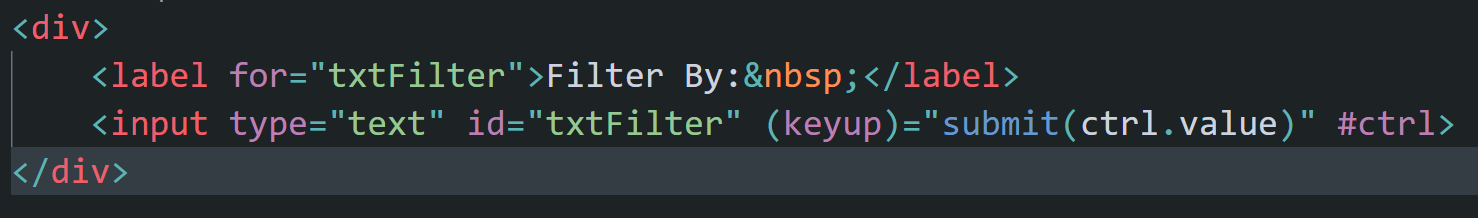
**Incomplete codes:**

1. Display date string in calendar and also fetch the selected date [refer the updated code of EditProductComponent]
2. Get the selected image file and convert the image data into blob [refer the updated code of EditProductComponent]

The topics I couldn’t complete properly

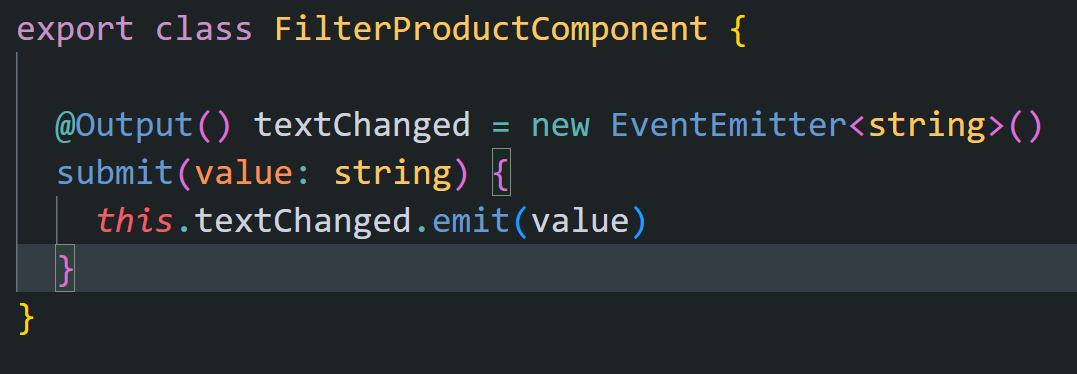
1. **Passing data from a nested component to its immediate parent component:**
2. Created FilterProductComponent class [app/modules/products/components/filter-product]

Design:

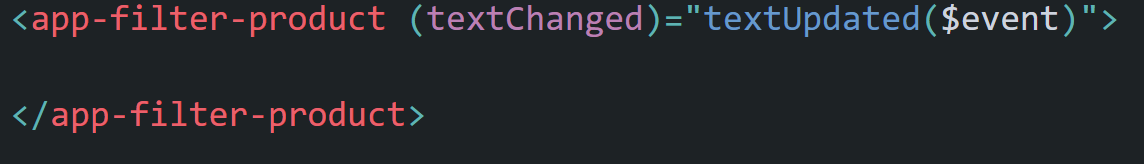


This component on **keyup** event of the **input** type, calls a method in the component code “**submit()**” and passes the value of the text box as a parameter

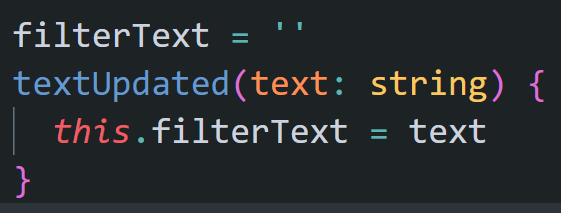
1. Following is the component code:



1. Since the component has to pass the data to its parent component (ProductListComponent), this can be achieved by firing an event from this component, so that the parent can handle the same and attach an event handler (a method in the parent component) to the event. Hence a property has been created which refers to an EventEmitter<T> object (this is used to create and fire a custom event). The kind of data you want to emit, when the event is fired, decides the type of T. But the property must be decorated with @Output() decorator (from @angular/core package), so that the property can be accessed and event bound in the parent component, as you remember, any property of a component code can be used only in that component’s view (design).
2. Now, following is the ProductListComponent design



1. The textUpdated() method is a method in the parent component code, as shown below. The value emitted, when the event was fired from nested (FilterProductComponent) component code, is bypassed to the event handler attached (textUpdated() method), by using a special syntax, **$event. This represents the event related argument.**



1. The value passed to the method from the design of the ProductListComponent is finally assigned to a property, **filterText,** in the component code, which will be used to filter the products, which have this as a sub-string, in their product names.

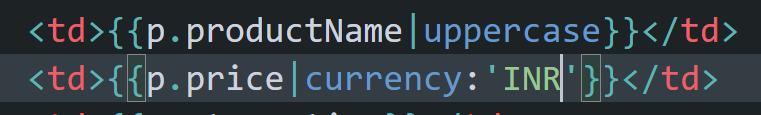
**Conclusion:** *In order to pass data from nested to its parent component* ***event-binding*** *technique is used, where nested component emits a value by firing an event, and the parent component handles that event by attaching an event handler from parent component code, to which finally, the emitted value (from nested component) is passed.*

**D. Pipe:**

Pipes are simple functions to use in template expressions to accept an input value and return a transformed value. Pipes are useful because you can use them throughout your application, while only declaring each pipe once. [Definition from angular.io website]

Simply put this way, Pipes are used for transformation of data.

Example: code from ProductListComponent deisgn



Here, **uppercase and currency** two built-in pipes. The syntax is as follows, for any pipe:

***Value-to-be-transformed | pipe-name: additional-arguments***

If you look at the declarative syntax, on the left side of the pipe name, there is a value. This is the value that needs to be converted to something. Such as, the productName value needs to be converted to uppercase. Hence it is p.productName|uppercase.

Again, suppose, I want the price of a product to be displayed with currency symbol. Hence the syntax is: p.price | currency.

But then, you need also to mention which currency symbol to use. Hence that additional value has to be passed after the pipe name, separated by colon (:).

Hence, the complete syntax is: p.price | currency: ’currency-symbol-name’ . Currency symbol value can be ‘INR’ or ‘USD’ etc.

Behind the scene, the pipe-name represents a class, decorated with @Pipe() decorator method, containing a special method, named, **transform.** The signature of the method is as follows:

***transform(value: any, …args: any[]):any { //code }***

When the pipe is used in the design, through declarative syntax, an object of pipe class is created and then the transform() method is called, where, as the first argument (value:any), the value to be transformed (mentioned on the left side of the pipe) is passed. The value mentioned on the right side of the pipe name (separated by colon), is passed to the second argument (args:any[] - an array which can directly accept parameters).

The second argument is known as parameter array (just like an array declared with **params** keyword in C#, in function argument). To declare a parameter array as part of a function arguments, in JS/TS, a new feature is used, known as **rest parameter, where an array (args:any[]) is declared with 3 dots (…).**

Actually, you can pass more than one argument to this array. In that case, the declarative syntax of pipe in the component design should be as follows:

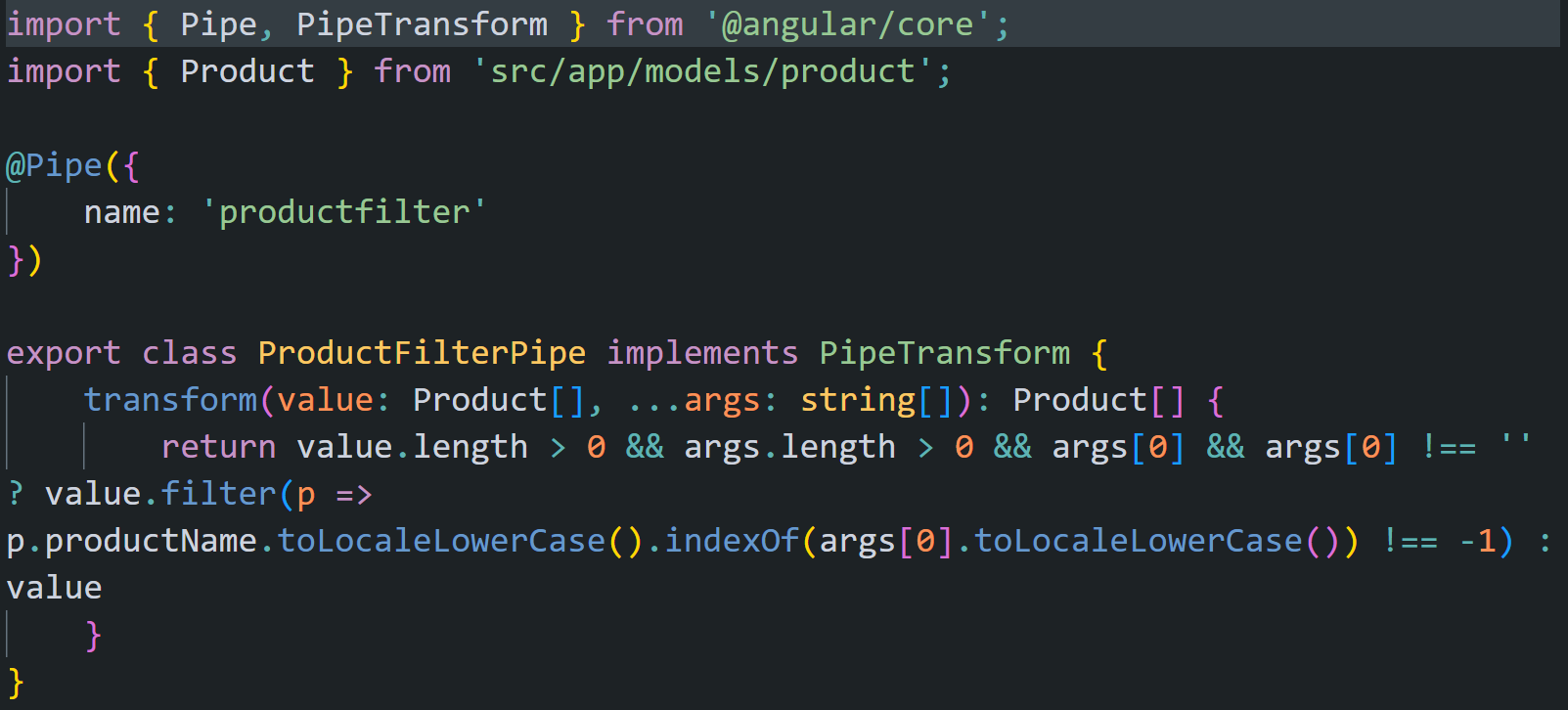
**Value-to-be-transformed | pipe-name: arg1: arg2: arg3: …..**

All these arguments will be passed to the second argument in the tranfrom() method

**Create Custom Pipe to Filter the Products Displayed in the ProductListComponent design:**

In our ProductListComponent, we want to filter the products displayed in the design by a filter text (as stored in the **filterText** property in the ProductListComponent – This value is entered in a text box, as part of the FilterProductComponent design and the value has been passed from nested component, FilterProductComponent, to the parent component, ProdictListComponent already – *refer the previous segment for the same*). This value will be used to filter the products. Hence, a pipe needs to be created (with a name, say, **productfilter**).

1. Create ProductFilterPipe class in a file [app/modules/products/pipes/product-filter.pipe.ts]:
2. Create the class ProductFilterPipe and decorate with @Pipe() decorator
3. The **name** property in the metadata object (like **selector** property in case of a component), passed to Pipe() decorator function, is used to expose the pipe class. **This is the name of the pipe to be used in any design.**
4. The pipe class must have a method transform(value: any, …args: any[]):any { //code }. In order to get the method, implement the interface, PipeTransform (from @angular/core) in the class
5. Now, write the logic in the method
6. Here in our case, to the transform() method, one must an array of product object (products) as the first parameter and the text (filterText) to be used to filter the products by their product name, as the second parameter
7. Hence, the code looks like the following

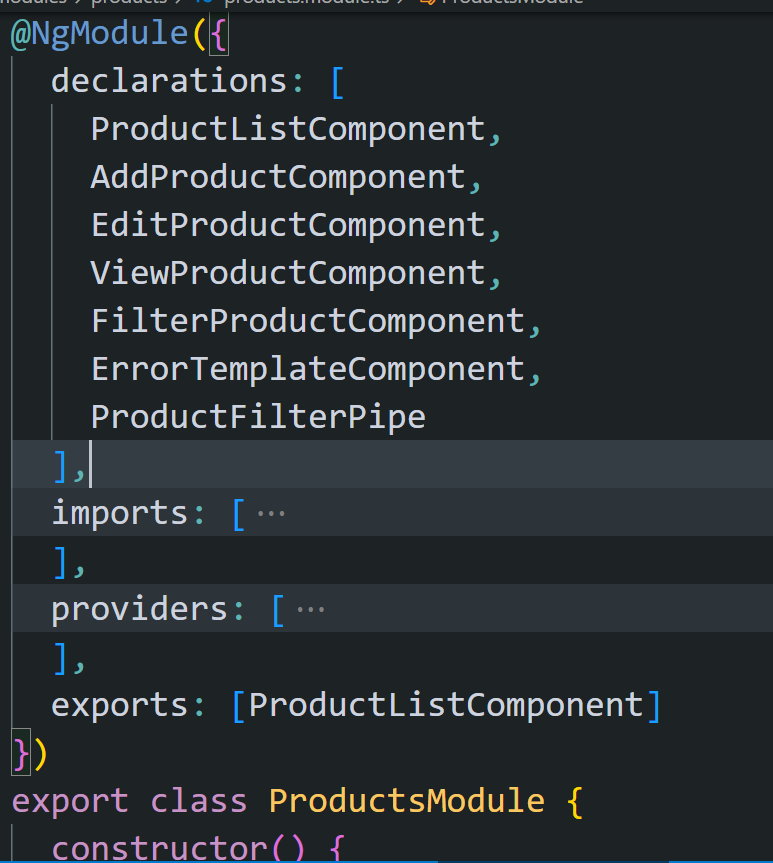


Since, you are passing just one value (the filter text) as part of the additional arguments, to the pipe, the value is stored in the 2nd argument (…args: any[]) and that too in the 1st index of the array

My logic checks whether the products array is having at least one element or not, the args array is defined and has a value at least and that value is not an empty string. If all the conditions succed, then we filter the products array and return the filtered array. Otherwise, we return the originally passed array as it is, without any filtration.

That is the return type of the method is Product[] (array of products)

1. Now, register the name of the pipe in a module file, in our case, ProductsModule class, in the declarations array (I just want to use this pipe within the assets of ProductsModule)



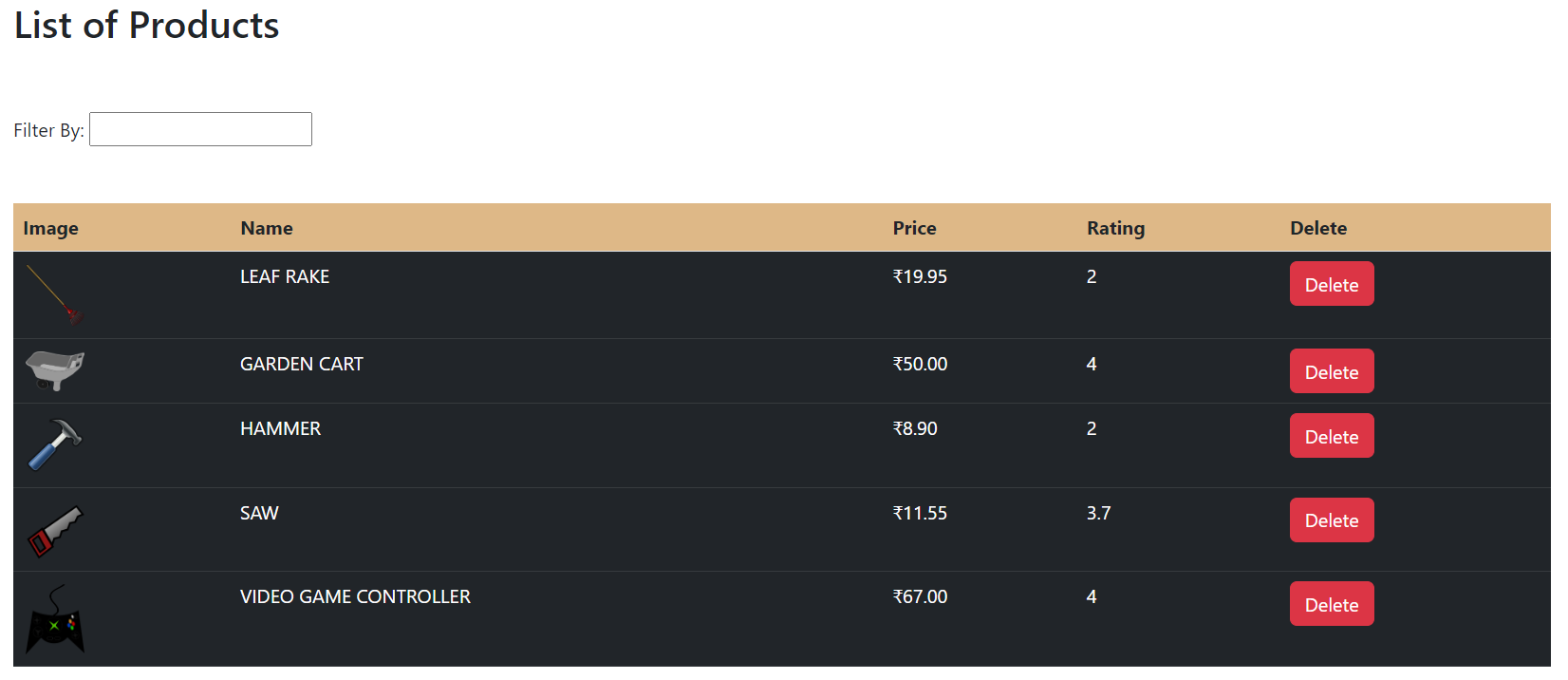
1. Use the pipe in a component:

Here, we are going to use this pipe in ProductListComponent design. Hence, he design of the ProductListComponent is as follows:



Note: using round brackets (), is not mandatory. I just wanted to make a point here and that is: first the products array will be passed to the transform method as the first parameter and the filterText as the 2nd parameter and then the returned filtered array will be iterated using ngForOf directive.

No filtration:



With filtration: (as soon as you enter a letter in the text box, say, l)

