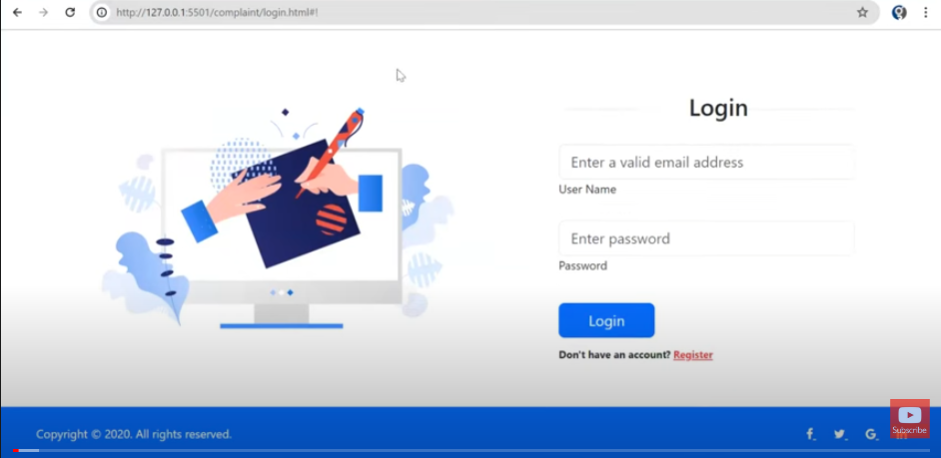
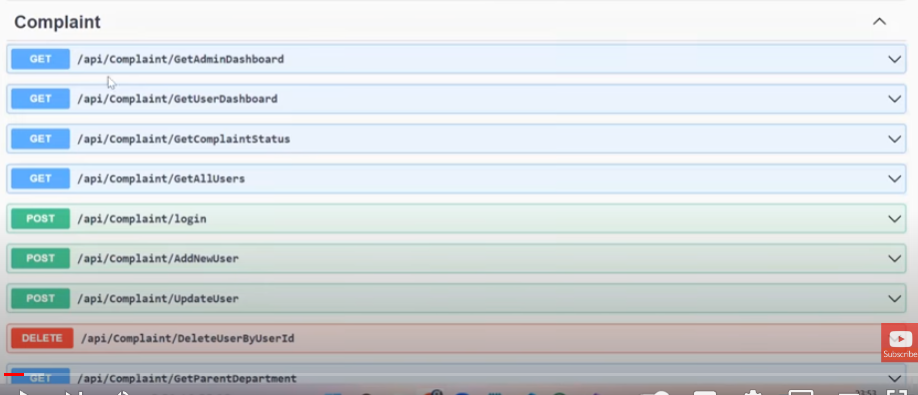
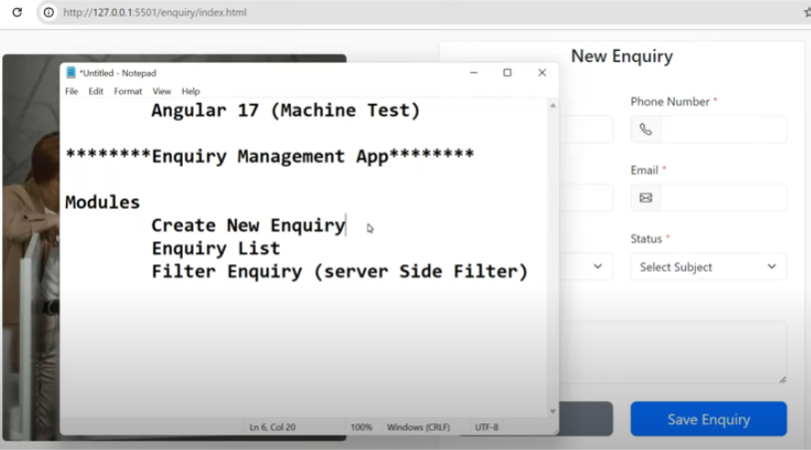
A screenshot of a computer

Description automatically generatedA computer screen with a computer and a pen

Description automatically generatedComplaint Management System

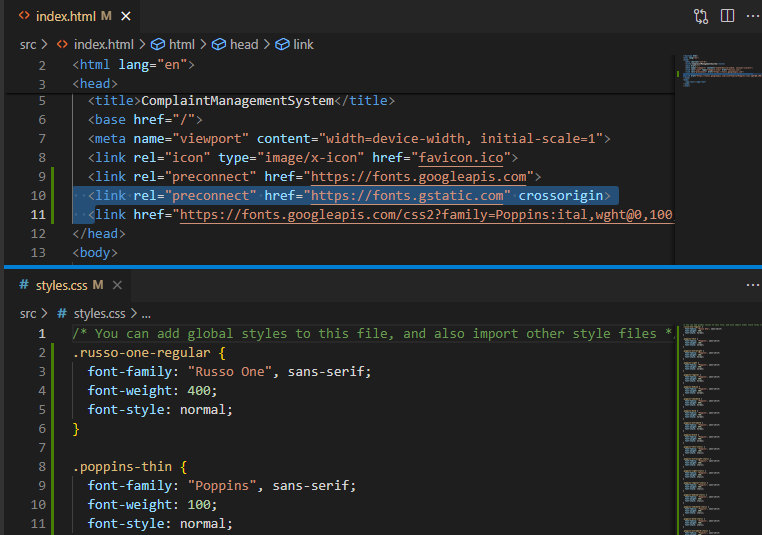
so, we will be focusing on on authentication, list of complaints, a form for new complaints and then a dashboard to give some general overview of all complaints.

A screenshot of a computer

Description automatically generated

NB: list complaints and dashboard is going to look and feel different for both user and admin. Once we are done with this, we will be looking at an [enquiry management crud application](https://www.youtube.com/watch?v=KHiHa4kxD4g).

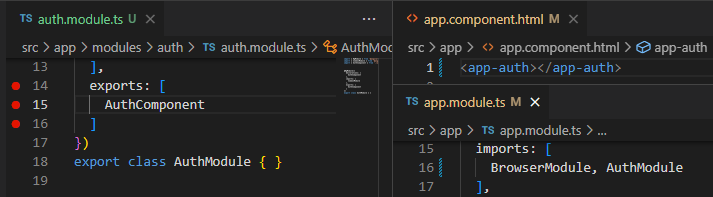
# Authentication Module

got a login template from figma, tweaked it a bit to suit our taste and we will start building.

Got fonts from google fonts and did the basic setup.



A screen shot of a computer

Description automatically generatedWe are going to have login and registration in our auth component, with toggle by Boolean and if statement.

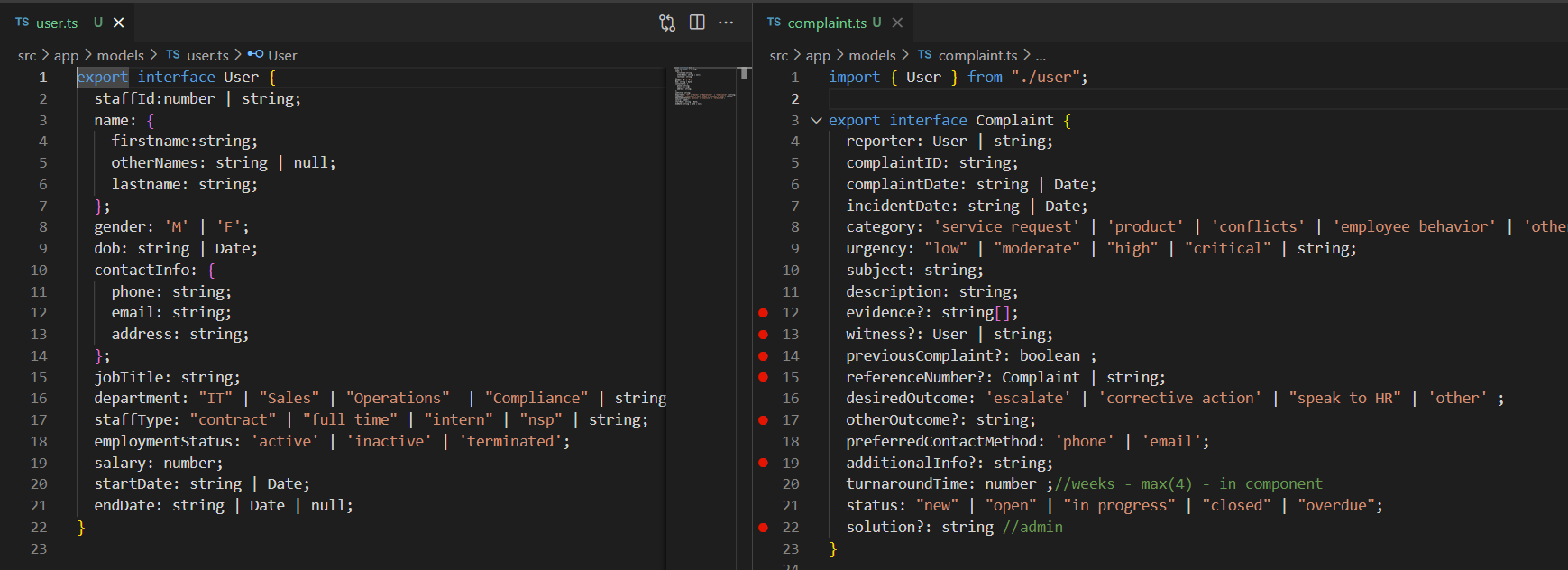
# Models

In here, we are going to create an interface for our user and complaints then configure json webserver for queries.

## Interfaces

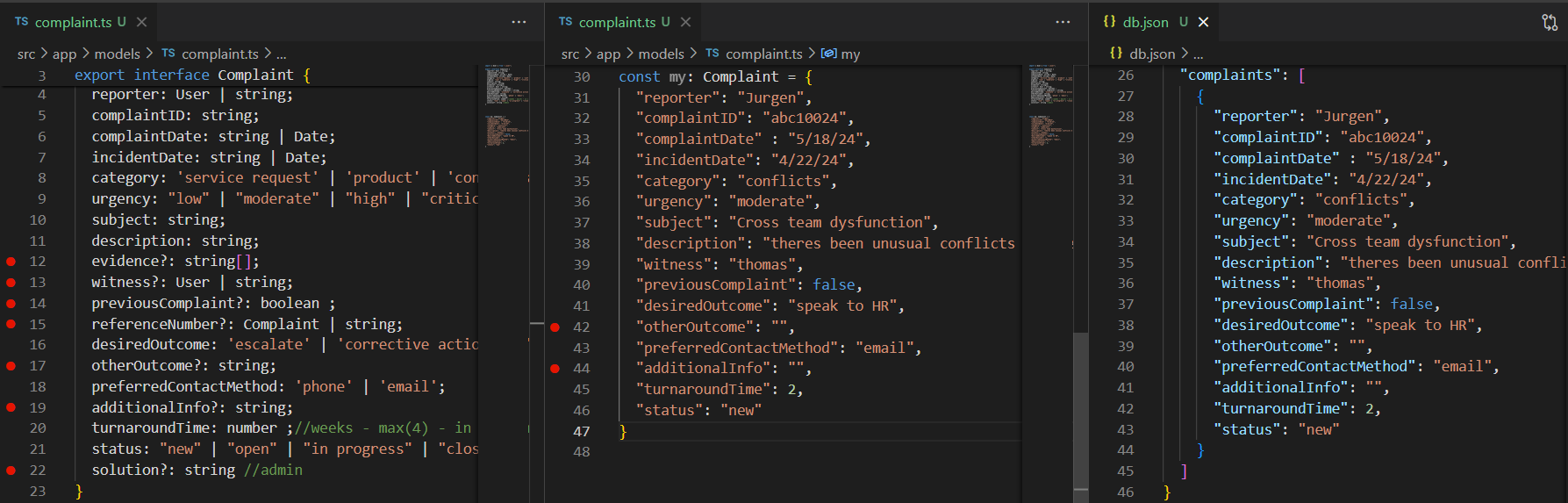
Below is a snippet of our interfaces: we might need to revisit this some other time.

User interface might need to have a password field for authentication purposes.

Complaints interface also has a few fields to be provided within the component and not by the reporter.

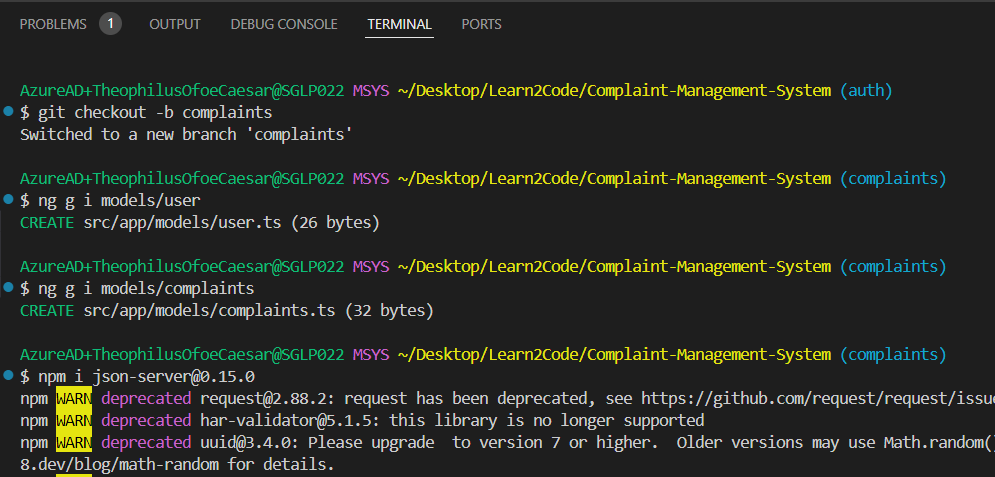
## Optional Operator (?:)

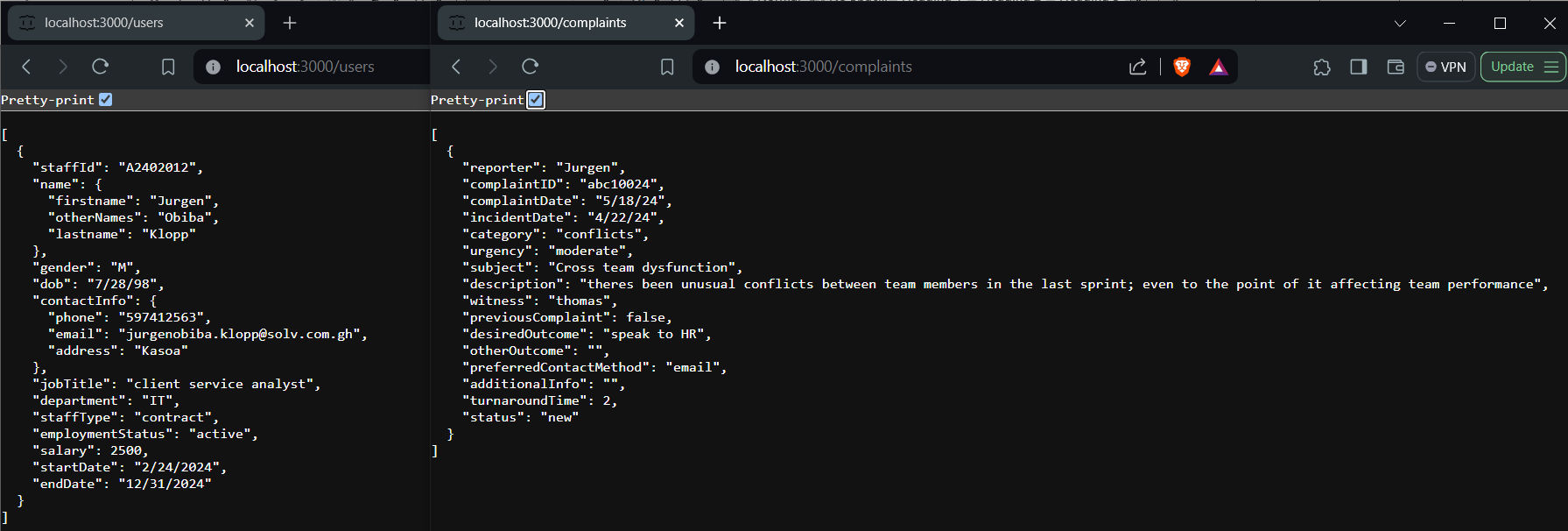
The optional operator(?:) used in our interface basically allows us set properties of objects as optional. In the breakpoint marked lines of our Complaint interface in the snippet above, we set those properties to optional so the form being field could lack those fields and work fine all the same.

 in snippet below, you will notice some fields are lacking in our instance within our db.json.

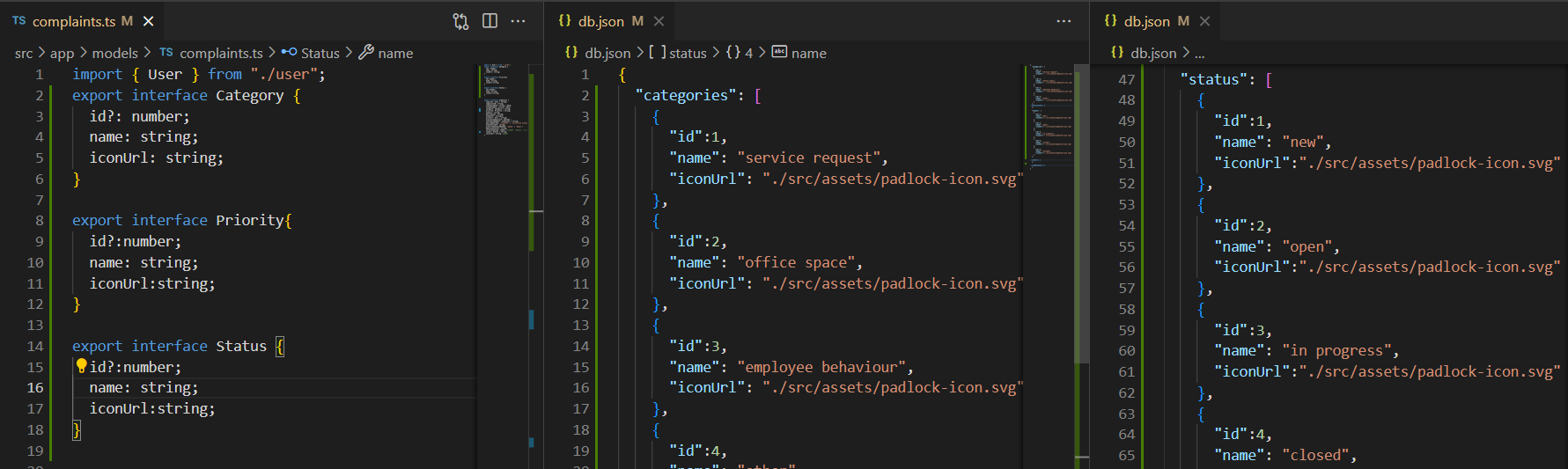
Went as far as to add some optional fields set to an empty string to an object inheriting from our interface but no errors. Success!

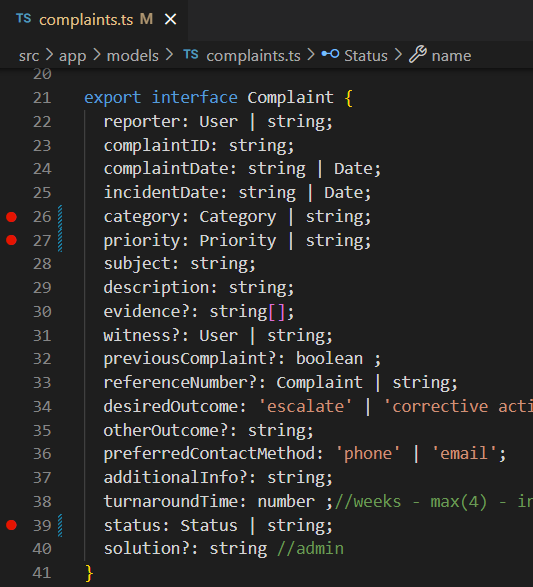
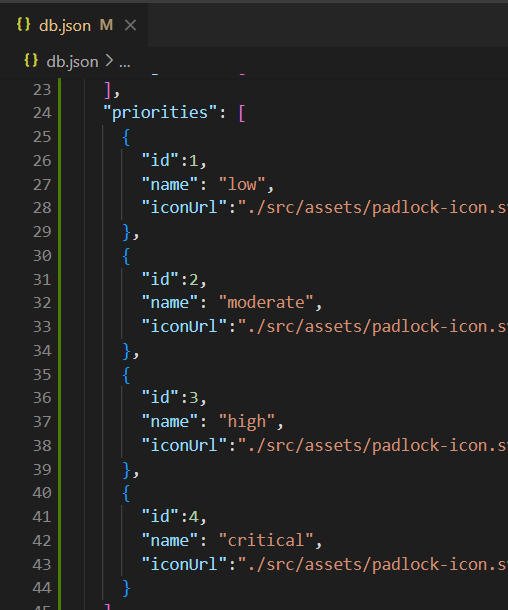
## Json server

Successfully installed version 0.15.0 for our version of angular.



## More models

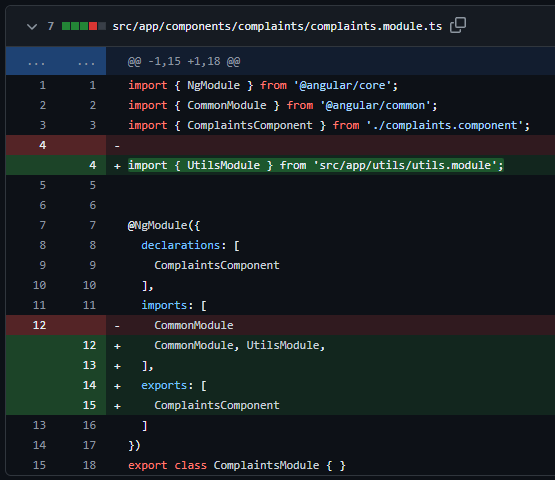
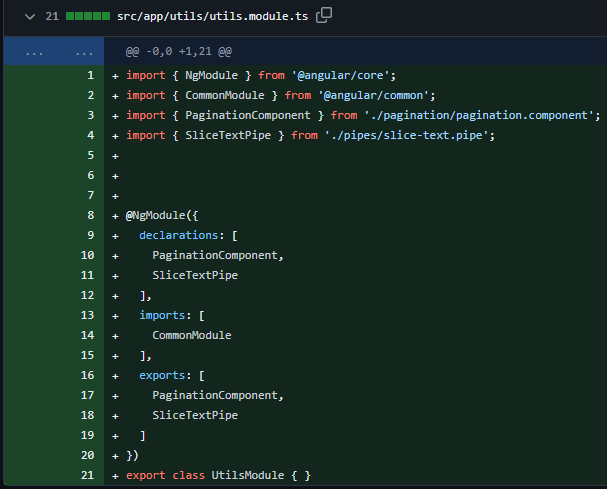
For the purpose of our dashboard cards, we added a few interfaces to our models: Category, Priority & status.

NB: we added our various options to our db.json file.

We have also altered our complaint interface’s properties to be our new interfaces declared. There are 2 other’s we might later consider for interfaces.

The altered props we give an alternative type of string for dev purposes.

# Complaints & Utils Modules.

We will be querying complaints from json server, but we are looking to apply pagination and slicing – to truncate our complaints for reading, hence the utils module.

We have two modules – complaints and utils; the complaints is pretty much straightforward. Utilities on the other side is going to handle some things that will cut across all other modules where necessary. Utilities is going to handle our pagination and slice – a custom directive.

A screenshot of a computer

Description automatically generatedthis is our complaints service, where we have just two methods to begin with.

The first is seeking to paginate the queried data.

the method takes two parameters as arguments.

These arguments are assigned to a variable in line 15.

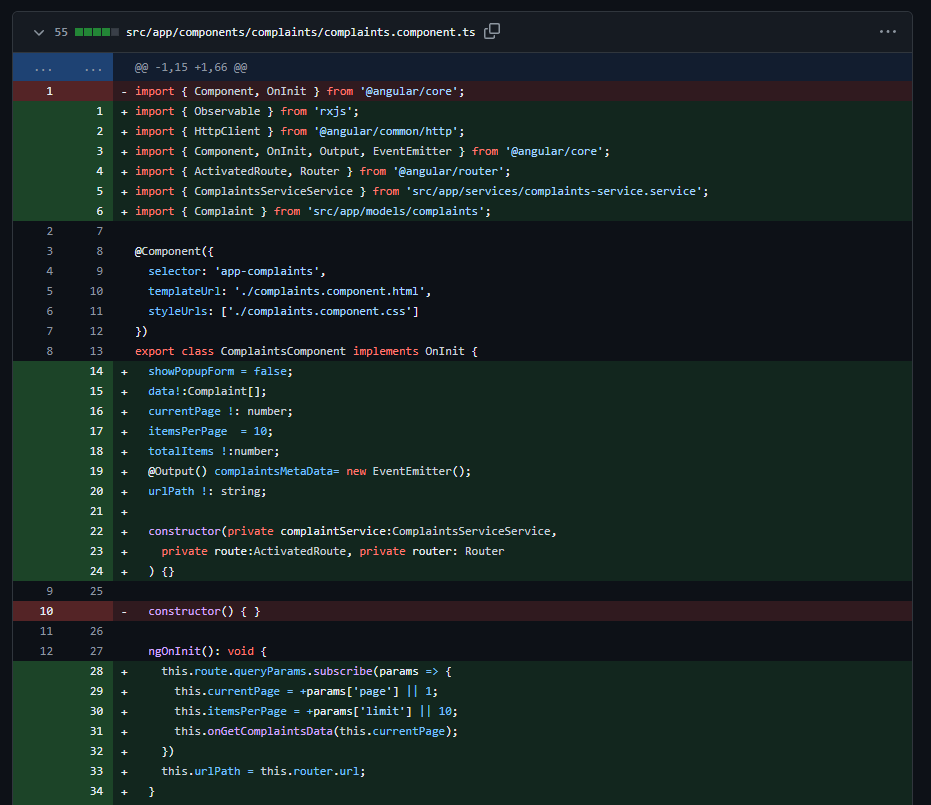
The HttpParams is a class that allows us manipulate URL parameters for HTTP requests.

The set method is used to add or update a query parameter. Its called or invoked by chaining based on as much params your url might have.

The set method takes two arguments – the first is the key and the latter, the value. NB: you must ensure you pass what the endpoint you are calling with the parameters will be expecting.

Line 16 simply passes the params to our get method and voila, we have paginated data.

A screenshot of a computer program

Description automatically generated**OnInit** in lines 28 to 32, primarily has us assign values to properties by getting values from our URLs in the browser’s search bar. We then pass the values to our getComplaintsData method in line 31.

We could only get the URL params with the help of Activated route which was imported in line 4.

1. Lines 39 to 44 define our getComplaintsData method. In there we leverage dependency injection to subscribe to Observable returning methods defined in the complaints service – getComplaintsSet() which returns a set of complaints by taking the page number and the items per page as parameters.

We save the response in an array and then run 2 additional methods, update query params and get complaints metadata – which simply makes a call to our get all complaints method and assigns a count value to one of the properties.

1. Update query params method in lines 54 to 60 take two parameters with which we are going to update our URL. We then invoke the navigate method from our router class we had imported in line 4.

The navigate method first takes an empty array as an argument, which is supposed to contain url fragments as elements of the array. An empty array means that is our parameters that are going to be updated.

The second property – extras, is an object with properties like relativeTo and queryParams.

relativeTo is assigned, this route (an instance of activatedRoute i.e. the current activated route)

we then update the url parameters with the parameters passed to our function earlier

line 58 assigns a value of merge to the params handling property simply to either add or update the params if they already exist

## Pagination Component

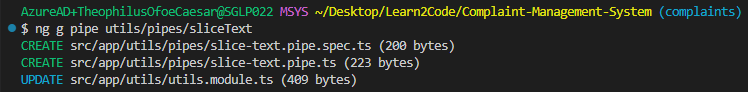
A computer screen shot of a program code

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generated

## Pipe



A screen shot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generated

A screen shot of a computer program

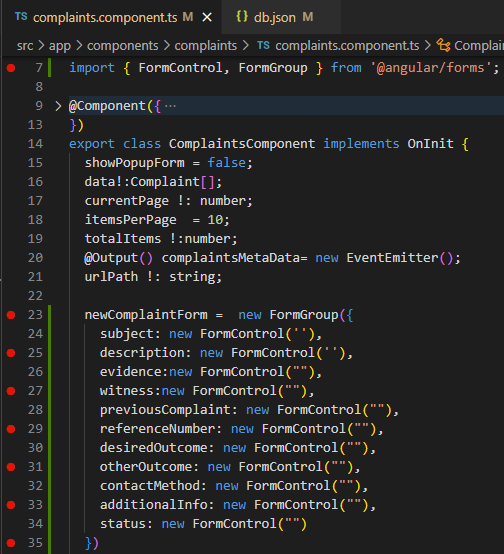
Description automatically generated

# HTTP Client

## Post method

A screen shot of a computer

Description automatically generatedFirst create a method that returns a post request with our http client injectable, which takes two arguments, the api url and the object you intend to post.

We are going to need a form to get the data we intend to post into our db; which will be reactive forms, we had to import form group and form control into our component, with the reactive forms module as a prerequisite.

A screen shot of a computer screen

Description automatically generatedWe instantiate our form group in lines 23 to 35 of our component. With all our form fields being instances of our form control.

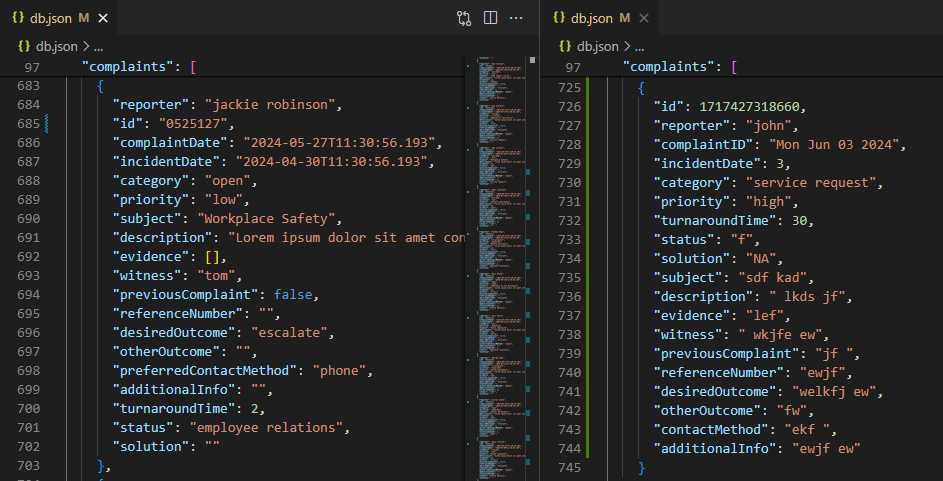
In our template, we bind our form to the formGroup instance we created in our template. The input fields are also bound to the form control instances in our component.

A screen shot of a computer program

Description automatically generatedWe get a button to trigger a function which is going to trigger the post method in our service file.

Since our form is not getting all fields of our interface, we are going to update the data received in the function triggered on submit before we invoke the post method offered by our service.

The end product is captured in the snippet of our db.json below.



## Put or Patch Method

A screen shot of a computer code

Description automatically generatedThis is for editing existing data in our database, but first we are going need to use the get method so we can fetch the instance we need edited and then fill our form with it.

Get method takes just one argument; the URL with the object’s id attribute.

Put method takes two arguments: URL with the object’s id attribute and then the object with updated properties.

In our component, we get the service method with dependency injection. Our implementation will be in two halves though. We first need to get the object and then update our form with it

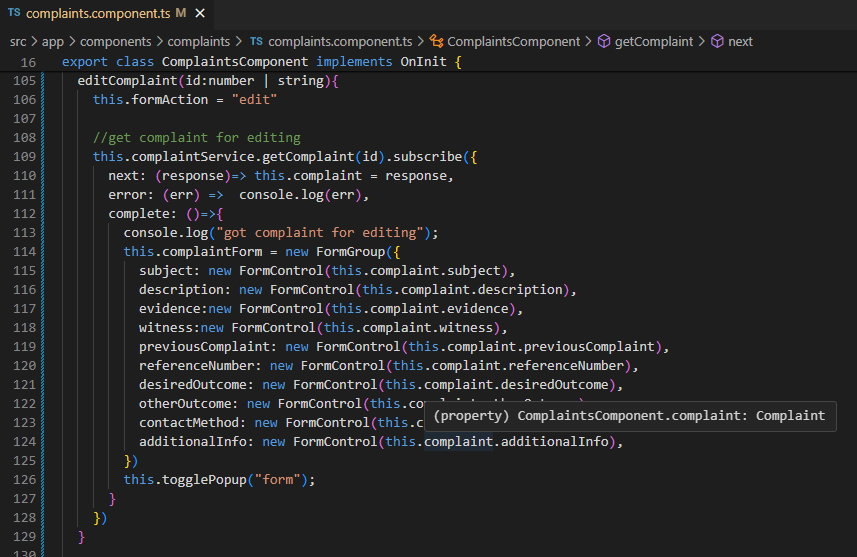
A screen shot of a computer code

Description automatically generatedWe decided to use the add/edit form’s popup feature for deletion confirmation as well. As a result, we used the togglePopup() function that triggers our form.

In the form we have it take an argument whether it should popup our form or otherwise.

In our component, we create a function to get the id of the clicked object, and then trigger the get complaint service and on complete we fill our form with it and subsequently render the form with it.

From the function we first see that a property is defined in our class detailing the action our form will be performing; in this case edit.

Then we trigger the get complaint service function whose next property upon subscription saves the response in a property of our class named ‘complaint’.

We then pass the properties to the respective form fields and subsequently render the form by triggering the popup function.

The second half of the editing is going to be our submit function. Based on the value of the formAction property, we will perform a series of actions to add a new entry or update an existing entry.

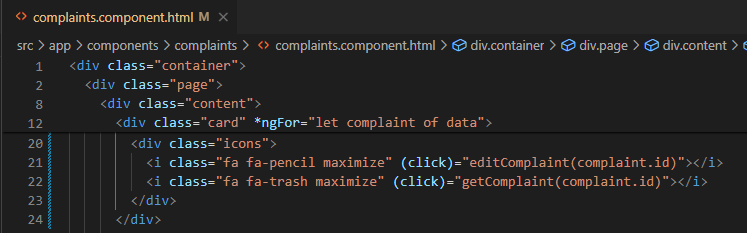
A screen shot of a computer program

Description automatically generatedWe first create a variable to store the complaint we will be passing to our put method. Then an if else condition to add or delete.

Within our else edit block, we update our new variable with the properties of our gotten object for editing and override the properties with the fields of our form.

Then pass the variable and its id to our put service method for update. If successful, we get toastr to render a success message, hide our form and reroute to the first page of our complaints – need to rework on the latter.

## Delete Method

In the template, we have our icons wrapped with the gateway functions to our http methods around our icons.

A screenshot of a computer screen

Description automatically generatedThe delete functionality triggers the get service method because we seek to actually delete the object with the confirmation popup.

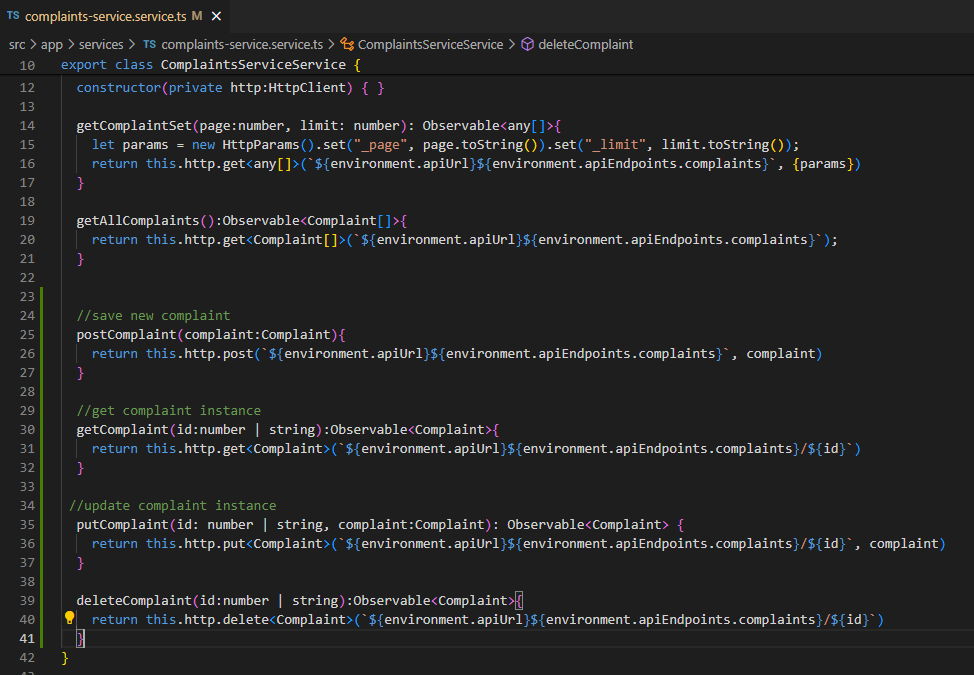
In our service, we trigger our http delete method by passing it an id attribute for appending to the url to get the object to be deleted.

As our template is bound with the getComplaint method, we have our component get the complaint with the said id and then toggle our confirmation popup which will later be added to our template.

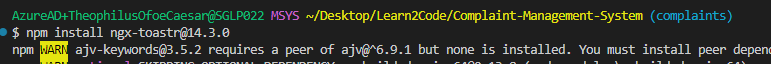
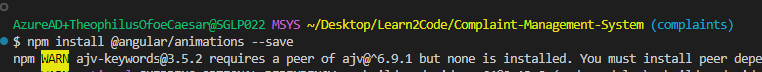
In our template we give the popup div a class named popup with a display property of none and then add class show based on the property of show confirmation popup to give our popup div a display property of block.

A screenshot of a computer

Description automatically generatedWe have buttons that will either close our pop up box or trigger our delete complaint method and service.



## Notification with toastr

Just to display a prompt when we interact with our API; we will be leveraging the ngx toastr library.

A screen shot of a computer

Description automatically generatedStep 1: Install ngx toastr and angular animations.

Step 2: Update app module with toastr and browser animations module.

Step 3: Copy the toastr styling from github repo and paste in a file.

A screenshot of a computer screen

Description automatically generated

A screen shot of a computer screen

Description automatically generatedStep 4: add the path to the toastr styling in our angular.json file, as is the case with styles.css.

From here on out, we should be able to use it within our component.

A screen shot of a computer screen

Description automatically generatedWe leverage dependency injection with the toastrService in our constructor, then create a function to display some text in the popup.

We call this popup function within our submit function as well as clear our form.

## A screen shot of a computer program Description automatically generatedComplaint Status Cards

Create a service to fetch our status from our ‘server’ and leverage it with dependency injection within our complaint’s component file.

In our component, we created a variable to store all our complaints, and all the statuses.

A computer screen with many colorful text

Description automatically generatedWe inject our status service into our constructor and then subscribe to it’s getAllStatuses() method within one function which is run on initialization.

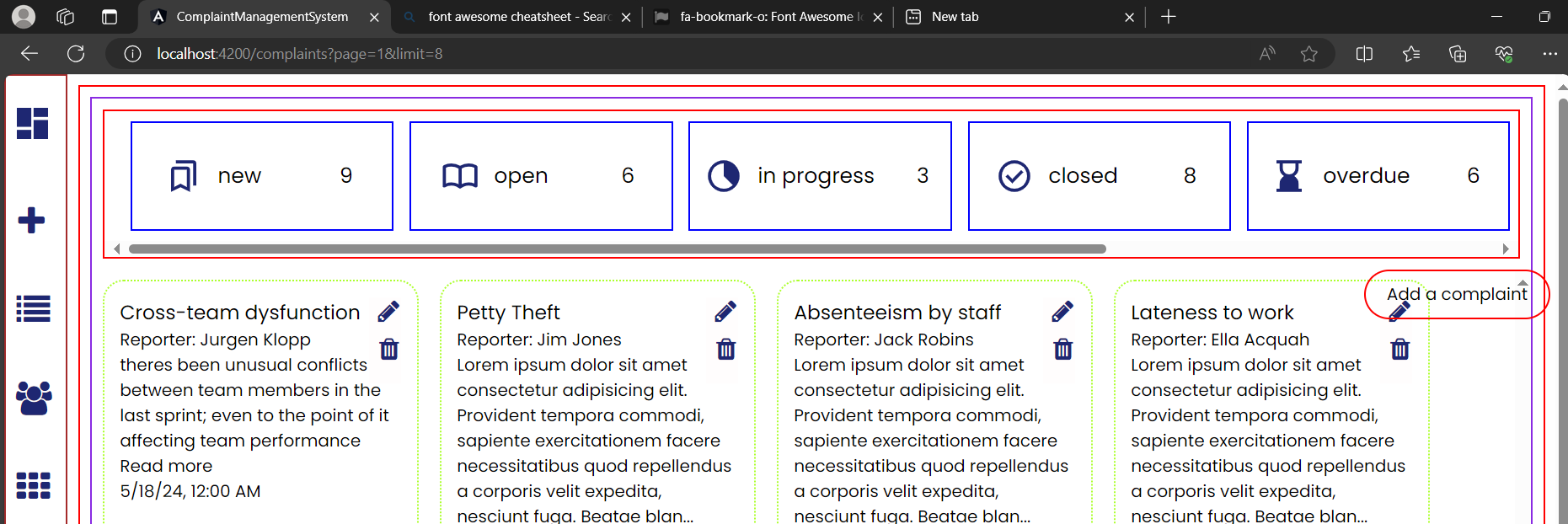
1. In our subscription method, we store the response in our array of statuses.
2. Our complete method, is where the focus is going to be. We get the list of all our complaints and then store it in a variable in line 71.
3. The next line has us check all of our status and then filter our complaints by their status and count them then add a property count, with their corresponding values for rendition on our homepage.

A screen shot of a computer

Description automatically generated

In our template we use the for structural directive to loop through our now update list of statuses and used interpolation to have their details rendered in the template.

Also downloaded the icons from google fonts.

next, we’ll be looking at filtering complaints based on the status on click.

# Filtering with Pagination

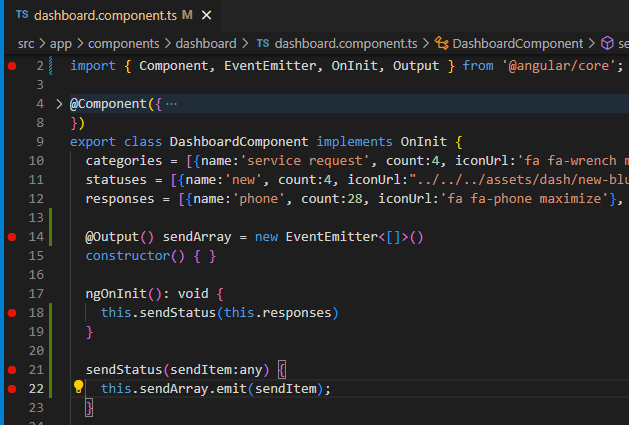
When implementing pagination and filtering together, you should ideally filter the entire list of complaints first and then apply pagination to the filtered results. This ensures that the user gets accurate pagination based on the filtered data, not just the currently displayed paginated subset.

Here's how you can achieve this:

#### Step-by-Step Guide

1. **Fetch the Entire Data**: Fetch the entire list of complaints from your JSON source.
2. **Filter the Data**: Apply your filter criteria to the entire list of complaints.
3. **Paginate the Filtered Data**: Paginate the filtered list of complaints.

## Output Input Decorators – using Shared Service or @ViewChild - must revisit

Now we made a query once to fetch our list of statuses, we can also share this list with other components using output and input decorators. Let’s give it a shot. Since our dashboard is the default page loaded first upon routing, lets try send from dashboard to home and take it within our complaints component.

### Output

Section A – child send to parent

* Line 2 - We import Output and EventEmitter from angular core.
* Line 14 - Then we create a property ‘sendArray’ and assign it an instance of EventEmitter of type array.
* Since we will be sending the data based on an event, we need to create a function that will trigger our emission by an event – often bound to click, but this time I prefer on init.
* Line 21 - creates a simple function that takes the argument passed as a param and passes it to our ouput emitter property to send
* Line 18 – Finally, we trigger our function in oninit. We could equally have a button with a click event trigger our function in the dashboard template.

Section B – parent receive from child

## Shared Service

Since we render child components with router outlets, we cant directly communicate with parent and children, hence Shared Service.

# A screenshot of a computer screen Description automatically generatedUser Management

Update user interface and list of users in JSON file.

Code in the interface is going to be the password.