Contents

[**part 01 – API Calls** 2](#_Toc80795989)

[**part 02 – Observables** 4](#_Toc80795990)

[**part 03 – Displaying Data** 6](#_Toc80795991)

[**part 04 – Post Request** 8](#_Toc80795992)

[**part 05 – Logging In** 9](#_Toc80795993)

[**part 06 – Persist Login** 12](#_Toc80795994)

[**part 07 – Moving to a Service** 15](#_Toc80795995)

[**part 08 – Adapt Login to use Service** 19](#_Toc80795996)

[**part 09 – (Optional) Using an Interface** 20](#_Toc80795997)

[**Appendix A – Subject** 22](#_Toc80795998)

[**Appendix B – Auth Service Explained** 23](#_Toc80795999)

Day02 Introduction to NG 11

# part 01 – API Calls

1. Create a new component like we did on Day01 Part01 section 5. So:  
   **ng g c customerlist**

We will use this component to just show all customers in our database, nothing special about this one.

1. Add this new component to the routing like you did in Part03 of Day01

|  |
| --- |
| **import { RegisterComponent } from "./register/register.component";**  **import { customerListComponent } from "./customerlist/customerlist.component";**  **const routes: Routes = [**  **{ path: '', redirectTo: '/home', pathMatch: 'full' },**  **{ path: 'home', component: HomeComponent },**  **{ path: 'register', component: RegisterComponent },**  **{ path: 'customers', component: customerListComponent }**  **];** |

1. We would need Angular’s *http* module, so in the parent app.module.ts file import this module

|  |
| --- |
| **import { AppComponent } from './app.component';**  **import { HomeComponent } from './home/home.component';**  **import { RegisterComponent } from './register/register.component';**  **import { HttpClientModule } from '@angular/common/http';** |

1. Add this module to the imports section:

|  |
| --- |
| **],**  **imports: [**  **BrowserModule,**  **AppRoutingModule,**  **ReactiveFormsModule,**  **HttpClientModule**  **],**  **providers: [],** |

(remember to insert a comma at the line above)

1. Back in the child component, so customerlist.component.ts file import the **HttpClient** like we did in the parent .ts file

|  |
| --- |
| **import { Component, OnInit } from '@angular/core';**  **import { HttpClient } from '@angular/common/http';**  **@Component({** |

1. Now in the constructor, pass in (inject) the HttpClient

|  |
| --- |
| **export class customerListComponent implements OnInit {**    **constructor(private http:HttpClient) {**    **}** |

1. Within that class, so the customerListComponent class, create a new property call it *customerList*, it will be of the *any* data type:

|  |
| --- |
| **export class customerListComponent implements OnInit {**  **customerList: any;**  **constructor(private http:HttpClient) { }**  **ngOnInit(): void {**  **}** |

1. Once the app starts, the method **ngOnInit()** will fire, in here we can use our *http* object to make the call to our mock database

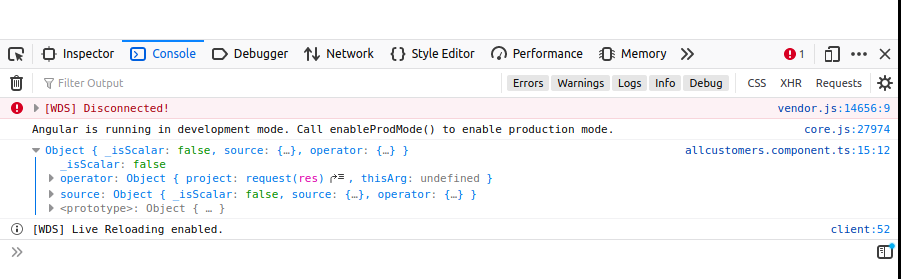
|  |
| --- |
| **customerList: any;**  **constructor(private http: HttpClient) { }**  **ngOnInit() {**  **this.http.get('http://localhost:3000/customers');**  **}** |

1. Assign the *observable* returned from the **get()** to **customerList.** Do this in the **ngOnInit()** mehtod.

|  |
| --- |
| **ngOnInit() {**  **this.customerList = this.http.get('http://localhost:8000/customers);**  **}** |

1. Pass this object to the console log and take a look at it:

|  |
| --- |
| **ngOnInit(): void {**  **this.customerList = this.http.get('http://localhost:3000/customers');**  **console.log(this.customerList);**  **}** |



The customers data is in this Observable object, but we will have to do some more work to get to it

# part 02 – Observables

1. Since the **get()** method of the http object returns an observable we need an observable object to store the return data

|  |
| --- |
| **export class customerListComponent implements OnInit {**  **customerList$: Observable<any>;**  **constructor(private http:HttpClient) { }** |

We can change our any *customerList* variable to a type of **Observable**, which will handle any data type. You will have to import the Observable class from the *RxJS* library also.

1. Now we can change the assignment in the **ngOnInit**() method:

|  |
| --- |
| **ngOnInit(): void {**  **this.customerList$ = this.http.get('http://localhost:3000/customers');**  **console.log(this.customerList$);**  **}** |

1. Now we can subscribe to the **customerList**$ observable, but we could skip all of that for now and just subscribe to the **get**() method itself. First return *customerList* to a regular variable of the type of **any**

|  |
| --- |
| **export class customerListComponent implements OnInit {**  **customerList: any;**  **constructor(private http:HttpClient) { }** |

1. Within the **ngOnInit**() method, just run the **get**() method and add a **subscribe**() method, also don’t log anything as yet:

|  |
| --- |
| **ngOnInit(): void {**  **this.http.get('http://localhost:3000/customers')**  **.subscribe();**  **console.log();**  **}** |

1. Now inside of the subscribe method, we have access to the customers data, so provide a variable to store it and also pass it to the **customerList** property:

|  |
| --- |
| **ngOnInit(): void {**  **this.http.get('http://localhost:3000/customers')**  **.subscribe(data => this.customerList = data);**  **console.log();**  **}** |

Always use the **this** keyword when using properties of the class. This function can also be written as shown below:

|  |
| --- |
| **ngOnInit(): void {**  **this.http.get('http://localhost:3000/customers')**  **.subscribe(function(data){**  **this.customerList = data;**  **});** |

1. If you wanted to see the data, you would need to move the **log()** method into the subscribe() method, remember to use curly braces:

|  |
| --- |
| **ngOnInit(): void {**  **this.http.get('http://localhost:3000/customers')**  **.subscribe(data => {**  **this.customerList = data;**  **console.log(this.customerList);**  **});**  **}** |

As you can see, the data is finally available, asynchronously

# part 03 – Displaying Data

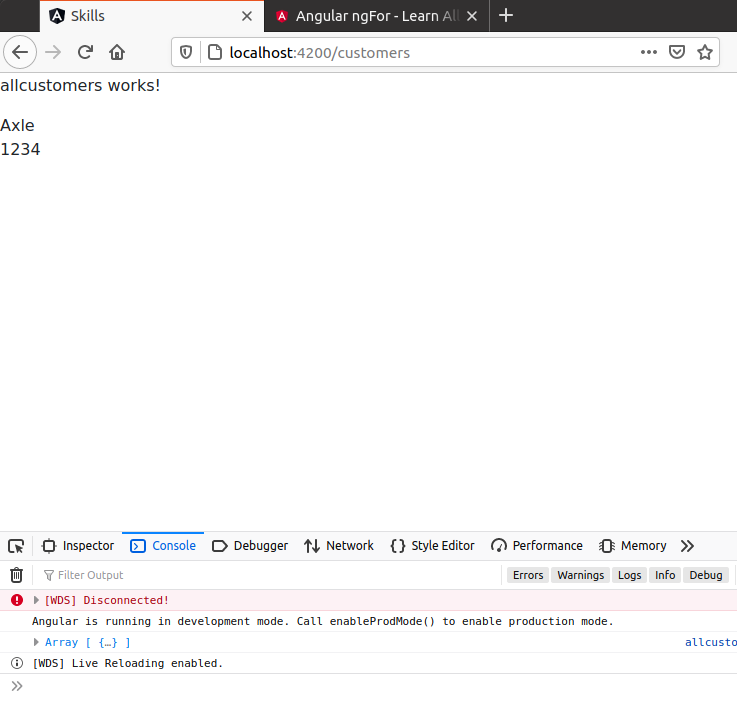
1. On the template, so on customerlist.component.html begin to structure a few html tags to handle the array of data coming shortly:

|  |
| --- |
| **<p>customerlist works!</p>**  **<div>**  **<div></div>**  **<div></div>**  **</div>** |

The first div is to accept the array, the inner pair of divs are used to display username and password

1. Finally we can iterate over the array and use interpolation characters to extract individual pieces of data:

|  |
| --- |
| **<p>customerList works!</p>**  **<div \*ngFor="let customer of customerList">**  **<div>{{customer.username}}</div>**  **<div>{{customer.password}}</div>**  **</div>** |



1. All the code so far for the class:

|  |
| --- |
| **import { Component, OnInit } from '@angular/core';**  **import { HttpClient } from "@angular/common/http";**  **import { Observable } from 'rxjs';**  **@Component({**  **selector: 'app-customerlist',**  **templateUrl: './customerlist.component.html',**  **styleUrls: ['./customerlist.component.css']**  **})**  **export class CustomerlistComponent implements OnInit {**  **customerList: any;**  **constructor(private http:HttpClient) { }**  **ngOnInit(): void {**  **this.http.get("http://localhost:3000/customers")**  **.subscribe(data => {**  **this.customerList = data**  **console.log(this.customerList);**  **});**  **}**  **}** |

1. (Optional alternative way of doing this). Return the import statement of **Observable** and re-create the **customerList** property of type **Observable**:

|  |
| --- |
| **import { Component, OnInit } from '@angular/core';**  **import { HttpClient } from '@angular/common/http';**  **import { Observable } from 'rxjs/internal/Observable';**  **@Component({**  **selector: 'app-customerList',**  **templateUrl: './customerlist.component.html',**  **styleUrls: ['./customerList.component.css']**  **})**  **export class customerListComponent implements OnInit {**  **allCustomers$: Observable<any>;**  **constructor(private http:HttpClient) { }** |

1. Within the **ngOnInit()** method, all you have to do is pass the observable returned from the **get()** method straight onto the customerList$ property:

|  |
| --- |
| **ngOnInit(): void {**  **this.allCustomers$ = this.http.get('http://localhost:3000/customers');**  **}** |

1. Then, over in the template, just pipe the data through an **async** method:

|  |
| --- |
| **<p>customerList works!</p>**  **<div \*ngFor="let customer of allcustomerList$ | async">**  **<div>{{customer.username}}</div>**  **<div>{{customer.password}}</div>**  **</div>** |

# part 04 – Post Request

On Day01 Parts 6 and 7 we developed the *register* form but we just logged the form’s contents to the console, now we will make a post request and send that data to the db.json file.

1. Before moving on, the json-server will expect our customers to have ids, so depending on how many records you have, assign incrementing ids to each:

|  |
| --- |
| **[**  **{**  **"id":"1",**  **"username": "Axle",**  **"password": "1234"**  **},**  **{**  **"id":"2",**  **"username": "John",**  **"password": "john"**  **}**  **]** |

1. On Day01 Parts 6 and 7 we developed the *register* form but we just logged the form’s contents to the console, now we will make a post request and send that data to the db.json file. Before we can do anything, we need the HttpClient

|  |
| --- |
| **import { FormGroup, FormControl, Validators } from "@angular/forms";**  **import { HttpClient } from "@angular/common/http";**  **@Component({** |

1. Remember to inject the service into the class via the constructor:

|  |
| --- |
| **export class RegisterComponent implements OnInit {**  **frmRegister: FormGroup;**  **constructor(private http:HttpClient) {**  **this.frmRegister = this.createFormGroup();**  **}** |

1. Change the **onSubmit()** function to this:

|  |
| --- |
| **onSubmit() {**  **//console.log(this.frmRegister.value);**  **this.http.post(**  **'http://localhost:3000/customers',**  **this.frmRegister.value**  **);**  **}** |

1. The above code is just the request, we also need to chain on a **subscribe()** method:

|  |
| --- |
| **onSubmit() {**  **//console.log(this.frmRegister.value);**  **this.http.post(**  **'http://localhost:3000/customers',**  **this.frmRegister.value**  **).subscribe(**  **data => {**  **console.log('POSTing data successful ', data);**  **}**  **);**  **}** |

p

1. Normally though, we also add as a second parameter to the **subscribe()** method, an error handler:

|  |
| --- |
| **onSubmit() {**  **//console.log(this.frmRegister.value);**  **this.http.post(**  **'http://localhost:3000/customers',**  **this.frmRegister.value**  **).subscribe(**  **data => {**  **console.log('POSTing data successful ', data);**  **},**  **error => {**  **console.log('Error', error);**  **}**  **);**  **}** |

An Observable can either be resolved or rejected, which is why we need an error handler.

# part 05 – Logging In

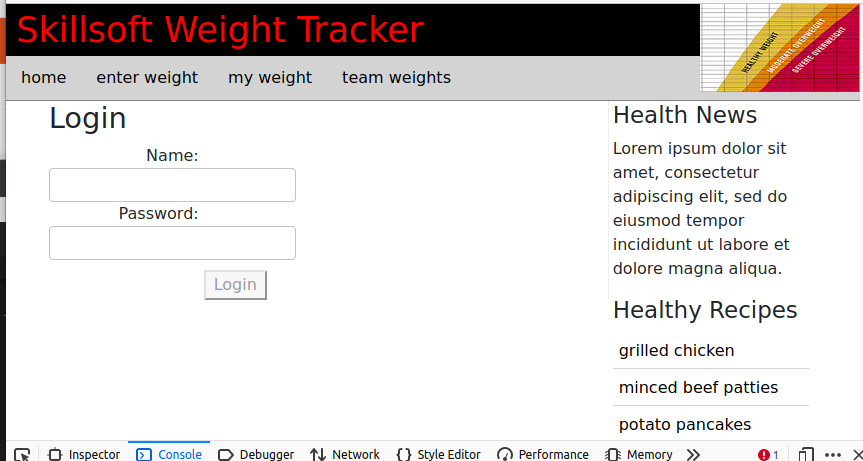
1. Repeat all the steps for create a new component, in fact this will be almost exactly the same form for registering but this time we will use it for logging in.   
   a. in a terminal window, execute this line: ng g c login
2. Import the HttpClient from @angular/common/http as well as all the Form modules from @angular/forms
3. copy all the code between the class definition and the end of the register.component.ts file
4. change the FormGroup property from frmRegister to frmLogin
5. remove everything from the onSubmit() function

This is what the login.component.ts file should look like now:

|  |
| --- |
| **import { Component, OnInit } from '@angular/core';**  **import { FormGroup, FormControl, Validators } from '@angular/forms';**  **import { HttpClient } from '@angular/common/http';**  **@Component({**  **selector: 'app-login',**  **templateUrl: './login.component.html',**  **styleUrls: ['./login.component.css']**  **})**  **export class LoginComponent implements OnInit {**  **frmLogin: FormGroup;**  **constructor(private http:HttpClient) {**  **this.frmLogin = this.createFormGroup();**  **}**  **createFormGroup() {**  **return new FormGroup({**  **username: new FormControl(**  **'',**  **[**  **Validators.required,**  **Validators.minLength(2)**  **]**  **),**  **password: new FormControl('', [Validators.required])**  **});**  **}**  **onSubmit() {**  **}**  **//**  **ngOnInit(): void {**  **}**  **}** |

1. Now in the template, copy all the code from register.component.html to login.component.html, just change the formGroup name to frmLogin. Change all occurrences of frmRegister to frmLogin.
2. Create a path for this component in the routing ts file:

|  |
| --- |
| **import { customerListComponent } from "./customerList/customerList.component";**  **import { LoginComponent } from "./login/login.component";**  **const routes: Routes = [**  **{ path: '', redirectTo: '/home', pathMatch: 'full' },**  **{ path: 'home', component: HomeComponent },**  **{ path: 'register', component: RegisterComponent },**  **{ path: 'customers', component: customerListComponent },**  **{ path: 'login', component: LoginComponent }**  **];** |

1. Change all prompts to reflect that this is a login form and not the register form, so the button and heading needs to be changed.

# part 06 – Persist Login

We are at the point where we need to check the login credentials against our database and also devise a long-term strategy to store a successful login

1. Create an Observable called user$ to hold the data being returned once we find our user:

|  |
| --- |
| **export class LoginComponent implements OnInit {**  **frmLogin: FormGroup;**  **user$: Observable<any>;**  **constructor(private http:HttpClient) {**  **this.frmLogin = this.createFormGroup();**  **}** |

Remember to import the Observable module from rxjs

1. Then complete the **onSubmit()** function to hit the database and return the user. The code is similar to what we did in the *customerList* component:

|  |
| --- |
| **onSubmit() {**  **this.user$ = this.http.get('http://localhost:3000/customers');**  **}** |

Remember to import the Observable module from rxjs

1. Next step, not necessary but convenient, create two local variables to hold the current user and password

|  |
| --- |
| **onSubmit(): void {**  **let currentUser = this.frmLogin.value.username;**  **let currentPassword = this.frmLogin.value.password;**  **this.user$ = this.http.get('http://localhost:3000/customers');**  **}** |

1. When we hit the /customers endpoint, if we just go with what we had in customers we will get all customers, we want a specific customer, so:

|  |
| --- |
| **onSubmit(): void {**  **let currentUser = this.frmLogin.value.username;**  **let currentPassword = this.frmLogin.value.password;**  **this.user$ = this.http.get('http://localhost:3000/customers',**  **{**  **params:{username:currentUser}**  **}**  **);**  **}** |

Notice the params object, this is so that we can achieve something like this:  
<http://localhost:3000/customers/?username=John>

1. Now we subscribe to the user$ in order to work with the return from our get() call

|  |
| --- |
| **this.user$ = this.http.get('http://localhost:3000/customers',**  **{**  **params:{username:currentUser}**  **}**  **);**  **//**  **this.user$.subscribe();**  **}** |

1. (Optional) check the data being returned using the console window:

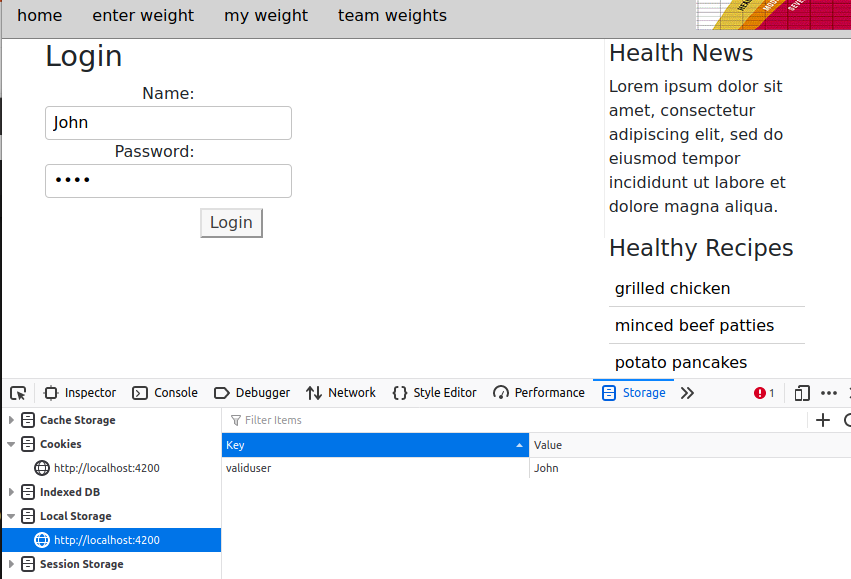
|  |
| --- |
| **this.user$.subscribe(data=>{console.log(data);**  **});** |

1. Now we can use that data to see if we have a match:

|  |
| --- |
| **this.user$.subscribe(data=>{**  **if(currentUser == data[0].username && currentUser == data[0].password){**  **//we have a match**  **console.log("User is valid");**  **}**  **});**  **}**  **ngOnInit(): void {** |

1. Lets store the current user in the browser’s local storage so we can retrieve this value in the future

|  |
| --- |
| **this.user$.subscribe(data=>{**  **if(data[0].username == currentUser && data[0].password == currentPassword){**  **console.log("Valid User");**  **localStorage.setItem('validuser', currentUser);**  **}**  **else**  **console.log("Invalid User");**  **})** |



1. If we have a successful login, we can re-direct the user to the home page, otherwise have them do the challenge again:

|  |
| --- |
| **this.user$.subscribe(data=>{**  **if(data[0].username == currentUser && data[0].password == currentPassword){**  **console.log("Valid User");**  **localStorage.setItem('validuser', currentUser);**  **this.router.navigateByUrl('/home');**  **}**  **else{**  **console.log("Invalid User");**  **this.router.navigateByUrl('/login');**  **}**  **})}** |

1. You will have to import the Router module from @angular/router and inject this class via the constructor:

|  |
| --- |
| **import { Observable } from 'rxjs';**  **import { Router } from "@angular/router";**  **@Component({**  **….**  **})**  **export class LoginComponent implements OnInit {**  **…**  **constructor(private http:HttpClient, private router:Router) {**  **this.frmLogin = this.createFormGroup();**  **}** |

# part 07 – Moving to a Service

Services in Angular are just classes that contain one or more functions related to a specific concern like data access or in our case authentication. Services allow us to share functionality among unrelated classes. Services are injectable, meaning we do not need to use the *new* keyword. Also services implement the singleton pattern, so one object serves multiple components.

1. Use the folder where the Angular application is running, then run the following command to install. For this, I would stop the application.   
    **ng generate service auth**

Restart the application using **ng serve**

Angular services are built to be used out of the box, just provide the functionality you need, which in this case is to pass the username and password to an API for authentication. Notice the @[Injectable](https://angular.io/api/core/Injectable) decorator that takes a metadata object. This tells Angular to inject this service where necessary and also to perform garbage collection.

1. We will be using another built-in service, the HttpClient service which we will *inject* via the constructor of this service:

|  |
| --- |
| **import { Injectable } from '@angular/core';**  **import { HttpClient } from '@angular/common/http';**  **@Injectable({**  **providedIn: 'root'**  **})**  **export class AuthService {**  **constructor(private http: HttpClient) {**  **}** |

1. We need two other packages so import the following:

|  |
| --- |
| **import { Injectable } from '@angular/core';**  **import { HttpClient } from '@angular/common/http';**  **import { Observable } from 'rxjs/internal/Observable'; import { Subject } from 'rxjs/internal/Subject';**  **@Injectable({** |

An Observable is like a Promise object, it accumulates data over time and then does something with the data.

1. The first function in our service class, the **login()** function, should be able to tell us if the user is valid or not and for that we need a valid name and a password. We already have this data via the login.component.ts file, so we can supply it here:

|  |
| --- |
| **login(userData):Observable<boolean> {**  } |

The userData will be the username and password, so this is passed when this service is used. Also this login function has to return an Observable since the operation being performed is asynchronous.

1. We will create a property at the class level, to hold the result of our checking the user.

|  |
| --- |
| **export class AuthService {**  **user$: Observable<any>;**  **constructor(private http: HttpClient) { }**  **login(userData:any): Observable<boolean> {** |

1. Also we can create two variables to hold the username and password, just like we did for the original login function in login.component.ts. one more variable isLoggedIn will be set to false initially and we do need a Subject variable:

|  |
| --- |
| **export class AuthService {**  **user$: Observable<any>;**  **constructor(private http: HttpClient) { }**  **login(userData:any): Observable<boolean> {**  **let currentUser = userData.username;**  **let currentPassword = userData.password;**  **let isLoggedIn = false;**  **let subject = new Subject<boolean>();** |

1. The rest of the code is almost the same as in the original function, the user$ is used to hold/store the result of the get() request

|  |
| --- |
| **login(userData:any): Observable<boolean> {**  **let currentUser = userData.username;**  **let currentPassword = userData.password;**  **let isLoggedIn = false;**  **let subject = new Subject<boolean>();**  **this.user$ = this.http.get(**  **'http://localhost:3000/customers',**  **{**  **params:{username:currentUser}**  **}**  **);** |

1. We then subscribe to the user$ observable and check if it found the current user and if it did, store that info in the local storage:

|  |
| --- |
| **this.user$ = this.http.get(**  **'http://localhost:3000/customers',**  **{**  **params:{username:currentUser}**  **}**  **);**  **this.user$.subscribe(data=>{**    **});** |

1. Check verify the current user and store that info in the local storage:

|  |
| --- |
| **this.user$ = this.http.get(**  **'http://localhost:3000/customers',**  **{**  **params:{username:currentUser}**  **}**  **);**  **this.user$.subscribe(data=>{**  **if(currentUser == data[0].username && currentPassword == data[0].password){**  **localStorage.setItem('validuser', currentUser);**  **}**  **});** |

1. It is better to make sure we have data, also make the isLoggedIn variable true

|  |
| --- |
| **this.user$ = this.http.get(**  **'http://localhost:3000/customers',**  **{**  **params:{username:currentUser}**  **}**  **);**  **this.user$.subscribe(data=>{**  **if(data[0]){**  **if(currentUser == data[0].username && currentPassword == data[0].password){**  **localStorage.setItem('validuser', currentUser);**  **isLoggedIn = true;**  **}**  **}**  **});** |

1. If there were no users or invalid login credentials, return false:

|  |
| --- |
| **this.user$ = this.http.get(**  **'http://localhost:3000/customers',**  **{**  **params:{username:currentUser}**  **}**  **);**  **this.user$.subscribe(data=>{**  **if(data[0]){**  **if(currentUser == data[0].username && currentPassword == data[0].password){**  **localStorage.setItem('validuser', currentUser);**  **isLoggedIn = true;**  **}**  **} else{**  **isLoggedIn = false;**  **}** |

The question now is how do we make all of this asynchronous, the answer is the Subject.

1. One solution is to pass the isLoggedIn variable to the next() method of a subject and then return that subject as an observable in the end

|  |
| --- |
| **this.user$.subscribe(data=>{**  **if(data[0]){**  **if(currentUser == data[0].username && currentPassword == data[0].password){**  **localStorage.setItem('validuser', currentUser);**  **isLoggedIn = true;**  **subject.next(isLoggedIn);**  **}**  **} else{**  **isLoggedIn = false;**  **subject.next(isLoggedIn);**  **}**  **});**  **return subject.asObservable();**  **}** |

We cannot just return true or false because we would return from this function even before the user was checked

# part 08 – Adapt Login to use Service

1. Back in the login.component.ts file we can remove anything to do with logging in (we will insert the service in the next step)

|  |
| --- |
| **import { Component, OnInit } from '@angular/core';**  **import { FormGroup, FormControl, Validators } from "@angular/forms";**  **import { Router } from "@angular/router";**  **@Component({**  **selector: 'app-login',**  **templateUrl: './login.component.html',**  **styleUrls: ['./login.component.css']**  **})**  **export class LoginComponent implements OnInit {**  **frmLogin: FormGroup;**  **constructor() {**  **this.frmLogin = this.createFormGroup();**  **}**  **createFormGroup() {**  **return new FormGroup({**  **username: new FormControl('', [Validators.required]),**  **password: new FormControl('', [Validators.required])**  **});**  **}**  **onSubmit(): void {**  **let currentUser = this.frmLogin.value.username;**  **let currentPassword = this.frmLogin.value.password;**  **}**  **ngOnInit(): void {**  **}**  **}** |

1. First import the service

|  |
| --- |
| **import { FormGroup, FormControl, Validators } from "@angular/forms";**  **import { Router } from "@angular/router"; import { Observable } from 'rxjs/internal/Observable';**  **import { AuthService } from "./../auth.service";** |

1. Let’s now *inject* our service via the constructor:

|  |
| --- |
| **export class LoginComponent implements OnInit {**  **frmLogin: FormGroup;**    **constructor(private router:Router, auth:AuthService) {**  **this.frmLogin = this.createFormGroup();**  **}**  **createFormGroup() {** |

Remember to import the service at the top of this file, VS Code will assist you

1. Create a property of the Observable type to handle the return from our service:

|  |
| --- |
| **export class LoginComponent implements OnInit {**  **frmLogin: FormGroup; loginStatus$:Observable<boolean>;**    **constructor(private router:Router, auth:AuthService) {**  **this.frmLogin = this.createFormGroup();**  **}**  **createFormGroup() {** |

Remember to import the service at the top of this file, VS Code will assist you

1. In the onSubmit() method, we can implement that service like we did in similar situations, remember to pass the user data:

|  |
| --- |
| **onSubmit(): void {**  **this.loginStatus$ = this.auth.login(this.frmLogin.value);**  **this.loginStatus$.subscribe(data=>console.log(data));**  **}** |

1. Here is the final code for this function:

|  |
| --- |
| **onSubmit(): void {**  **this.loginStatus$ = this.auth.login(this.frmLogin.value);**  **this.loginStatus$.subscribe(status =>{**  **if(!status)**  **this.router.navigateByUrl('/login');**  **else**  **this.router.navigateByUrl('/home');**  **});**  **}** |

# part 09 – (Optional) Using an Interface

1. In the customerlist.component.ts file we can implement an interface which makes the code a bit more robust. Add this interface that describes our user so far:

|  |
| --- |
| **import { HttpClient } from "@angular/common/http";**  **import { Observable } from 'rxjs/internal/Observable';**  **interface User {**  **username: string;**  **password: string;**  **}**  **@Component({}** |

1. Now instead of our Observable working with the *any* data type, it now has a specific type to work with

|  |
| --- |
| **export class customerListComponent implements OnInit {**  **customerList$: Observable<User>;**  **constructor(private http:HttpClient) {** |

1. Let’s now make an array to hold our users, even if it is just one user, this will work

|  |
| --- |
| **export class customerListComponent implements OnInit {**  **customerList$: Observable<User>;**  **users: User[] = [];**  **constructor(private http:HttpClient) {** |

1. In the onInit() method we return the User[] type, which is an array:

|  |
| --- |
| **ngOnInit(): void {**  **this.http.get<User[]>('http://localhost:3000/customers')**  **.subscribe(data => this.users=data);**  **}** |

1. In the template we no longer have to use async also we use users instead of the observable:

|  |
| --- |
| **<div \*ngFor="let customer of users">**  **<div>Name: {{customer.username}}</div>**  **<div>Password: {{customer.password}}</div>**  **</div>** |

1. You can also make a path for the customer list:

|  |
| --- |
| **<div \*ngFor="let customer of users">**  **<div>Name: {{customer.username}}</div>**  **<div>Password: {{customer.password}}</div>**  **</div>** |

1. Finally copy all the html from any of the other components and display a list of customers inside the view

|  |
| --- |
| **<div id="container">**  **<main>**  **<div \*ngFor="let customer of users">**  **<div>Name: {{customer.username}}</div>**  **<div>Password: {{customer.password}}</div>**  **</div>**  **</main>**  **<aside>**  **<section>** |

1. You could try to enhance the look of the display a little with some BS code:

|  |
| --- |
| **<main>**  **<div>**  **<h2>Our Customers</h2>**  **</div>**  **<div class="container" \*ngFor="let customer of users">**  **<div class="row">**  **<div class="col-md-4 col-sm-6 col-xs-12">Name: {{customer.username}}</div>**  **<div class="col-md-4 col-sm-6 col-xs-12">Password: {{customer.password}}</div>**  **</div>**  **</div>**  **</main>** |

# Appendix A – Subject

Subject is just a class that extends the Observable type, behind the scenes. It is both an Observable and an Observer and it allows values to be multi-casted to more than one Observers.

This means that we can subscribe to a Subject to view values from its stream or we can give it values to put into the stream by calling the next() method.

A Subject will keep an array of observers as new observers subscribe to it.

When the method next() is called, the Subject will loop through the observers and emit the same value to each one of them (multicasting). This process also takes place when an error occurs.

As soon as a Subject completes, all the observers will be unsubscribed automatically.

# Appendix B – Auth Service Explained

An explanation of why the login() method of auth.service.ts MUST return an observable.

1. Without using asynchronous code, the login.coponent.ts code will simply call the login() method on auth.service.ts:

|  |
| --- |
| **onSubmit() {**  **this.loginStatus = this.auth.login(this.frmLogin.value);**  **if(this.loginStatus)**  **this.router.navigateByUrl('/login');**  **else**  **this.router.navigateByUrl('/home');**  **};** |

In this case, this.loginStatus is just a property of the LoginComponent class. It makes a synchroneous call to this.auth.login(), passing of course the username and password.

1. On the service side, this is the code:

|  |
| --- |
| **export class AuthService {**  **isLoggedIn:boolean = false;**  **constructor(private http: HttpClient) { }**  **login(userData:any):boolean {**  **let currentUser = userData.username;**  **let currentPassword = userData.password;**  **this.http.get(**  **'http://localhost:3000/customers',**  **{**  **params:{username:currentUser}**  **}**  **).subscribe(data=>{**  **if(data[0]){**  **if(currentUser == data[0].username && currentPassword == data[0].password){**  **localStorage.setItem('validuser', currentUser);**  **this.isLoggedIn = true;**  **}**  **} else{**  **this.isLoggedIn = false;**  **}**  **});**  **return this.isLoggedIn;**  **}**  **}** |

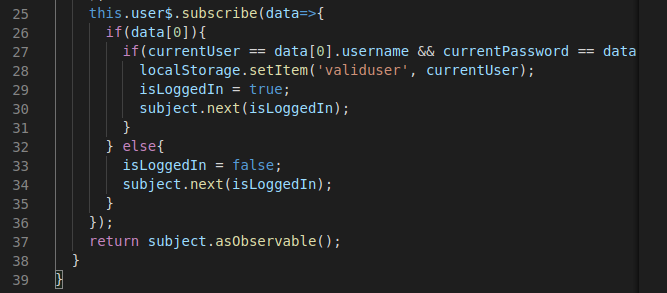
Notice that the only asynchronous call is the one to get() as we have no choice in this case. However everything else is synchronous.

1. The assumption here is that the property this.isLoggedIn will be true if the username and password are correct and false if they are not. Then we simply return the value of this.isLoggedIn, which starts off as false but may change if we have a verified user.

|  |
| --- |
| **).subscribe(data=>{**  **if(data[0]){**  **if(currentUser == data[0].username && currentPassword == data[0].password){**  **localStorage.setItem('validuser', currentUser);**  **this.isLoggedIn = true;**  **}**  **} else{**  **this.isLoggedIn = false;**  **}** |

1. We can test this by checking the value of loginStatus on the login.component.ts side:

|  |
| --- |
| **onSubmit() {**  **this.loginStatus = this.auth.login(this.frmLogin.value);**  **console.log(this.loginStatus);**  **// if(this.loginStatus)**  **// this.router.navigateByUrl('/login');**  **// else**  **// this.router.navigateByUrl('/home');**  **};** |

1. The problem here is that we get *false* on the first try, and may occasionally get a *true*, the values are inconsistent:
2. The explanation is that the function returns faster than the subscribe() method in the login() method. As long as this happens we will always get false since the property has not yet gotten the chance to change based on the credentials we sent.
3. The solution is to make the login() function asynchronous and therefore return when the isLoggedIn property is properly initialized. Since we have the ability to create observables, we can use the Subject.next() method to create one and have the isLoggedIn value as the data of that Subject.
4. Since the login() function must return an observable, it has to wait for the new observable to be created, then return. This forces the entire function to wait until the Subject is resolved