# 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.

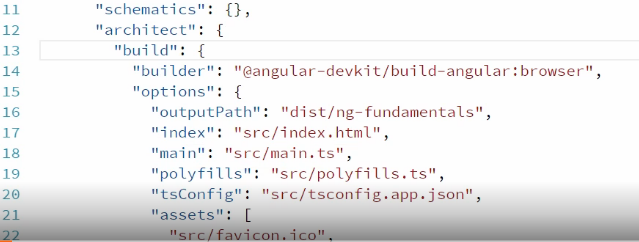
Delete the “node\_modules” and then npm install

$ 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

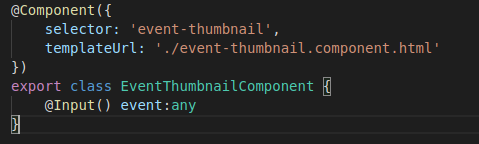


Register the component to root module file (app.module.ts) in **declarations[]**

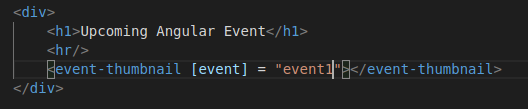
### Passing data to child component

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

Child component:



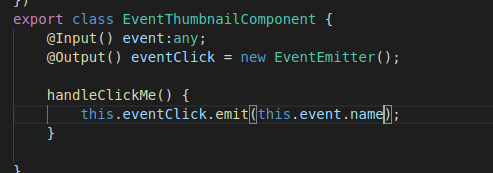
Parent component (especially template file):



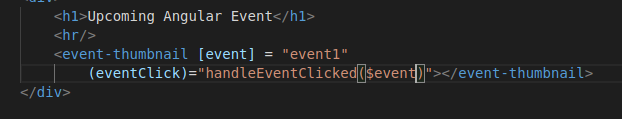
### Passing the data from child to parent

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:



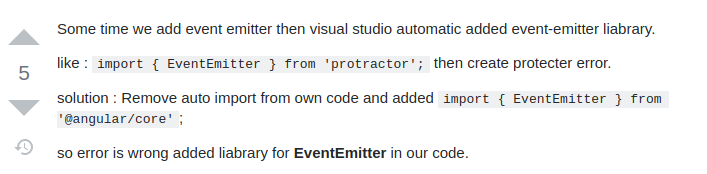
Capture the event emitted in parent template:



Define a method in parent component.ts

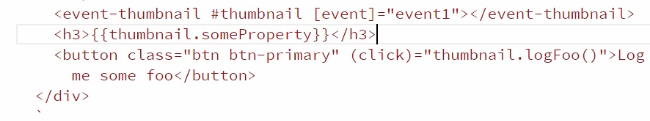


A Issue faced with **EventEmitter** import, many compilation packages were failing:



### Template Reference Variable

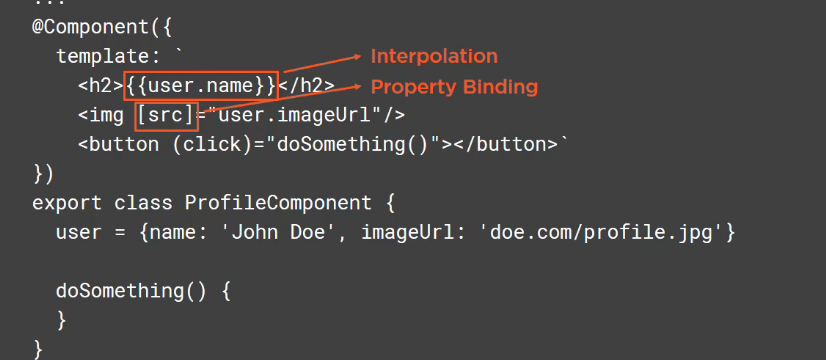
Template reference variable is a simple and straight forward way to use methods and variables of child component in parent component



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

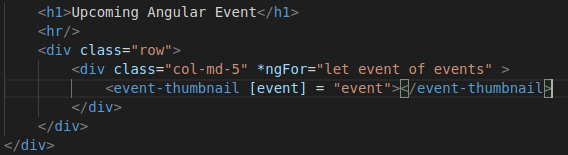
* Can’t use assignments (=, +=, ++, etc)
* Can’t use new keyword
* Can’t chain expression with ;
* Can’t use global name space as “console”

### 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”

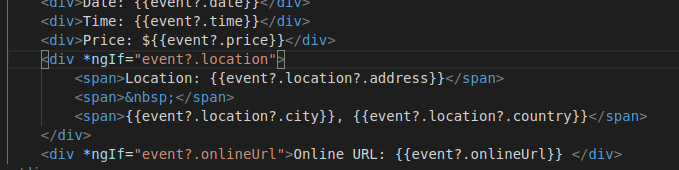
### Handling NULL values

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.

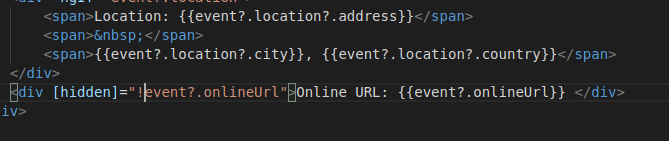
Which is a performance saver, but in some case also we just need to hide the element but not remove from DOM.

### Hiding content with [hidden]

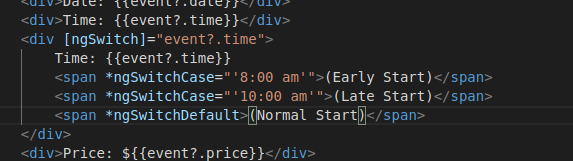
If we need something to show and hide frequently based on button click or something. It’s better to hide the element rather removing and adding to DOM every time (on every click).

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

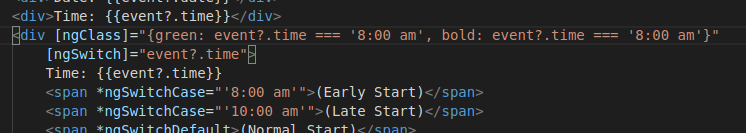
There is a special type of binding is called class binding as

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

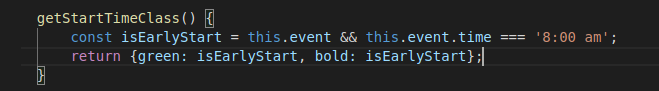
Here **green** is a css class we can create

.green {color: #003300 !important;}

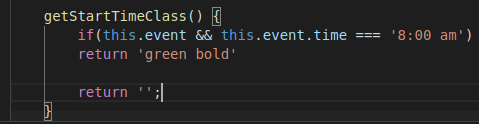
But if we have many class to bind we have to use **[ngClass]** binding. This binding expects a object representing key as class name and value is an boolean expression.



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



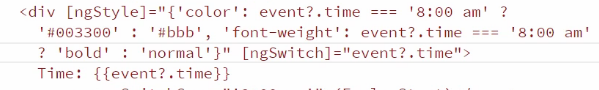
Or also



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

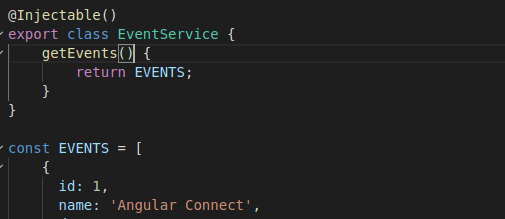
Just like class binding we can attach single style to tag using style binding. To use multiple style we need to use **ngStyle**

****

## Creating reusable Services

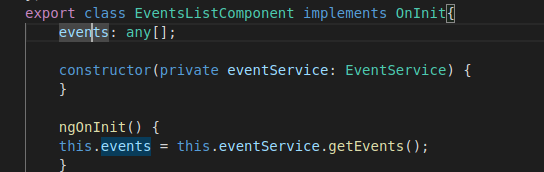
### Creating service

Create and export a class



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

Inject this class in the component where to be used

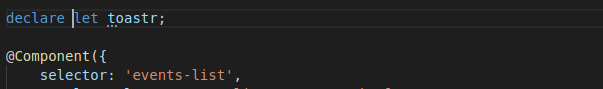


### Wrapping third party library into our app

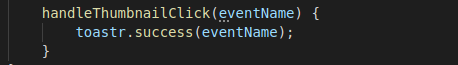
Here we will download and use a third part library toastr(create pop up notification message).

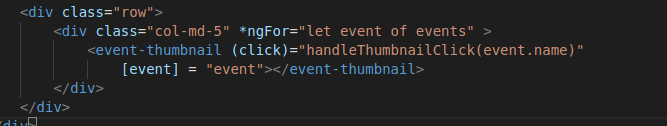
1. $ npm install toastr –save
2. Map the toastr css and js file under “styles” and “scripts” in angular.json. This will make toastr available **globally.** Now on the top of the class where we will be using we can declare variable as **declare let toastr;** This will let type script know the variable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





Inside the class we can have method bind to an event

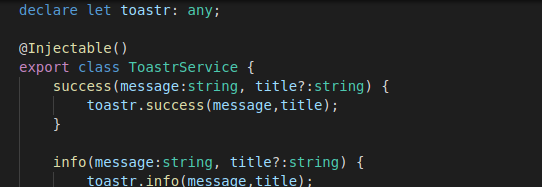




Problem with above approach

* 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

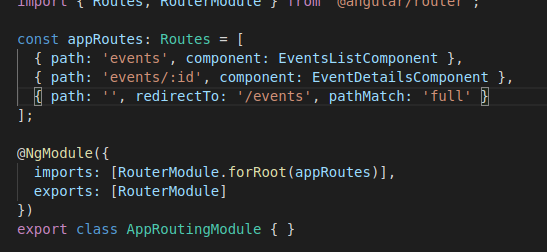
Before SPA(Single Page Application) every time user request the server will render a page based on user’s query. Here each file is different from other and every file will be loaded from the server and entire previous page will be replaced even 60% of the page is same.

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

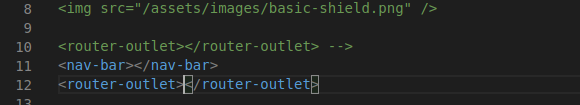
Create a component (event-details.component), when a user clicks on a specific event the detail will be shown. This is not a new page but rather that section will be brought to the page.

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



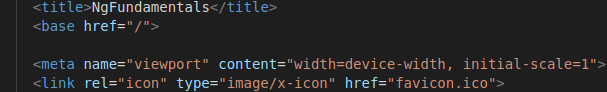
Configure the module under root “app.module.ts” in **imports[]**

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>**



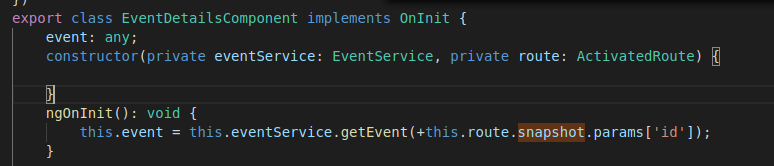
Now we can **remove** the **selector** from every component as we will be accessing through path and component in **appRoutes.**

**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

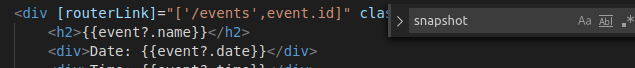
So we have **ActivatedRoute** class/package. We need to inject to the component to which we are accessing with a path parameter and then can use as below.



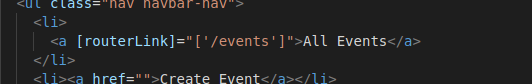
**+** used to convert to **number** as the method getEvent expects a number.

### 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.



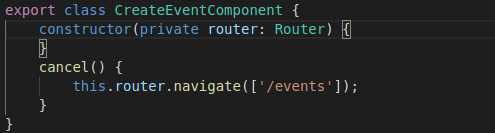
We can add [routerLink] to anchor <a> element as well



### 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

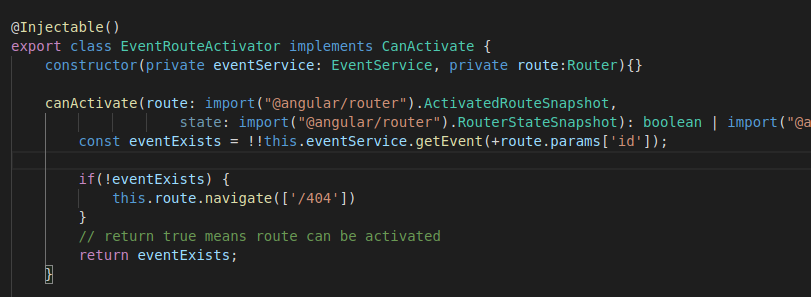
Sometimes we need to restrict a user to go to a certain page or discrete the user from leaving from a page. That’s what **Route Guards** are designed to do.

**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

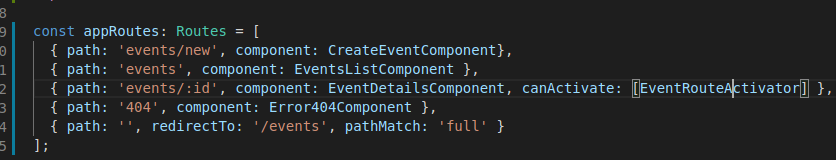
Create a EventRouteActivator service which will take a call based on logic to decide whether to activate the route or not and also can navigate to error page from here.

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.**