**Angular Fundamentals**

**Building blocks**

* Components
* Directives
* Services
* Templates

**Components**

A component encapsulates the data, html markup and the logic for view which is the area of the screen that user sees.

Say for example, we have Udemy website and we can see it as either a single component or bunch of components like Navigation bar component, side bar component, courses component, review component etc. So basically angular is component based architecture and we can write reusable pieces of code.

Every application has at least one component which is called App component. It is also called root component. A real world angular app is a tree of components starting from App component or the root component.

**Modules**

It is a container for a group of related component. Every angular app has at least one module which we call app module.

Say for example, we can have a module for classes, instructor dashboard, admin module, messaging module where all the messaging related components exits etc. As the application grows we will create smaller and more maintainable modules.

**Create a new angular project**

Now in order to create a basic angular project setup, run the following command

***ng new project-name***

**Components**

In order to use a component we should be doing following things

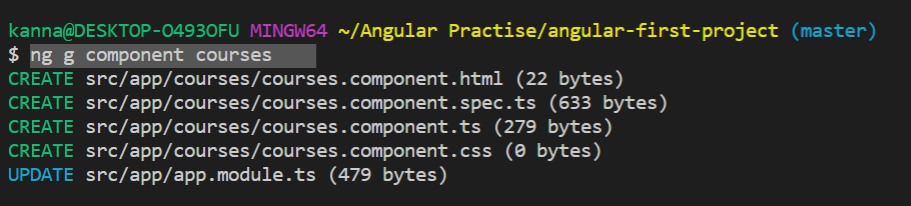
1. Create a component
2. Register it in a module
3. Add an element in HTML markup

First two steps can be done by using either manual way or by using the below command

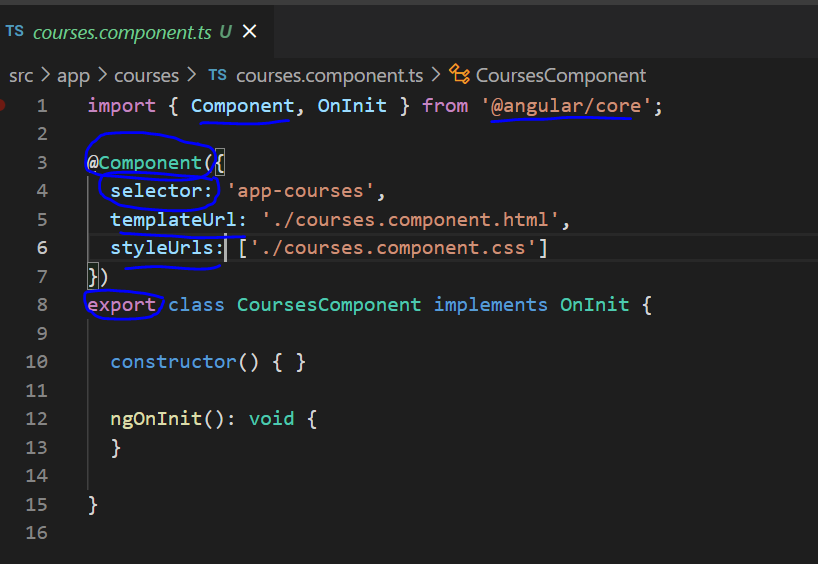
***ng g component component-name***

***Or***

***ng g c component-name***



If you notice here, after we ran the command, a new component was created which has component.ts, component.html, component.css files and also this module was registered in the app module.



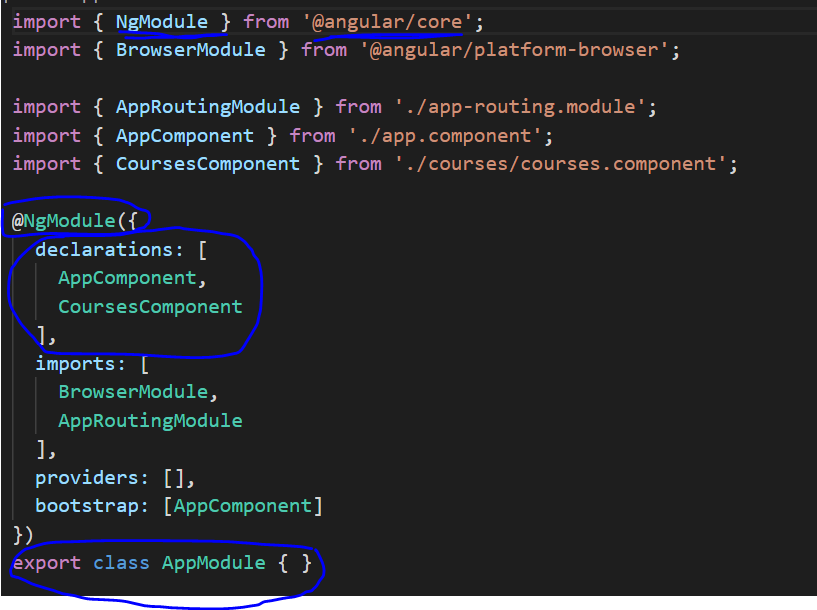
A component class looks like above, first we will have a normal typescript class and we use export keyword before class to make it available in other components. And then we wrap our class with a decorator called component which has three attributes,

selector – name of the element that we will be using to incorporate this component in another component. We can basically we use <app-courses></app-courses> element in another component template like any other html element, but for this component selector element the template code which is defined in html file will be rendered on the view.

template/templateUrl: we can give either path of the html template file or we can put inline html code

styleUrls: this is an array and we can give path to our component css files.

Now once we create a component we have to register it in the module. Check below module code for app module.



Again app module is a typescript class and we decorate it with @NgModule decorator with attributes like declarations where we register all our components that belong to this module, imports : we import all the other modules, providers is where we register our services.

**Directives**

We use directives to manipulate the DOM. We can use them to add a DOM element or remove a DOM element, change the class of a DOM element (in css style context),

Eg: \*ngFor, \*ngIf

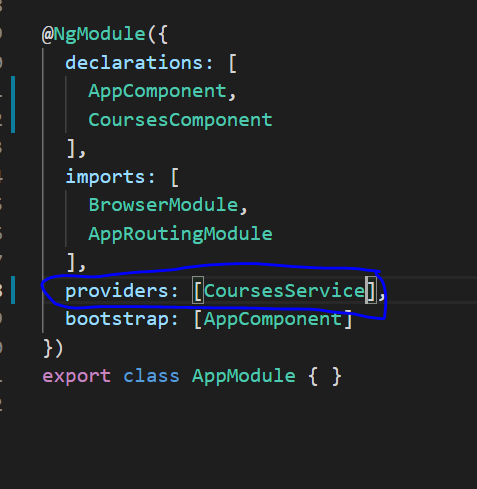
We should always have a prefix of asterisk to directive that manipulates the DOM element.

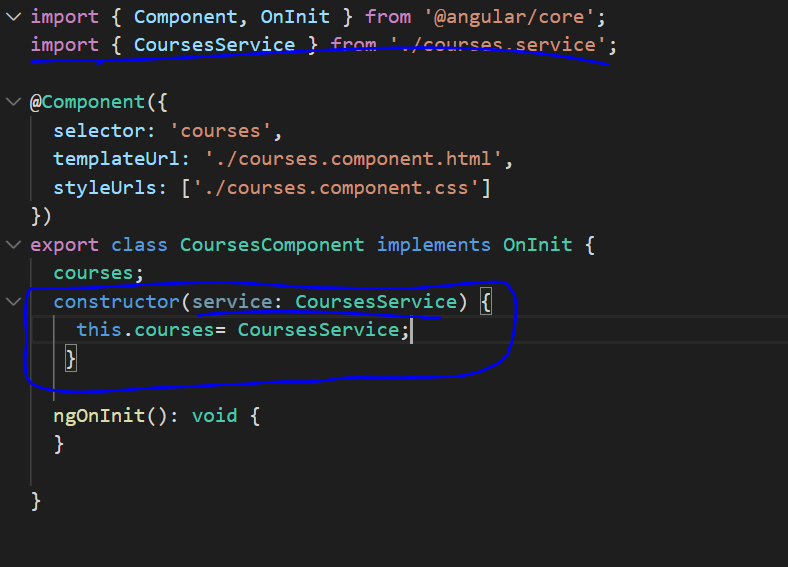
**Services**

We create service to make http calls. This logic is reusable. Because component should not have any other logic except the presentation logic, that’s why we create service class to make http requests.

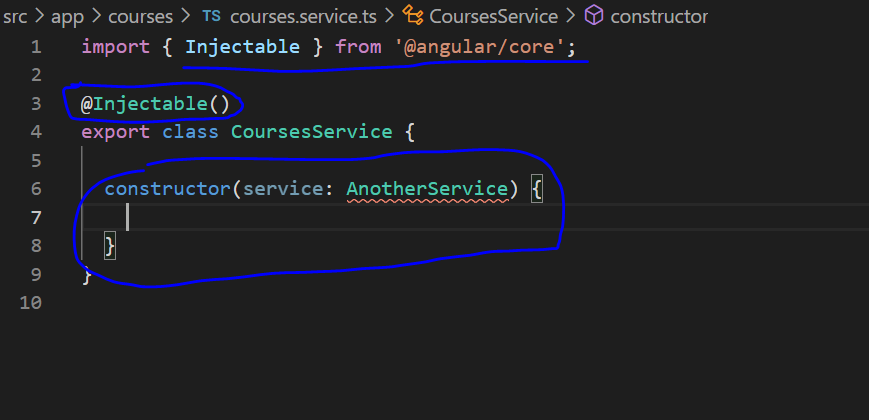
*ng g service service-name*

Service name example is courses.service.ts. Note using this ng g service command, it will not automatically register the service in app module in providers attribute of ngModule decorator.





In order to use a service in a component, we use dependency injection concept and we have to inject dependency via a constructor as show above. And also we have to register this service in ngModule providers attribute. Only if we register there, angular will know to create singleton object of service and inject into the component.



We have this decorator called Injectable that we can use to decorate service class. If we use this decorator then that means it has dependencies on other services. This is kind of auto wiring in spring framework. You may have question that why we are not using this in Component while doing dependency injection, simply because Component internally will have injectable. Injectable is also a Decorator that marks a class as available to be provided and injected as a dependency.

Displaying Data and handling events

Data binding help us coordinate communication between a component and its view template. Data binding consist of One-Way Data-Binding and Two-Way Data-Binding Interpolation is all about data binding and so as property binding as the name justify, but they both flow a value in one direction. it’s all about moving data in one direction from our components to HTML elements.

String interpolation and Property binding

<h1>{{ fullName }}</h1> -- **String interpolation**

<h1 [innerHtml]='fullName'></h1> -- **Property binding**

<h1>{{citedExample}}</h1>

<img [src]='imagePath'/>

Interpolation is a special syntax that Angular converts into property binding. It’s a convenient alternative to property binding.

**Property Binding:** to set an element property to a non-string data value, you must use property binding.

**<button [disabled]='isDisabled'>Try Me</button> </div>**

If we decide to use interpolation instead of property binding, the button will always be disabled irrespective of isDisabled class property value is true of false.

Now we need to know the difference between html and DOM. DOM – Document object model is a tree of objects in memory that represents structure of a document. html markup is a text format representation of DOM. When browser parses our html markup, it creates DOM in memory. Html elements and DOM objects will have attributes and variables.

<table>

<tr>

<td [colspan]=”colSpan”></td>

</tr>

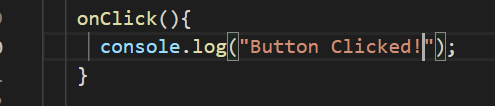
</table>

Here colSpan is a variable in component.ts file and we are doing property binding in html. But it will throw error. Most of the times html attributes and dom properties are same say for example src property of image is same name in both dom and html, but colspan is not same. So the above code has error. In order to tell angular to consider html elements attribute instead of dom property, we need to do like this <td [attr.colspan]=”colSpan”></td>

**Event Binding**

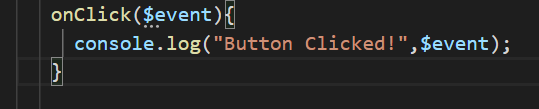
We use event binding to handle the events raised from the DOM, like keystrokes, mouse movements, button click etc.





Some times we may need access to the event object in the event handler, so for we have to pass $event to the function. Angular would know $event how to handle this object.





**Event Filtering**

In angular for events we can further filter them. For example we can write code in the following two different ways for keyup event.

Traditional way

<input type = “text” (keyup)=”onKeyup($event)” /> and in component

onKeyup($event){

if($event.keyCode ==13) console.log(“Key pressed!”);

}

Angular event filtering way

<input type = “text” (keyup.event)=”onKeyup()” /> and in component

onKeyup(){

console.log(“Key pressed!”);

}

**Template Variables:**

Traditional way

<input type = “text” (keyup.enter)=”onKeyup($event)” /> and in component

onKeyup($event){

console.log(“Key pressed!”+$event.target.value);

}

Angular event filtering way

<input #course type = “text” (keyup.event)=”onKeyup(course.value)” /> and in component

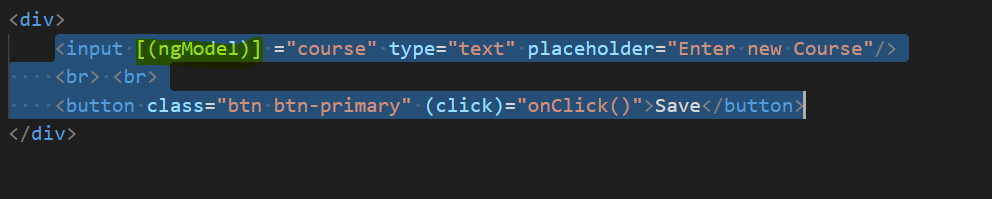
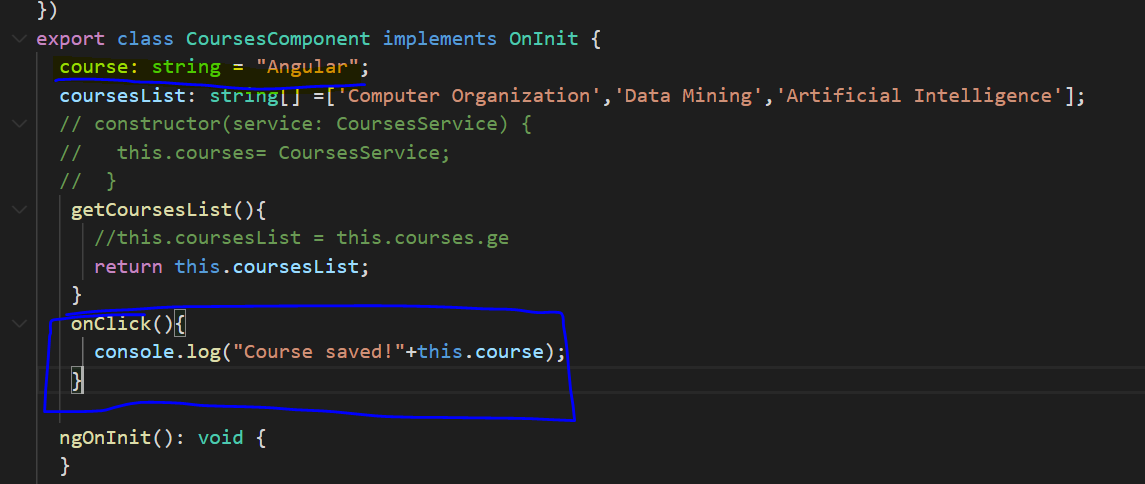
onKeyup(course){

console.log(“Key pressed!”+course);

}

**Two Way Binding:**

Until now we have seen one way binding from component class to the template using property binding and string interpolation. We were binding a component variable to template, when ever the value of variable changes in component it would get reflected in template. But the other way around was not done. Now with two way binding the other way around is also possible. We use banana in the brackets with an angular directive called ngModel this is present in FormsModule, so we have to include this in our App module and add in Modules attribute of ngModule directive.

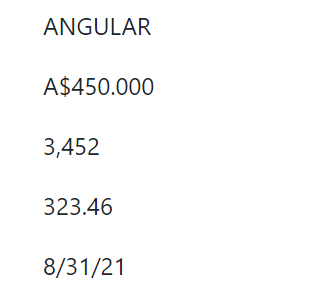
 

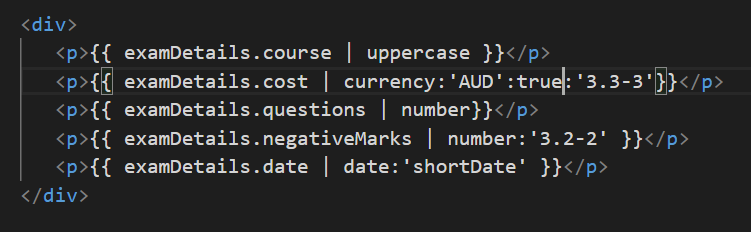
**Pipes:**

We use angular pipes to format data. We built-in pipes and we can create our own custom pipes.

Some built-in pipes are

* Uppercase
* Lowercase
* Decimal
* Currency
* Percent





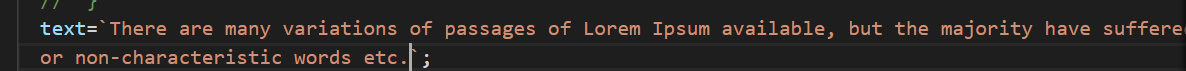
Here you can see we are using | symbol to transform the data using built in angular pipes like uppercase, currency, number, date etc. Also in currency, number, date we were passing arguments to further transform the data.

**Custom Pipes**

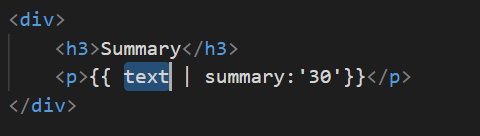
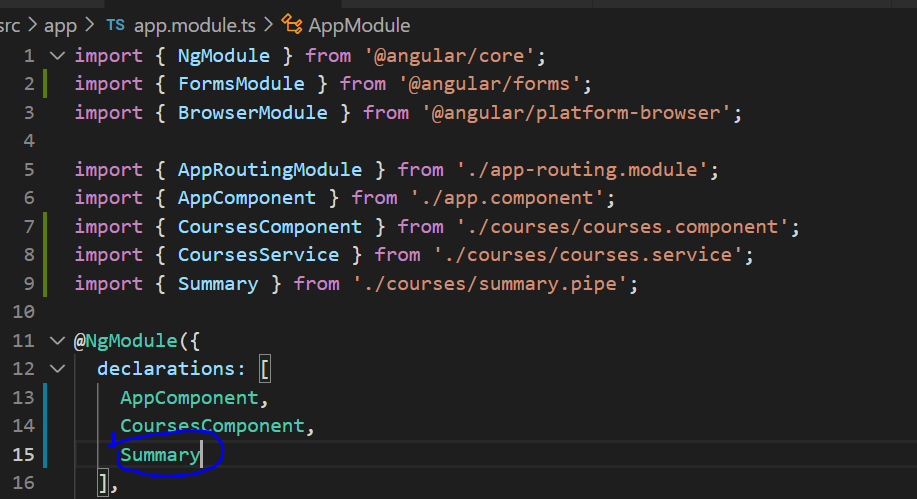
In order to create custom pipes, we have to create one class for example summar.pipe.ts

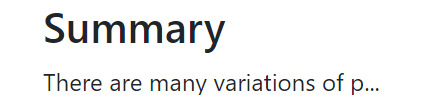
Now in this class we have to import Pipe and PipeTransform. Pipe is a decorator and PipeTransform is an interface and both are in @angular/core. Now our summary class has to implement PipeTransform interface and implement its method called “transform(value:any, args?:any……)”



Now in component lets have some text in a variable that we want to transform. 

And in template let display this text using string interpolation and use pipe to transfor the summary text and limit the size to some 30. And finally don’t forget to include this pipe in app module declarations.

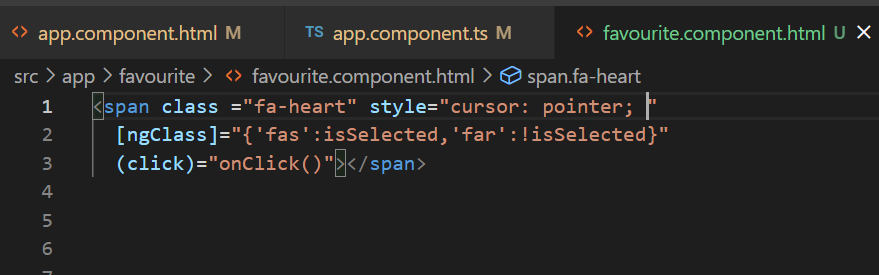


This will be view on UI.

**Building reusable directives**

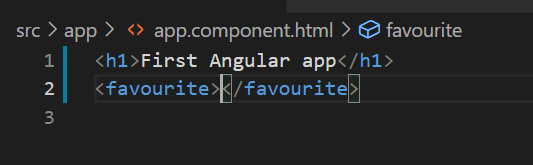
In order to understand this lets build a favorite component and toggle it when we click on it.

We will create a component with files favorite.component.ts, fovorite.component.html, favorite.component.css. Now inside component.ts of favorite, define a variable like isSelected and give some initial value. And then write a function onClick() to toggle this flag when ever someone clicks on it.

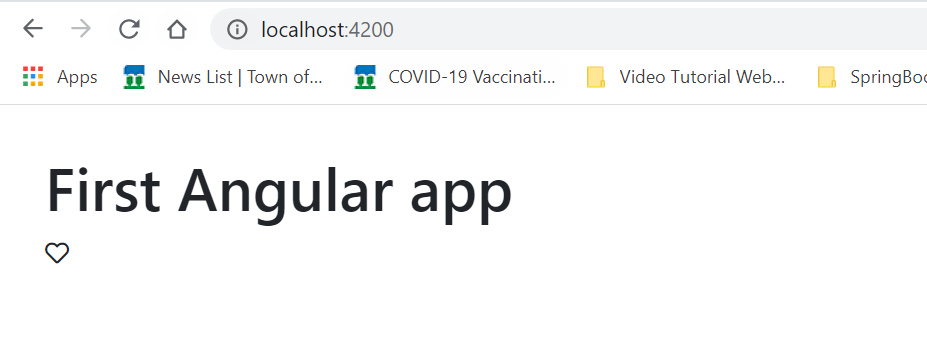




Now place favorite component in another component, in this example I am putting in app.component.html



Now we got the below screen

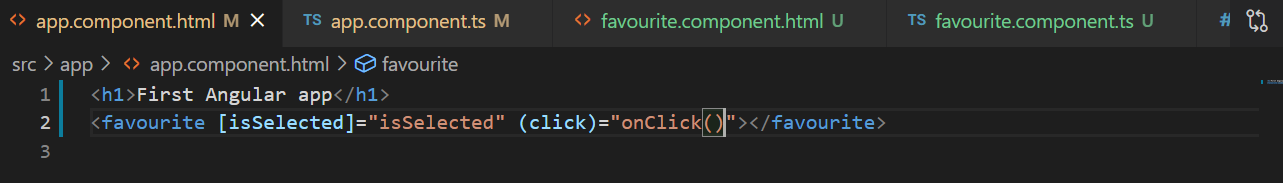


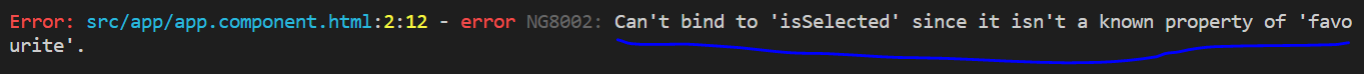
Note: I am using fontawsome icons to get this heart icon and if we click on this it be dark heart.

**Custom Component (Public Component API)**

Now this favorite component is not customizable. What I meant is we all know what property binding and even binding. Syntaxes are [property]=”variable” and (click)=”onClick()”. Here in property binding we are binding the DOM attribute (not html attribute) to the variable of the template component (favourite component.ts in this case) and we are handling DOM event inside the template component.

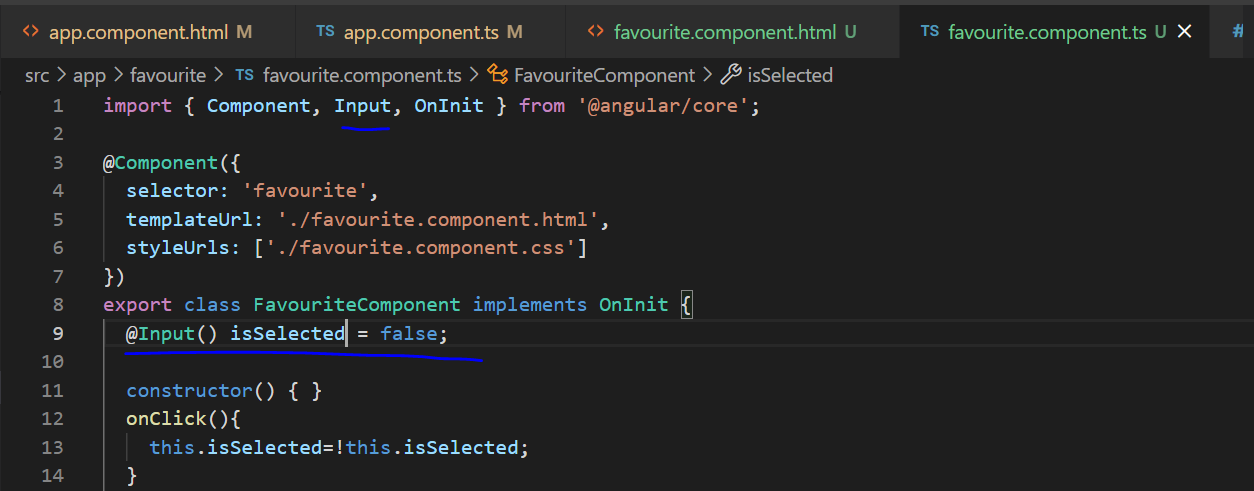
Now what if we want this favourite component to be custom and accept it’s own properties (not dom attributes but the variables like isSelected (and binding this isSelected variable of favorite component to the isSelected variable defined in host component, in this case app.component.ts)) and also handle dom event on this custom elemtn inside the host component (in this case app.component.ts). lets see what happens by adding them.





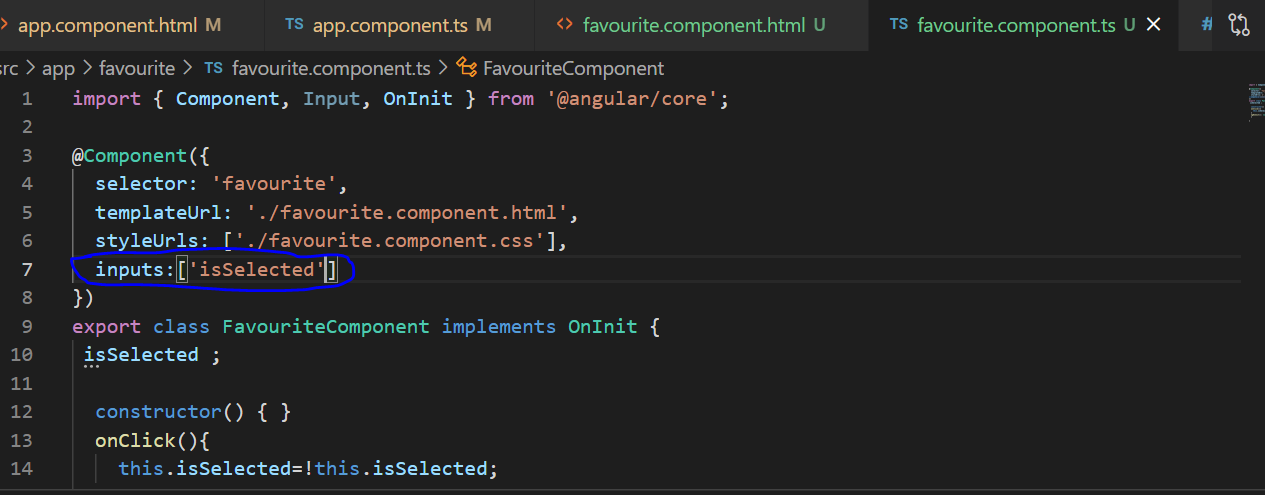
Here it complains that isSelected is not known property since this is not DOM attribute.

So now in order to make this happen we use @Input and @Output decorators.



Now with @Input() we are exposing isSelected property of favourite.component to other components as attribute.

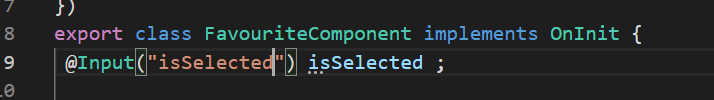
There is another way to expose component properties like below



But this is not recommended because if we change the variable name in code then we also have to change here.

**Aliases**:

For @Input() decorator we can pass alias name



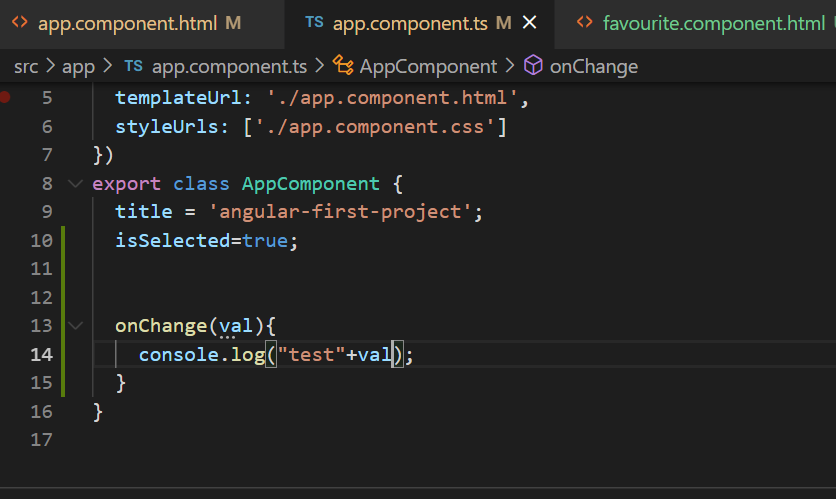
Now even if we change isSelected variable inside this component, where ever else we use iseSelected attribute, we will not have issue.

**@Output()**

This annotation we use to have a custom event on the custom component. In this example we have a custom component of change and a function gets triggered when thing change event happens.

Now to define this change event in original component or child component we annotate the change variable with @Output() decorator and assign it the object of EventEmmiter (new EventEmmiter()).

Now we will have the actual DOM event defined on the child component template as well and using this output variable event we will emit any values or objects and they get to parent component or host component.





We can have aliases for @Output(‘name’) just like @Input()

**Templates:**

We can use these two ways either externally or inline.

templateUrl: ‘./favourite.component.html’,

or

template:’<h1>Hello World</h1>’

**Styles:**

**styleUrls:[“./favourite.component.css”]**

**or**

**styles:[**

**` `**

**]**

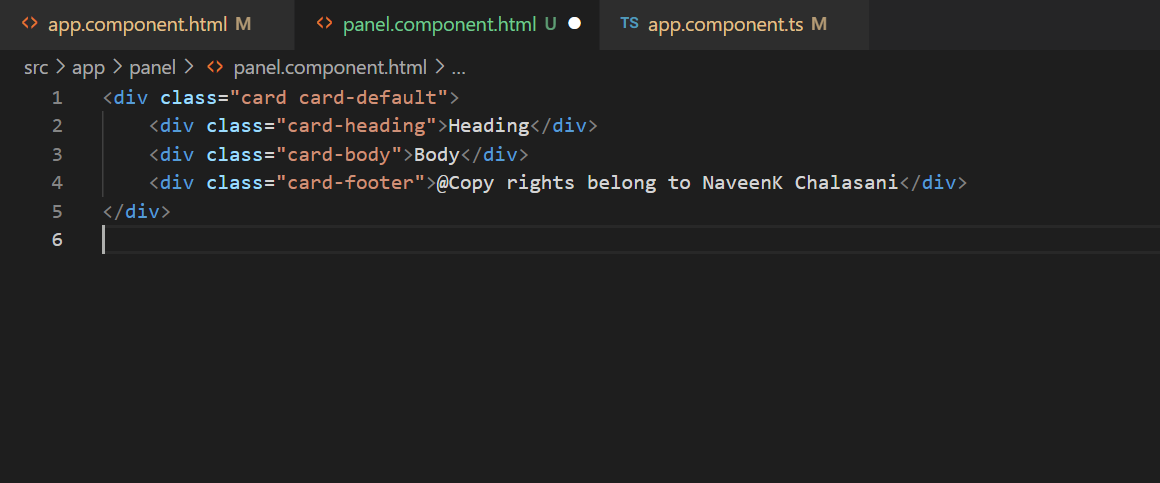
**View encapsulation:**

The styles we define in components are scope to those specific components. Before learning about this we should know **Shadow DOM.** It allows us to apply scoped styles to the elements without bleeding out to the outer world. This is not supported in older browsers.

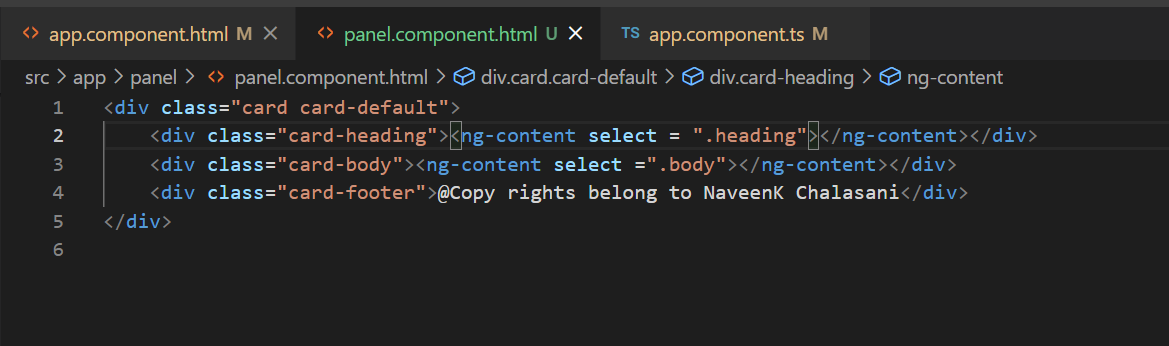
**ngContent:**

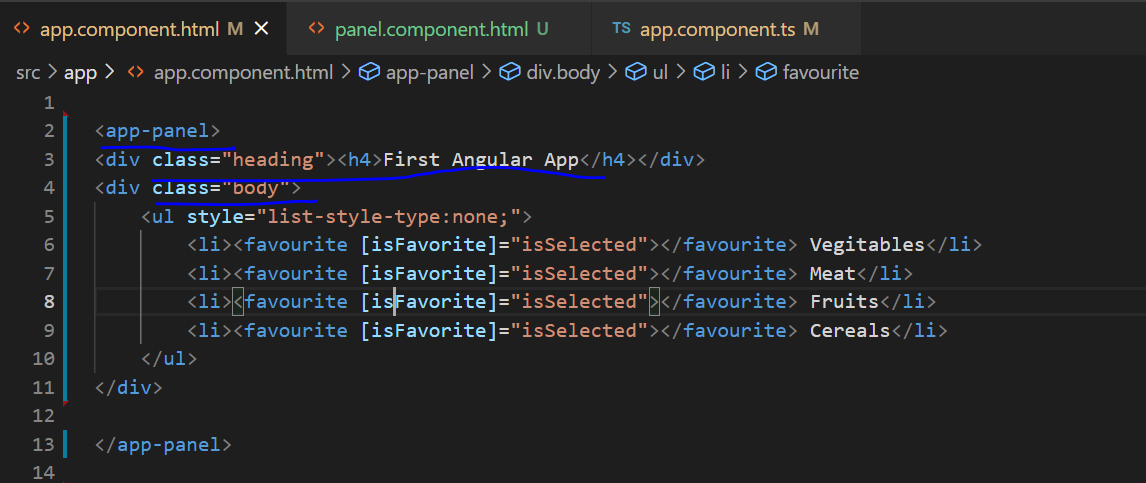
If we have custom components and if we want the users of these custom components or the hosts of these custom components to inject their custom content then we use ng-content.

For example we can have a panel component without ng-content like with hard coded values for heading, body and footer.



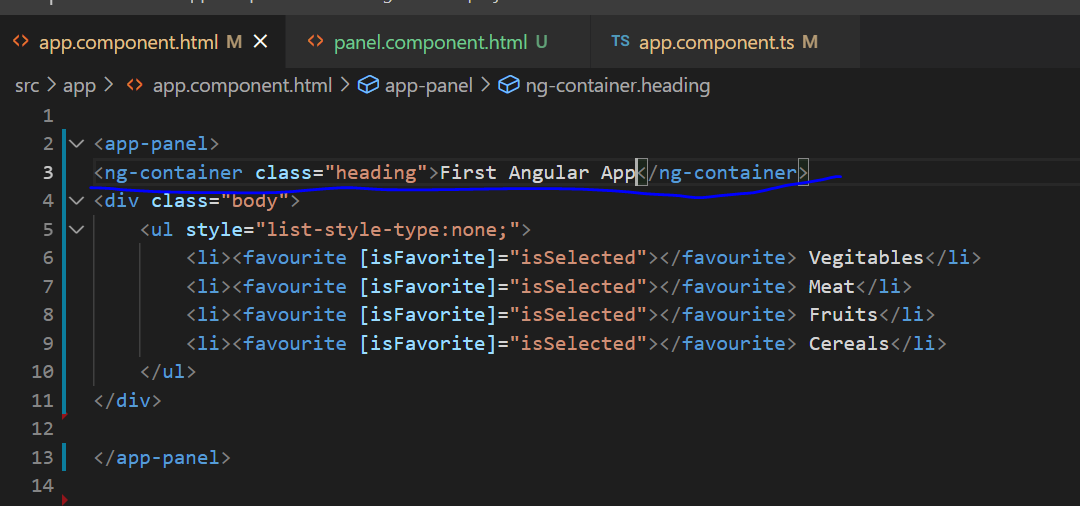
But we don’t want hard coding, we want this panel component to be custom component and any host or parent component should be able to inject it’s own content in this custom component. Using ng-content the code will be like below and we add select statement to provide class name or id in the panel component so that host component would be able to send text or content to appropriate panel component element.





**ngContainer:**

In the above code with ng-content, what happens is at run time, from app component a div or some html content with content in it is injected or will replace the ng-content element of child component. Now if don’t want to inject div or any other html element instead we want to inject directly the text then we use ng-container.



**Directives:**

There are two types of directives

**Structural :** Modify the structure of the DOM. All the structures directives must be prefixed by \*

**Attribute:** Modify the attributes of the DOM

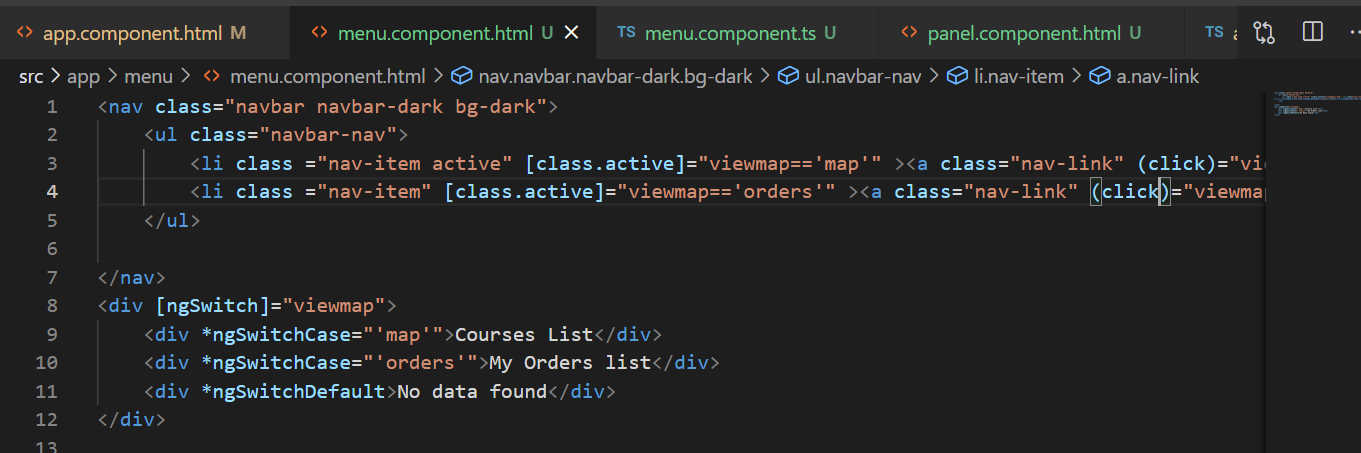
**ngIf**

Sometimes we may to hide or show some part of a page depending on some condition. Here show or hide means it will completely remove the element from the dom or add it to the dom.

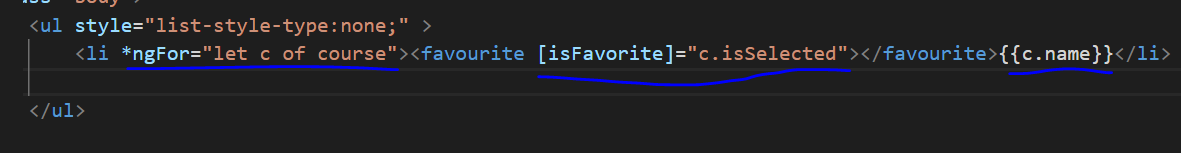


Instead of using ngIf angular directive we can also use hidden attribute of element. But the only difference is ngIf actually removes or add the element to the DOM while hidden attribute does not.

ngSwitchCase

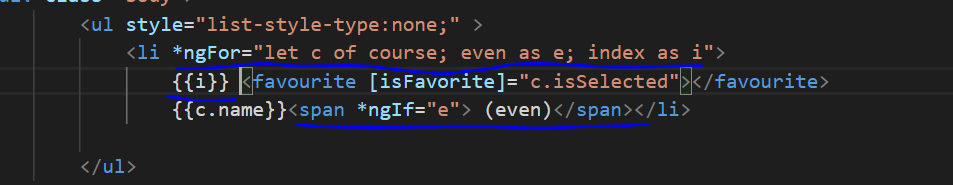


**ngFor:**



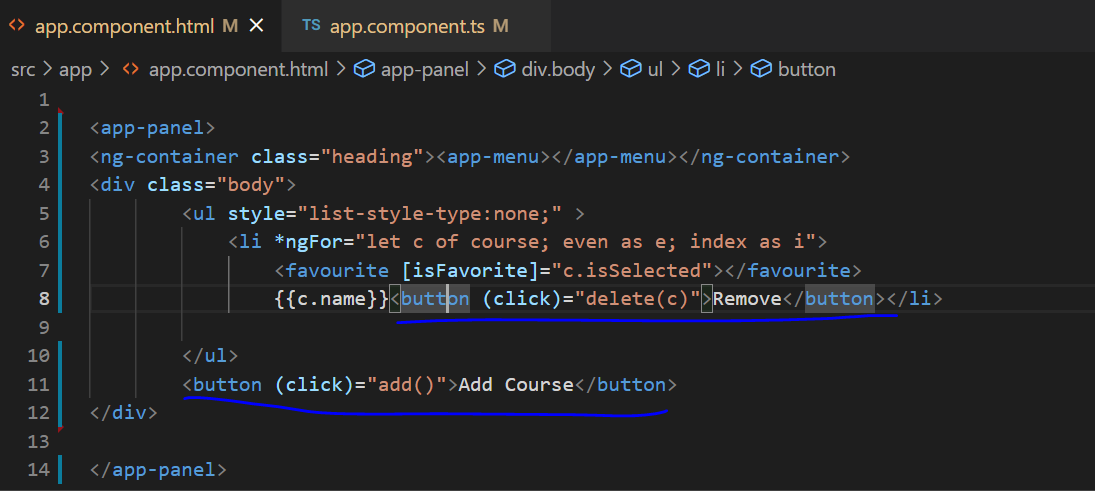
There are bunch of local variables that ngFor exposes like index, even, odd, count, first, last. Except index and count rest are Boolean.

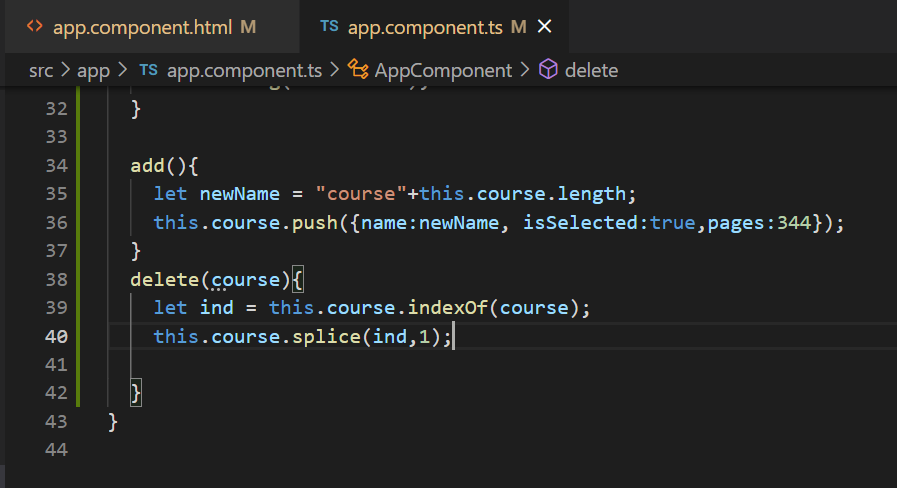
For example:



How ngFor reacts to changes?

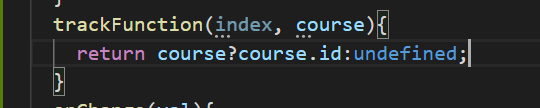
Angular has built in change detection mechanism. When an event is raised in Dom (like a button click), ajax requests happen, or some timer function completes angular checks for changes in the values of a variable and render them using template.

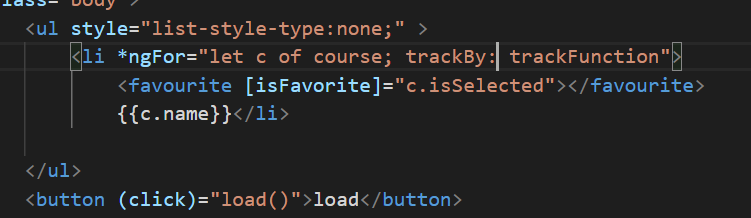




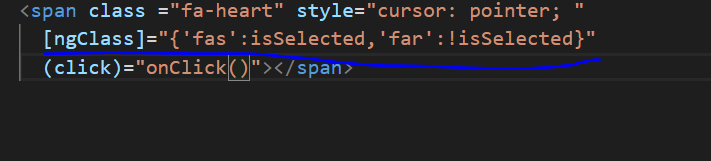
ngFor and trackBy

Say we have a button called Load and when every time we click on that button, we will either make a api call or get the hardcode list of object and load them in the component variable. Since we are using ngFor directive to render them all in the template, angular will take care of constructing the dom tree for us and it will do it everytime we click on load. But if we want angular to only construct the new items and doesn’t construct the ones that already exists then we can use trackBy in ngFor.





ngClass:



**Safe traversal operator:**

This safe traversal operator is denoted by ? symbol. Here using this ? we are checking whether a variable value is present or null and then only we will use it. So basically this makes the variable optional.

We can render an object value once we check if exists using two ways one is \*ngIf and other is ?

**<span \*ngIf=”course.name”></span>**

**Or**

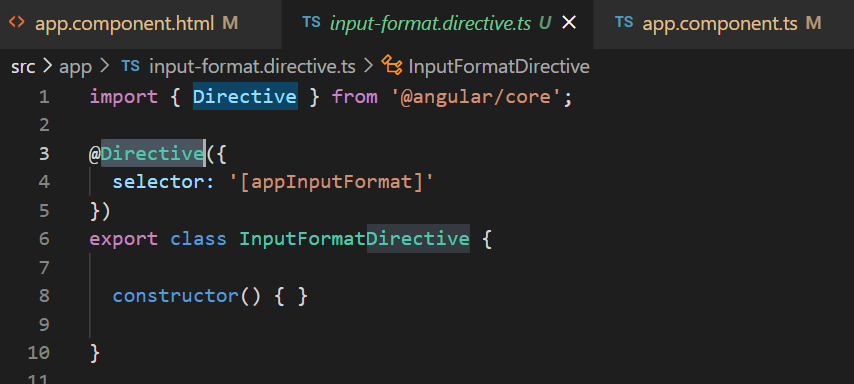
**<span>{{course.name?}} </span>**

**Custom Directives:**

Some times we want control over DOM elements. For example a phone number field where we automatically put hyphens and brackets for American numbers like (318)202-9367. In situations like this we can use custom directives.

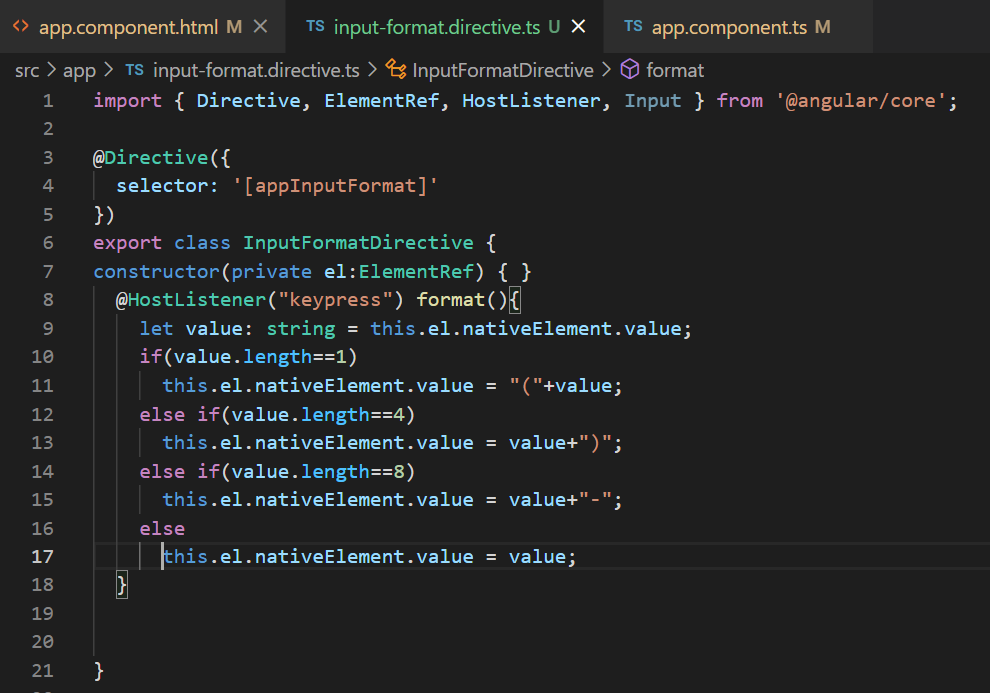
Now we can create a boiler plate code for directive using

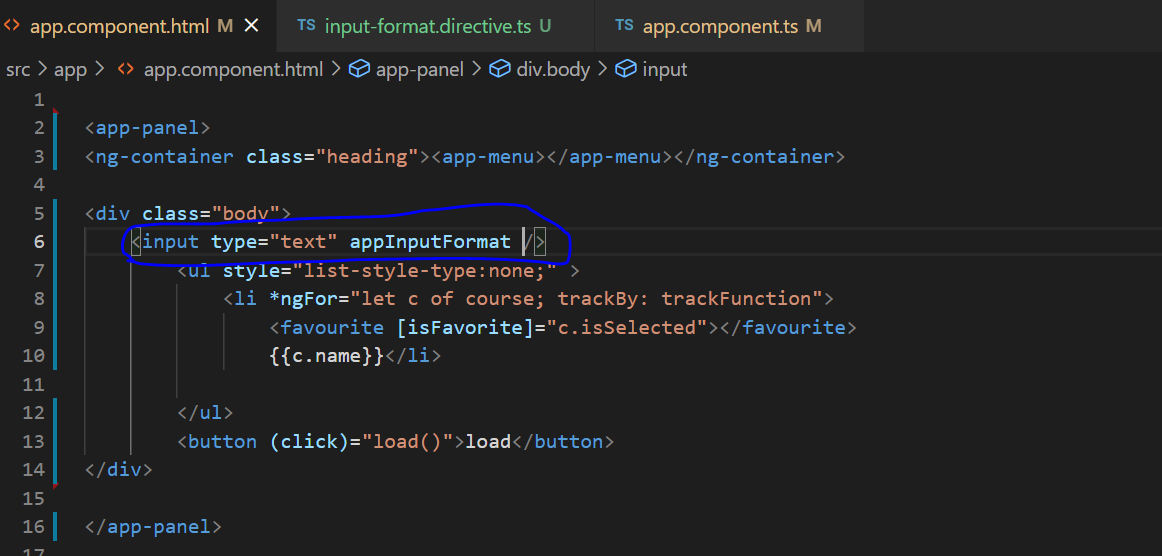
**ng g d input-format**



Here we have to use one decorator function **HostListener**, this allows us to subscribe to events raised from the dom elements in host component or host element that has this directive as attribute.

We also have to import ElementRef service into constructor. This allows us to use angular dom objects and gives us their access.





**Consuming HTTP Services:**

In order to use Http services in our code, first thing we need to do use go to app.module.ts and import HttpClientModule from @angular/common/http. Now in our service class we have to use HttpClient service from @angular/common/http and inject it through constructor.

I said service class, because we always have to use service classes to make http calls. We do this because

1. If components are using service classes to make http calls then it would be easy for us to test them, because all we have to do is mock this service by creating a mock class with mock data.
2. We always have to abide by the rule of object oriented programming which is “Separation of concerns” where we need to keep only component related code in component and outsource the http related calls to services. Because of this we can reuse these services in any of the component and if there are any changes to url to be done we just have to do it service class.

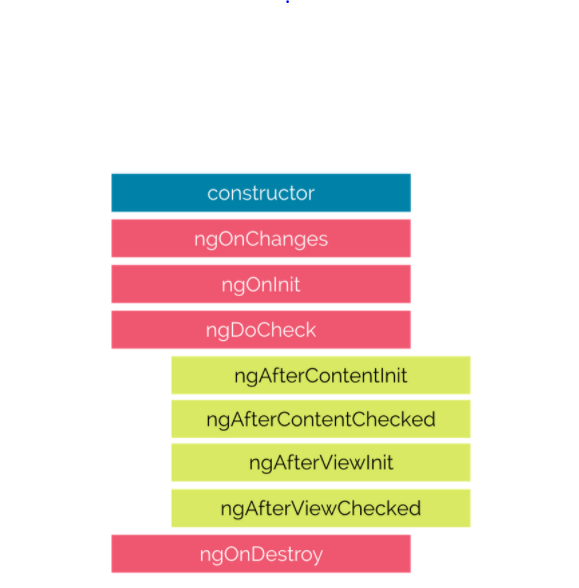
Now in order to generate service class it is always recommended to create a separate directory called services and run this command in the folder **ng g s service-name**

Now lets discuss about various http methods like GET, POST, PUT, PATCH, DELETE

1. GET method is used to get data from the backend server.
2. POST method is used to post data and create a new record or resource in the backend.
3. PUT is used to update an exiting resource.
4. PATCH is used to partially update an exiting resource which means we don’t have to send all the details of the record to update it, we just send the ones that needs to be updated in a record.
5. DELETE method is used to delete a resource in the backend server.

Now before talking about services we should also talk about angular **lifecycle hooks**. These are component methods that we can use to run a piece code during the lifecycle of a component.

A component instance has a lifecycle that starts when Angular instantiates the component class and renders the component view along with its child views. The lifecycle continues with change detection, as Angular checks to see when data-bound properties change, and updates both the view and the component instance as needed. The lifecycle ends when Angular destroys the component instance and removes its rendered template from the DOM. Directives have a similar lifecycle, as Angular creates, updates, and destroys instances in the course of execution.



**constructor**

This is invoked when Angular creates a component or directive by calling new on the class.

**ngOnChanges**

Invoked every time there is a change in one of the input properties of the component.

**ngOnInit**

Invoked when given component has been initialized.  
This hook is only called once after the first ngOnChanges

**ngDoCheck**

Invoked when the change detector of the given component is invoked. It allows us to implement our own change detection algorithm for the given component.

**ngOnDestroy**

This method will be invoked just before Angular destroys the component.  
Use this hook to unsubscribe observables and detach event handlers to avoid memory leaks.

**ngAfterContentInit**

Invoked after Angular performs any content projection into the component’s view.

**ngAfterContentChecked**

Invoked each time the content of the given component has been checked by the change detection mechanism of Angular.

**ngAfterViewInit**

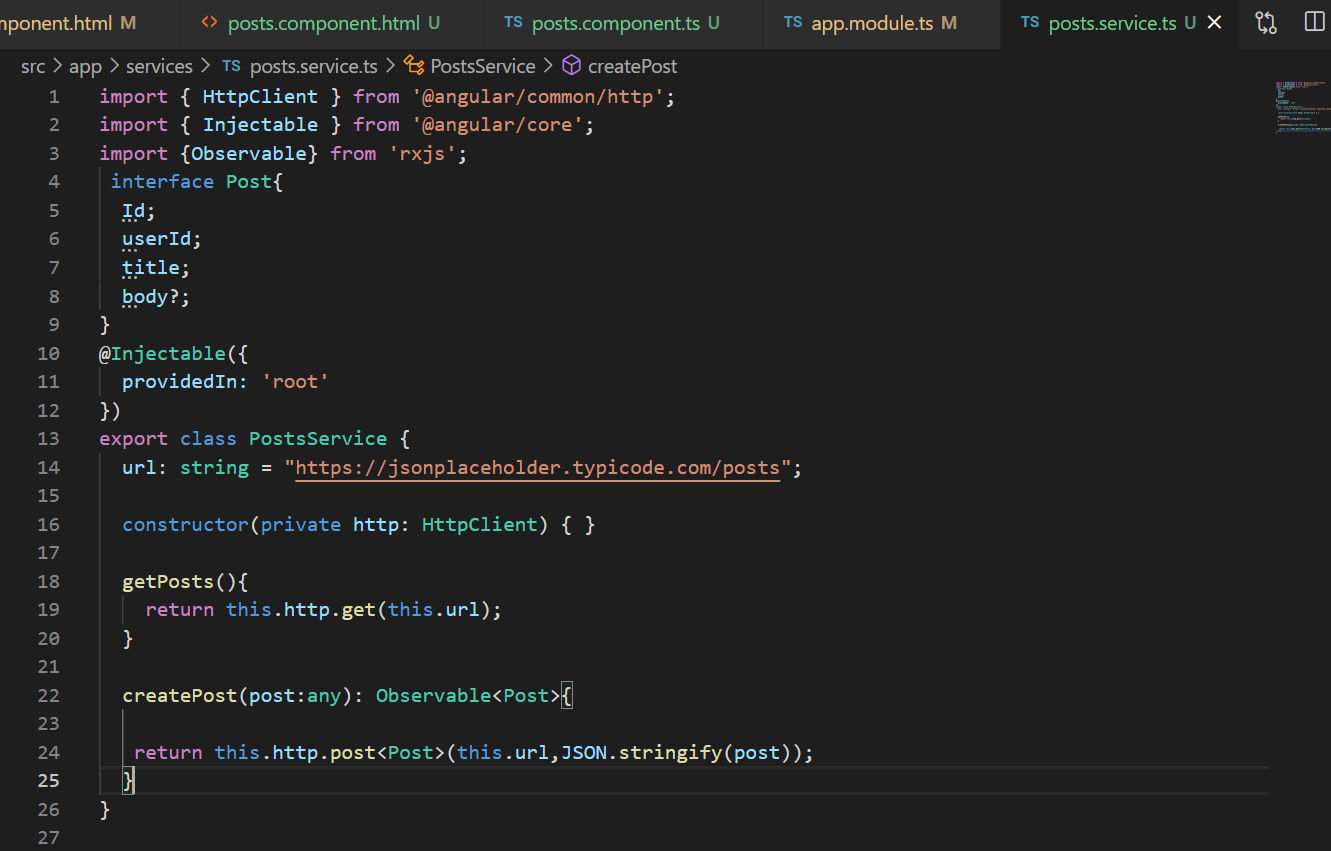
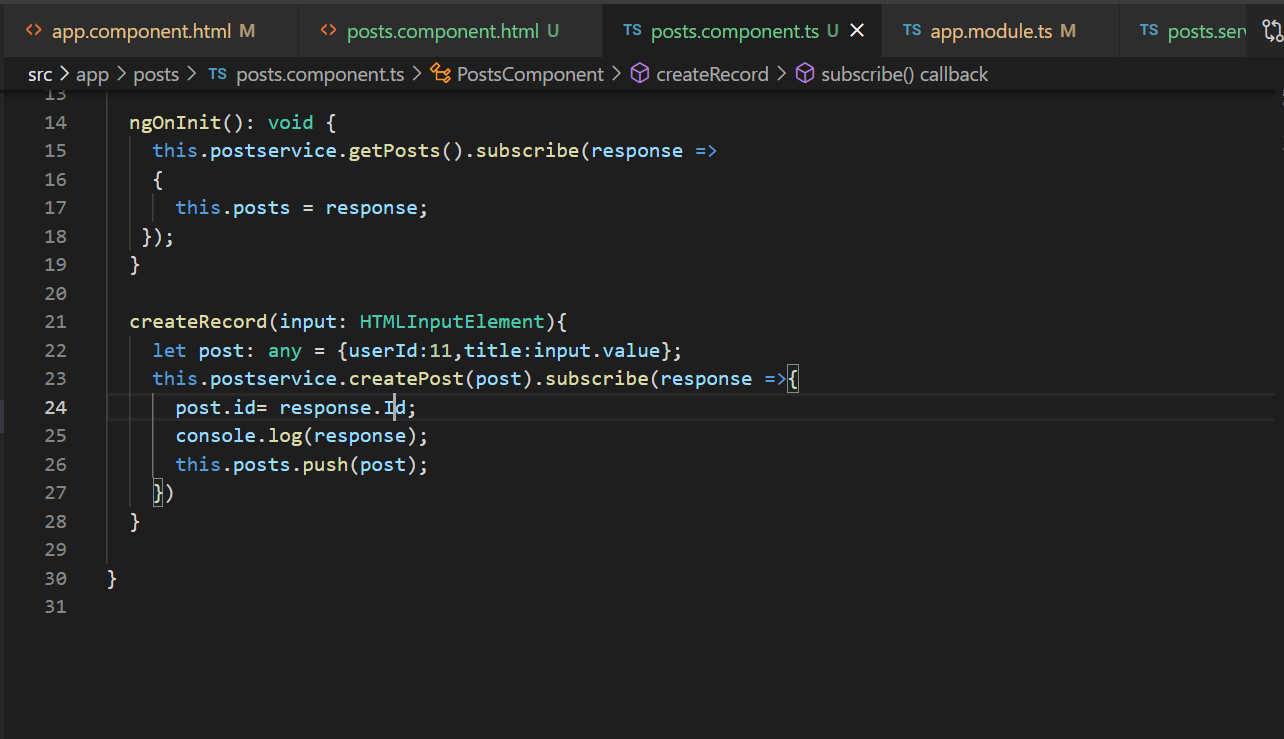
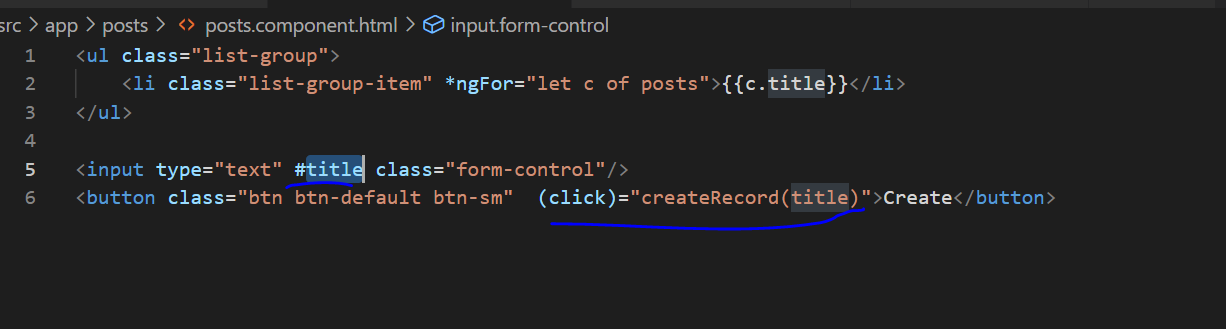
Invoked when the component’s view has been fully initialized.

**ngAfterViewChecked**

Invoked each time the view of the given component has been checked by the change detection mechanism of Angular.

Now we should talk about Observable. Observable is used when we are performing asynchronous or non-blocking operations where we don’t want our code to wait till the operation is completed.

Now we are talking about observable because the response from HttpClient service get(), post(),put(),patch(),delete() are of Observable return type. Observable doesn’t start until we subscribe to it. So we have a subscribe() method in the return object of HttpClient. In order to capture the response from subscriber() method we need to pass response as input param to lambda function and then in the code block we have to process the reponse. HttpClient response will be by default JSON format and we can assign it to an instance variable to access it in the component’s template.

**Error Handling:**

Now Lets talke about handling errors. Not all are ideal screnarios where we get response from the backend server. Sometimes we may have to handle unexpected error scenarios as well.

Two types of erros, unexpected and expected.

**Unexpected**

1. Server is offline: when the server is offline when client sends requests
2. Network is down: though the server is online, client cannot reach it.
3. Unhandled exceptions

**Expected**

1. Not found errors (404): if resource is not found in server that client wants then not found error.
2. Bad requests (400): for example registration of user, and if user already exists then server reponsds with bad request.

Handling Unexpected Errors:

Handling Expected Errors:

Throwing appilcation specific errors:

Handling Bad requests errors:

Importing Observable operators and factory methods:

Global Error Handling:

Extracting reusable error handling:

Extracting resuable data source:

The map operator:

Opimistic vs pecimistic:

Observable vs Promise:

**Routing and Navigation**