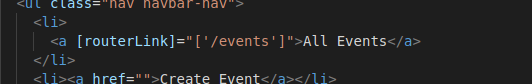
**+** used to convert to **number** as the method getEvent expects a number.

### Linking to Routes

If we want to make any of our html attribute as links/routes that’s why [**routerLink]** is for. It takes an array which contains the path as first argument and series of path values.



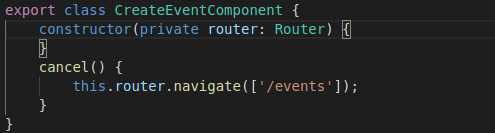
We can add [routerLink] to anchor <a> element as well



### Navigating to page from code

To navigate a path from the code and that’s why we have angular **Router** service is for.

Inject the Router service to the component and use **navigate** method. This method takes an array with first element as the path where we want to navigate



### Guarding against route activation

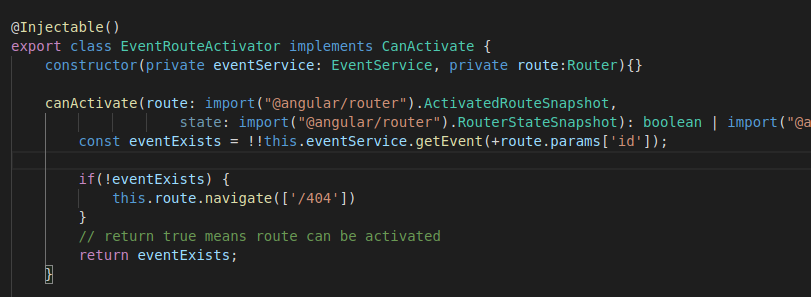
Sometimes we need to restrict a user to go to a certain page or discrete the user from leaving from a page. That’s what **Route Guards** are designed to do.

**canActivate** guard decides whether or not user is allowed to access the route path.

In above case if event id is not valid we want to route to **404 page.** Create 404-error.component under /app

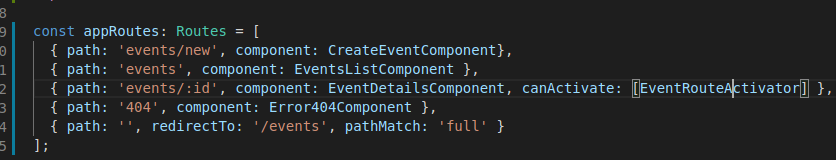
Create a EventRouteActivator service which will take a call based on logic to decide whether to activate the route or not and also can navigate to error page from here.

Our service should implement **CanActivate** service and implement method **canActivate()**

****

Now map this route service to the **route path** where we need a decision to make whether to route or not.

(In above **+ convert string to number** and **!! makes an result to Boolean**)

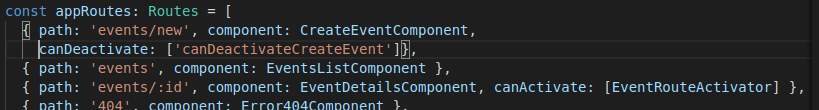


### Guarding against route de-activation

Just we use CanActivate to prevent a user to navigate to a page. In similar way we can use **CanDeactivate** to **prevent a user leaving a page.**

There are two ways to add route guard, either as a **function** or **service.** We already seen as service let’s here add as function.

Register ‘canDeactivate’ route guard to app routes as a function



Register in app.modules.ts as long hand syntax in “providers” and define the function



### Preloading data for component

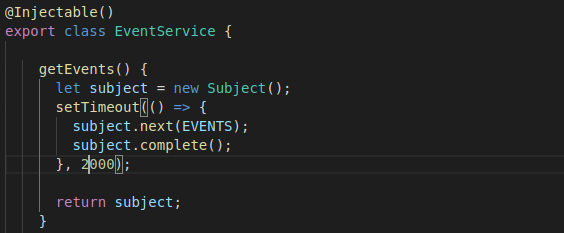
We need to use **rxjs** library and Observable. Need to return a type of Observable (here Subject is type of observable) from event.service.ts

From events-list.component we need to subscribe to the observable.

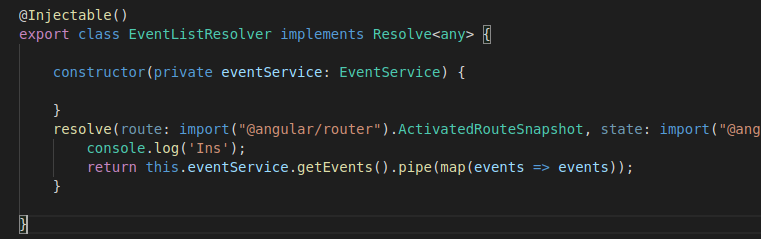
### Resolve guard

Using resolve guard we can load the entire page at once when data received from service (delayed call) rather half loading the page at beginning and waiting till data arrives to load and appear in data pane.

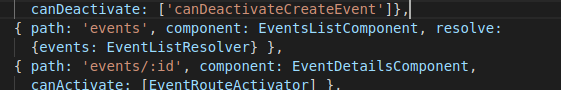
In the service make delay to 2sec



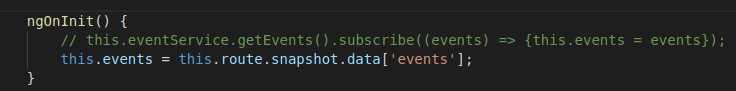
Create a resolve guard service and call the event service from there



Map the resolve guard to route, it will map the events data to request with name “events”

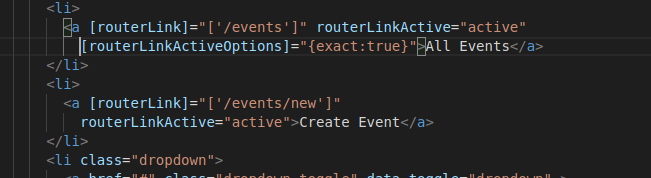


In the component get the events data



### Styling the Active links

We need to add **routerLinkActive=”active”** to all the events. Add the **active** cssclass to component.





### Lazy loading feature modules

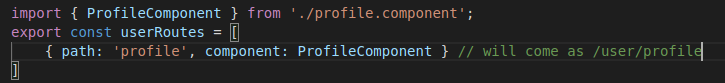
Here we will create a feature module or Lazy loadable module. There are 2 key differences between App Module and Feature Module

* Here in the imports we will add CommonModule where as in App Module we need to import BrowserModule
* Here we need to import RouterModule.forChild where in App module we import RouterModule.forRoot

Create UserModule as below also ProfileComponent



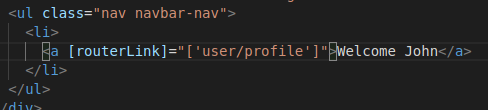
Define user routes (user.route.ts)



In App route load the children’s of UserModule

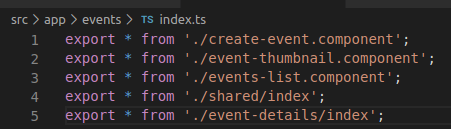


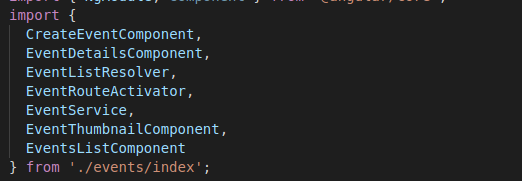
Configure [routerLink] to ‘user/profile’ from navbar html



### Organizing exports with Barrels

This is a way of managing imports in a single file **index.js** inside each folder and mapping it in root module where many files to be imported.





## Collecting Data With Angular Forms and Validation

### Using Model for Type Safety

We can create and user Model with defined type for type safety, instead of using “any” type.

### Creating first Template Based Form

Template based form are simple and easy and can be created entirely in HTML template. It works great for simple forms.

To use Forms we need to import **Angular FormsModule** in our module. Which allows to access number of template based form features.

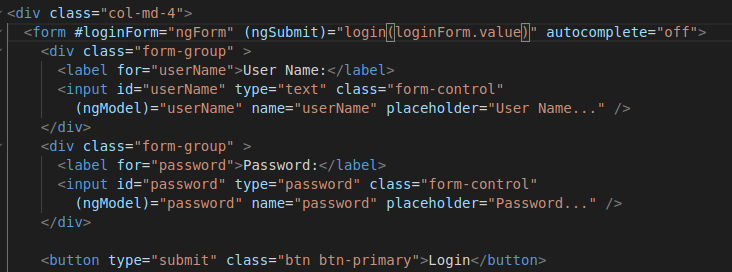
Angular provides [**(ngModel)]** directive to bind the input fields. This is a two ways binding.

If we make it **(ngModel)** this is one way binding. **ngModel requires to define a name attribute in the element.**

Now the (ngModel) will bind the data to form. And we can access it by using a local variable like #loginForm=”ngForm”. And we can pass this variable to (submit).

Angular provides a directive called (ngSubmit) we will use that because that provides few extra things like preventing submitting the form to server.

When we bind something with (ngModel) it’s better to declare on component too.



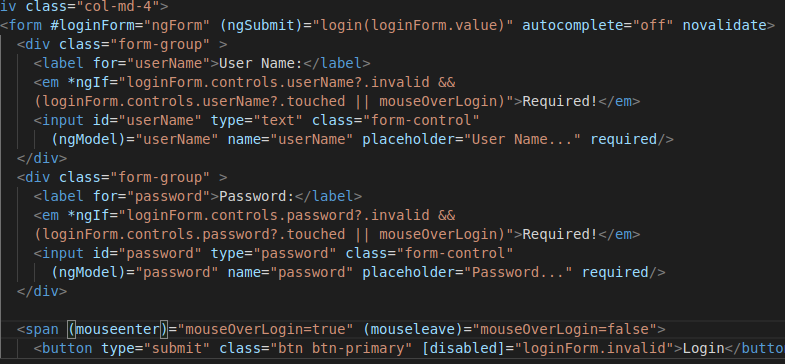
### Using the data from template based form

One important things to note **providers[] are shared between modules.** Means when we register a provider in root module it is visible in child module.

But same is not the case for **imports[]** or **declarations[]**

### Validating Form

We can use form control properties like touched, dirty, pristine



### Creating First Reactive Form

It is also called as model driven form. Basically we will define our fields and validation in component and wire them up to html template.

This approach requires some extra code but have advantage. For example we have option of building forms and validation more dynamically based on decision made on code.

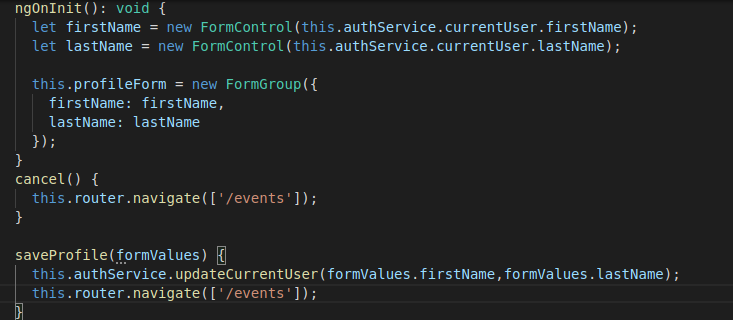
Another benefit is it makes all our validation logic unit testable.

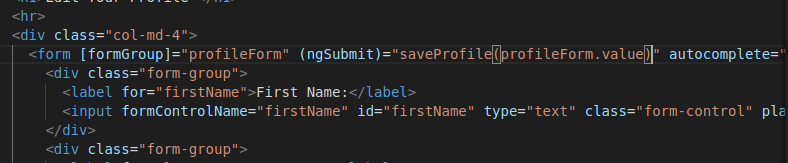
We need to import **ReactiveFormsModule** in our module.

Major components are **FormControl**, **FormGroup.**

To bind the form group to html **[formGroup]** This name should match with form group name in component.

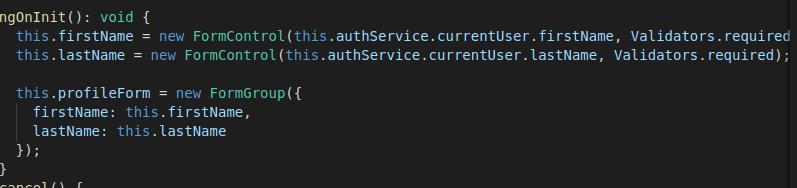
To bind the elements in html **formControlName** This name should match with name in component.

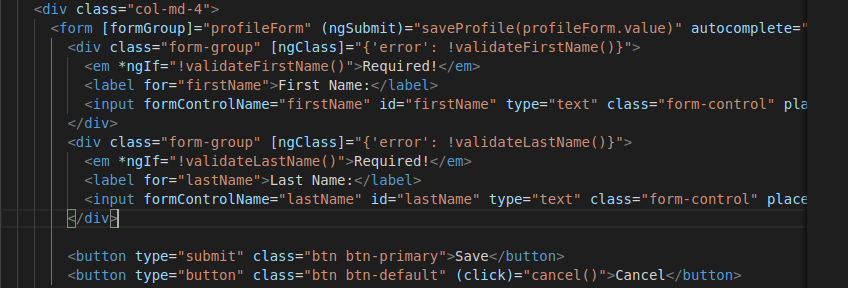




### Validating Reactive Forms

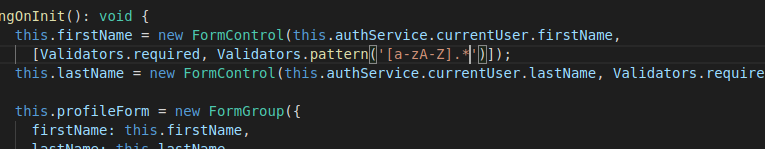
One of the benefit of reactive form is we can define our validation in component which is unit testable.

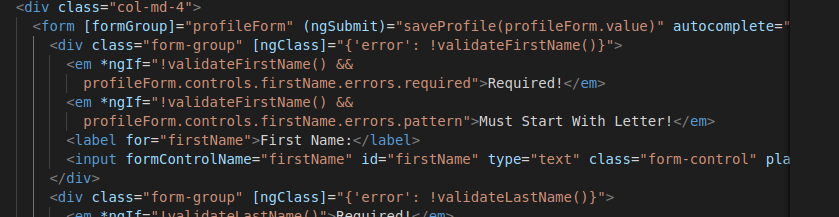




### Using multiple validators in reactive form

We can pass array of validators to our FormControl. In the template we can check error condition for each validation type ex. required, pattern



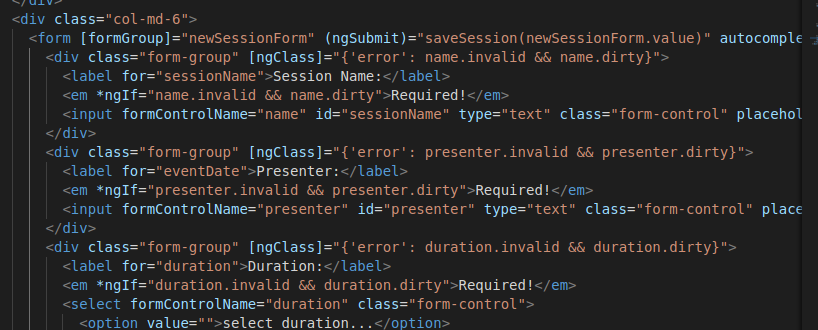


### Diving deeper into template based form

To bind multiple fields to a single object we can use **ngModelGroup** in our template

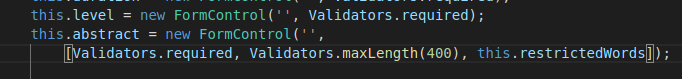


### Diving deeper into Reactive form



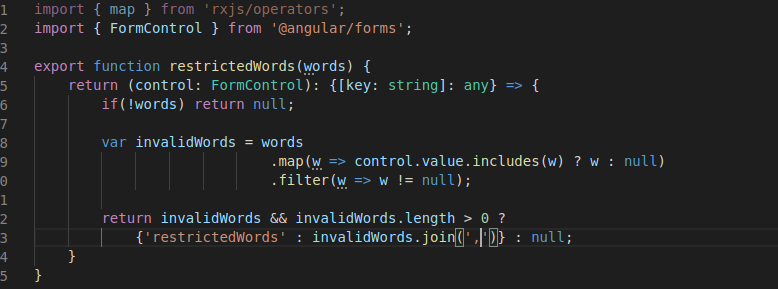
### Creating Custom Validators

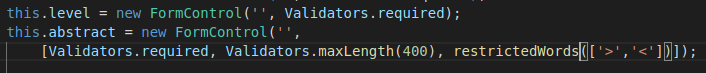
Custom validator will be a function. (can have a method in our component and reference that method in validator array that will get executed by angular runtime by passing the FormControl when something applied on the tagged field)

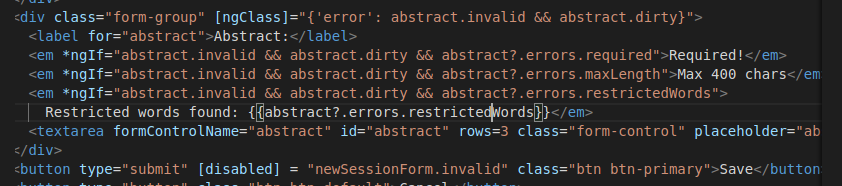


**As a reusable Function:**

We can have our custom validator as a function which returns a function.

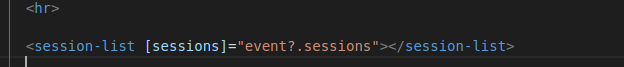






## Communicating between components

### Passing data to child component



Here **[sessions]** property of event-list component binds to **event.sessions** property of the parent component (in the HTML where this is declared)

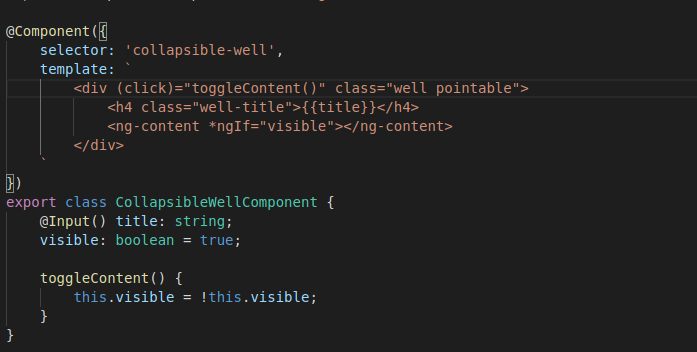
## Reusing Components with content projection

### Content Projection

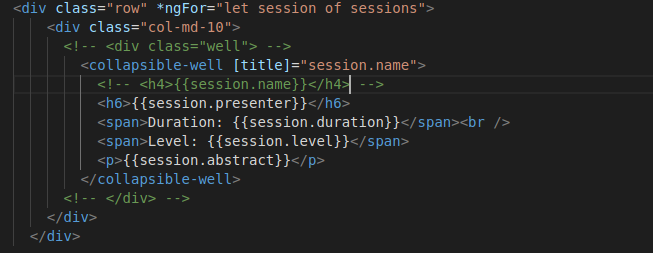
With content projection we can change the contents inside a component template based on the need of the application.

In the first step we will make the headers collapsible for our Sessions and will just show the title.

Create a common component as “CollapsibleWellComponent” as below



**<ng-content>** will bind the content into it.



Here we used <collapsible-well> in our session-list.component.html where the contents inside <collapsible-well> will bind to <**ng-content>** in our CollapsibleWellComponent class.

In our implementation when we click on the <div> it will toggle the visibility.

### Multiple slot content projection

We will indicate the session is popular with multiple votes. (Will add a flame icon if a session with certain number of votes)

**<ng-content >** has a **select** attribute we can give a css class selector with dot or id with hash or we can give an attribute our own (preferable). With select attribute we can decide which content we want inside the current <ng-content>

With this **select** attribute we can decide multi slot content projection.



Here we have two <div> one for title with some logic for hot icon and other for the body

“well-title” and “well-body” are the attributes which we will be using in CollapsibleWellComponent **select**.

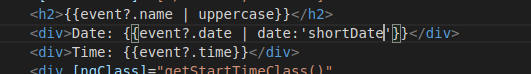


## Displaying Data with Pipes

### Using Built-in Pipes

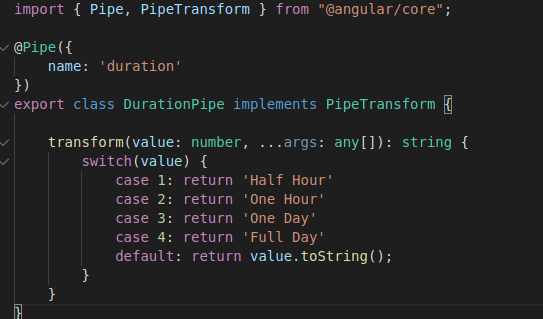
Angular pipes are majorly used for

* Formatting the data
* Sorting
* Filtering



### Custom pipes

We need to use @Pipe({}) decorator and “PipeTransform” interface implementation



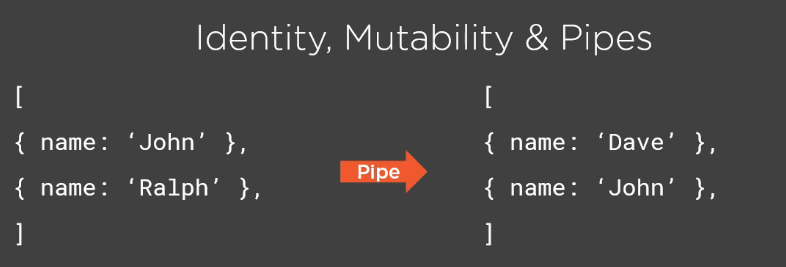
Import the pipe in declaration[] and use it (with name value) in the template where to be used.

### Sorting and Filtering

In-fact here we need to discuss why in angular we don’t use pipe for sorting and filtering.

In javascript Objects and Arrays are Mutable. So it can change the state without changing the identity (name).

In Angular pipes run when identity got changed. For objects in javascript if value got changed, identity still remains same (Mutable). So pipes don’t run if in an object value got change (Because this is an expensive operation to go through all the data in object to identify what got changed).



So there is another option in angular called **Impure Pipes.**

Impure pipe runs on every change detection cycle. That means the sorting and filtering logic we write will execute every times an event happens in the application (performance problem).

This Impure pipe is same as Angular 1 Filters. Because of performance problem this is hidden in angular 2 and pipes are not the recommended way to sort or filter the data.

So we need to add Sorting and Filtering our data **without use of pipes**.

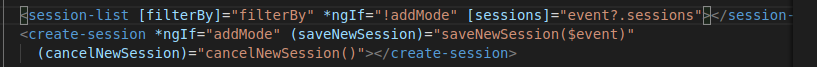
### Creating a Filtering Display

We can implement our component class from **OnChanges** interface and implement method **ngOnChanges()** method.

This method will execute every time when **@Input()** property changes.



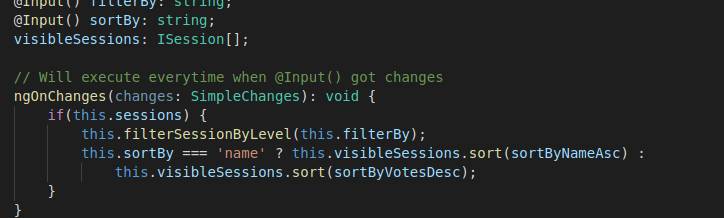
Here we passed the property filterBy from parent component (events-detail.componet.ts)

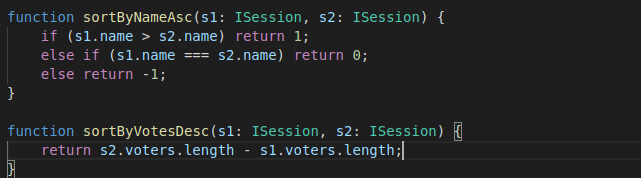


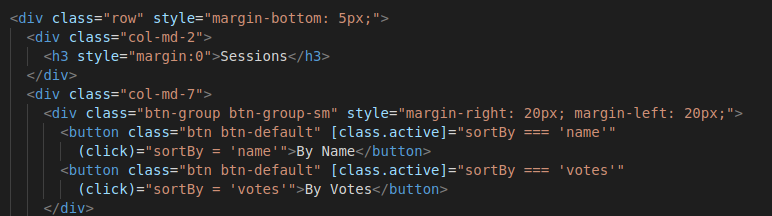
### Sorting the data

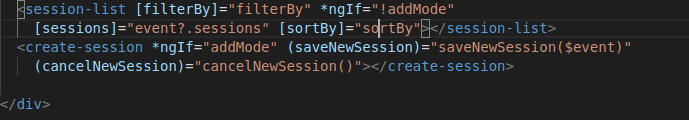
Sorting has here the same flow how we did it for filtering. Business logic varies we need to use sort method of array and pass comparator function.

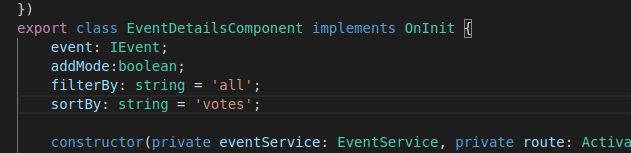
Apart from that we are passing an @Input() property sortBy









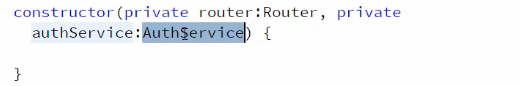


## Understanding Angular’s Dependency Injection

### Angular Dependency Injection

When we declare a service in module provider[]. Angular will create the instance of the service and add it in its service registry.

It uses type declaration as key



So with type declaration angular finds the object of the class we are looking for and inject it into our component class.

### Using Angular’s Injection Token

**Problem with third party library use**

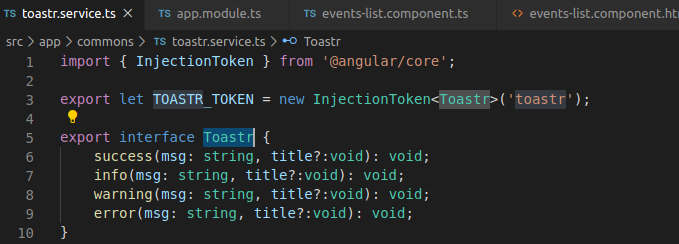
There are challenges when we use third party service/api (ex. toastr, jQuery). If we declare it globally (let declare toastr: any) it will be available global and which is not encouraged because we can’t inject this global variable reference.

So we wrapped it in a class (ToastrService) for our dependency injection and overridden the methods. This is also a bad idea because for simple API like toastr it could be ok but when we have huge library like jQuery it will be horrible to implement the methods.

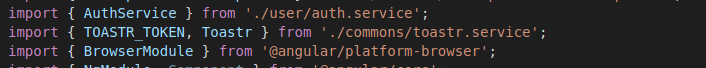
Here it come the Angular’s **injection token** as rescue. Which means angular provides a way to create a key or token for injection purpose, without creating and exporting a class.

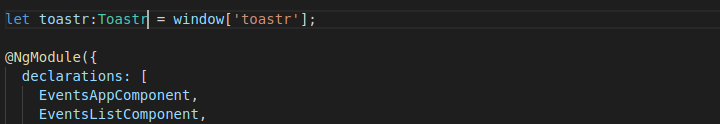
The “InjectionToken” job is simply to create a token for angular dependency management registry in order to find the instance of the service we want.

Create and export an injection token as below

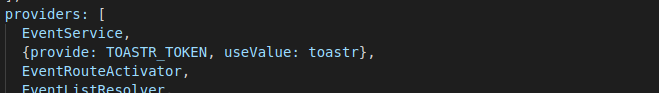


The global declaration of “toastr” we moved to app.module.ts as below



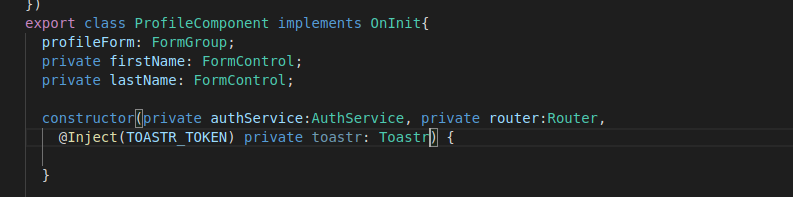


Above “Toastr” class type we exported from toastr.service.ts else we could use “any” here.



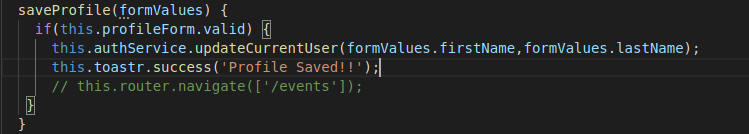
### Using Angular’s @Inject Decorator

@Inject is a decorator with which we can inject the service we created using “InjectionToken”



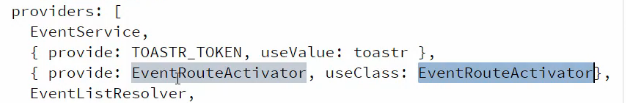
Here the object will be identified using key TOASTR\_TOKEN and with that it will be looked in Angular’s dependency management registry.

And now to use it



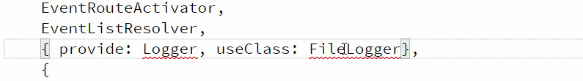
### The “useClass” Provider

In the providers [] “useClass” can be used as an long hand syntax for registering a provider.



It is same as just configuring simply ‘EventRouteActivator’. But this gives a elaborative description saying when someone looks with a key “EventRouteActivator” give them the class “EventRouteActivator” from Angular’s registry.

But with this syntax i.e. long hand style of declaration we can do some funky things which is very useful in below case.

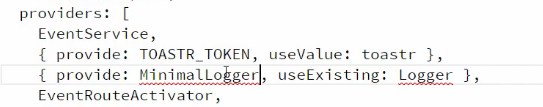


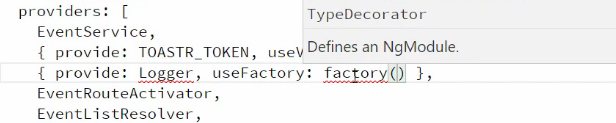
When someone asks for a “Logger” type we can give standardize “FileLogger” type.

### The “useExisting” and “useFactory” Providers

There are two more ways to register provider “useExisting” and “useFactory”.

“useExisting” also called as alias provider.





## Creating Declarative and Advanced Components in Angular

### Creating Directives – The Trigger Directive

We need a trigger to open-up the model when someone searches some text and click on “search”.

To open-up a modal we need to use jQuery as below

$(‘#id’).modal()

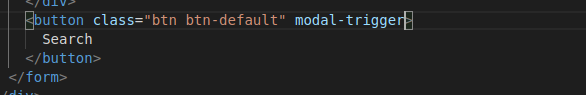
We can bind a (click) event handler to “search” button and put the above code. But we shouldn’t be doing it for a big reason as the navbar component will be tightly bounded to jQuery implementation of modal i.e. under the hood jQuery call of modal() method and which is there because bootstrap modal is also part of which is getting loaded.

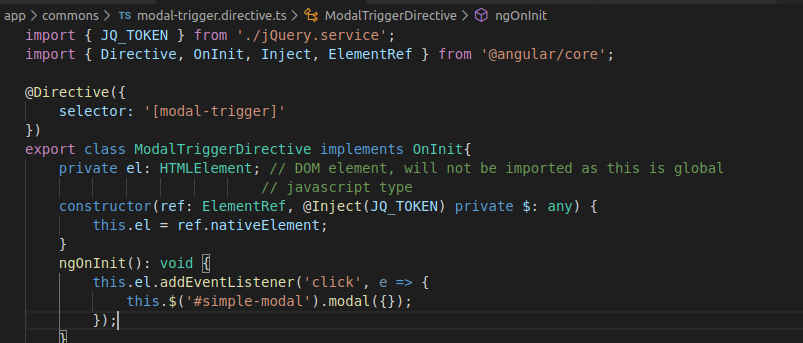
We want to hide this implementation details behind something else and a great way to do that will be a **directive.**

**Directive -** With angular directive we can hide the implementation details.

“Directive” is mapped to declarations[] in app.module.ts

So in HTML template we need to say to button when you are clicked you are a trigger to my modal component. So we need to give it an attribute like below.





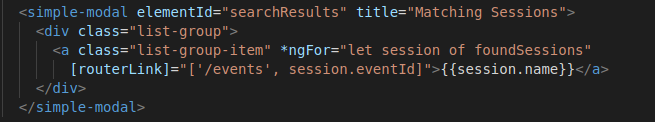
**ElementRef** is the pointer to the element. This tells angular when this element is created I need the reference to the element which is on, in our case “button” element.

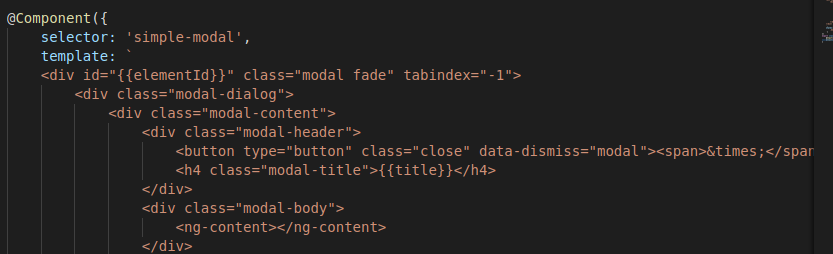
### Binding and ID, making modal directive reusable

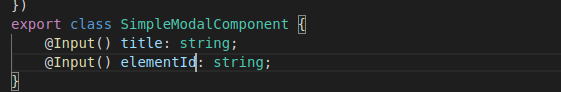
Now the modal-trigger.directive is bind to id ‘simple-modal’ which makes it tightly bounded. If I have other modal and trying to use the same directive can’t reuse as its bounded to ‘#simple-modal’.

To make it reusable we need to dynamically pass and use the id rather tightly binding it.

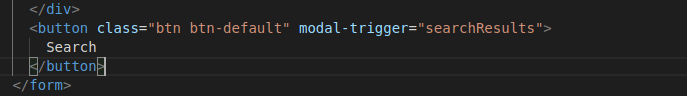
We need to pass id as input parameter



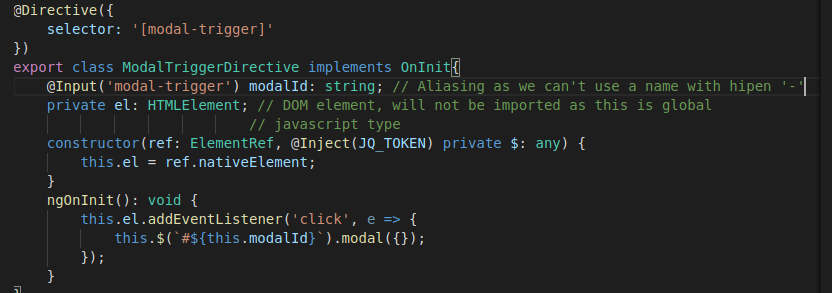




To pass it to common modal directive we need to pass it in attribute from navbar template.

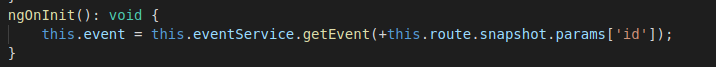


To directive we need to pass @Input and name should be same as attribute name.



### Routing to the same component

Literally we will be fixing a bug here which exists in our event-detail component, i.e. being in the same component (event-details) when we try to navigate to a different event-detail. Here the URL changes but doesn’t navigate.



We are passing the ‘id’ parameter from our route. The first time this component initializes. But when we navigate from /1 to /4 what angular doesn’t do is reset the entire state of component and re-initialize it and reconstruct it. Instead all happens is param[‘id’] changes.

It’s taking an advantage from the fact that the “param” is an Observable. Here we using ‘snapshot’ which is not an observable, which is not subscribe to any changes. So if the ‘id’ parameter changes we don’t know about it.

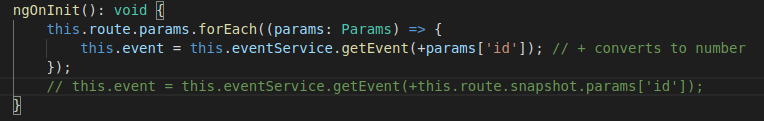
That’s how angular2 does the routing when we are on a parameterized page (event-details). And that’s the bug we will fix here.

It doesn’t happen when we navigate from other page (events) to event-detail because the init() method will execute every time.

But from event-details if we try to navigate to other event-details we will face the issue because state of the component remains same.

**Fix:**

In order to fix it, we need to subscribe to the route parameter



**Note:** One very important things we need to make a note we are calling event service and with ‘id’ parameter getting the event but **not resetting the other states of the event-details.** Which we need to do explicitly as well as the other event-detail where we will navigate its state should not be same where we currently are.

### Using @ViewChild Decorator