Contents

[part 01 – API Calls 2](#_Toc159307767)

[part 02 – Observables and Async 3](#_Toc159307768)

[part 03 – Post Request 5](#_Toc159307769)

[part 04 – Logging In 8](#_Toc159307770)

[part 05 – Persist Login 10](#_Toc159307771)

[part 06 – Signals an Introduction 13](#_Toc159307772)

[**Part 07 - Signals Objects and Mutate** 15](#_Toc159307773)

[Appendix A –Signal Sharing 17](#_Toc159307774)

[Appendix B – Signals in Forms 19](#_Toc159307775)

[Appendix C – API Calls using Signals 20](#_Toc159307776)

[appendix D – Moving to a Service 22](#_Toc159307777)

[Appendix E – Adapt Login to use Service 26](#_Toc159307778)

[Appendix F – Subject 28](#_Toc159307779)

[Appendix G – Auth Service Explained 29](#_Toc159307780)

[Appendix H – Subscribing to the Subject 32](#_Toc159307781)

Day02 Introduction to NG 17

# part 01 – API Calls

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

We will use this component to just show all employees 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 { EmployeesComponent } from "./employees/employees.component";**  **export const routes: Routes = [**  **{ path: '', redirectTo: '/home', pathMatch: 'full' },**  **{ path: 'home', component: HomeComponent },**  **{ path: 'register', component: RegisterComponent },**  **{ path: 'employees', component: EmployeesComponent }**  **];** |

1. We would need to export Angular’s **provideHttpClient()** method, so do this in app.config.ts:

|  |
| --- |
| **import { routes } from './app.routes';**  **import { provideHttpClient } from '@angular/common/http';**  **export const appConfig: ApplicationConfig = {**  **providers: [provideRouter(routes), provideHttpClient()]**  **};** |

You should be prompted to do the import. Note, this can also be done in main.ts.

1. Now to the consumer file, **employes**. In employees.component.ts file import the helper modules

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

1. Like in V.16, we can inject the **HttpClient** via the constructor since we do still have a class:

|  |
| --- |
| **styleUrl: './employees.component.css'**  **})**  **export class EmployeesComponent {**  **constructor(private http: HttpClient) {**  **// This service can now make HTTP requests via `this.http`.**  **}** |

1. Since the **get()** method of the **HttpClient** returns an **Observable**, let us create a variable of that type and have it be of the any type:

|  |
| --- |
| **export class EmployeesComponent {**  **employees$!:Observable<any>;**  **constructor(private http: HttpClient) { };**  **ngOnInit() {** |

1. If we now assign the return from the **get()** method to this new property, we can print it to see what it outputs:

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

A screen shot of a computer screen

Description automatically generated

# part 02 – Observables and Async

1. We can now use the same observable, **employees$** for our template output. NG17 now has the **@for** control flow available. This is used just like **@if.**

|  |
| --- |
| **<p>employees works!</p> @for (employee of employees$ | async; track $index) {**  **}** |

1. Add the **CommonModule** as we would need **async** support:

|  |
| --- |
| **import { HttpClient } from '@angular/common/http';**  **import { Observable } from 'rxjs';**  **import { CommonModule } from '@angular/common';** |

1. Then add it to the imports array, the **CommonModule** that is:

|  |
| --- |
| **@Component({**  **selector: 'app-employees',**  **standalone: true,**  **imports: [CommonModule],**  **templateUrl: './employees.component.html',**  **styles: ``**  **})** |

Note: you could just import and use NgFor instead of the entire CommonModule

1. All you do now is decide how you want to display **employee** and it’s parts:

|  |
| --- |
| **@for (employee of employees$ | async; track $index) {**  **<div>**  **Employee {{employee.username}} using {{employee.password}}**  **</div>**  **}** |

A screenshot of a computer

Description automatically generated

1. Within that class, so the **EmployeesComponent** class, create a new interface called **Employee**. Since the three fields have a certain signature, we can have a matching data type:

|  |
| --- |
| **import { HttpClient } from '@angular/common/http';**  **interface Employee {**  **id : string;**  **username : string;**  **password: string;**  **}**  **@Component({** |

Note: the three fields I refer here to are the ones in the db.json file. This piece of code must go above the @component part of the file.

1. Now we can subscribe to our data without **any** but by using an actual type:

|  |
| --- |
| **export class EmployeesComponent {**  **employees$!:Observable<Employee[]>;**  **constructor(private http: HttpClient) { };**  **ngOnInit() {**  **this.employees$ = this.http.get<Employee[]>('http://localhost:3000/employees');**  **}** |

# part 03 – 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 employees 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"**  **}**  **]** |

If you use the file provided or used the AI tool, it already has id’s so nothing to do here

1. With no module support in NG17, we import the **prodideHttpClient** directly into the app.config.ts file and add it to the **providers** array:

|  |
| --- |
| **import { routes } from './app.routes';**  **import { provideHttpClient } from '@angular/common/http';**  **export const appConfig: ApplicationConfig = {**  **providers: [provideRouter(routes), provideHttpClient()]**  **};** |

Notice that **providedHttpClient** is a function. You may have already done this task in Part 01 #3.

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** in the register component:

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

We do this in register.component.ts file

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

|  |
| --- |
| **password : new FormControl(null, Validators.required)**  **});**  **constructor(private http:HttpClient) {};**  **onSubmit() {** |

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

|  |
| --- |
| **constructor(private http:HttpClient) {};**  **onSubmit() {**  **this.http.post(**  **'http://localhost:3000/employees',**  **this.registerForm.value**  **)**  **}** |

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

|  |
| --- |
| **constructor(private http:HttpClient) {};**  **onSubmit() {**  **this.http.post(**  **'http://localhost:3000/employees',**  **this.registerForm.value**  **).subscribe()**  **}** |

1. At this point if we supply a bucket (variable) to catch and represent the aftermath of the POST transaction we should get an object back:

|  |
| --- |
| **constructor(private http:HttpClient) {};**  **onSubmit() {**  **this.http.post(**  **'http://localhost:3000/employees',**  **this.registerForm.value**  **).subscribe(response => console.log(response));**  **}** |

A screenshot of a computer

Description automatically generated

Notice the new id showing up.

Also if you refresh localhost port 3000 you should see the new employee Tom showing up as the fourth employee.

1. Although this works, we need to handle any errors. So since Angular uses the RxJS library and there are only three ways to handle an observable response, we will use two of those ways here, **next**() and **error**(). First remove everything from between the **subscribe()** method and replace it with a pair of curly braces:

|  |
| --- |
| **this.http.post(**  **'http://localhost:3000/employees',**  **this.registerForm.value**  **).subscribe({});** |

1. Now we can work with either of the three observable methods or any combination, so first the **next**() **method:**

|  |
| --- |
| **onSubmit() {**  **this.http.post(**  **'http://localhost:3000/employees',**  **this.registerForm.value**  **).subscribe({**  **next:data => console.log(data),**  **});**  **}** |

If you run the app at this point, a similar result to what we got in point #6 will result.

1. We also have to handle any errors so we add that Observable method in just like we did for **next():**

|  |
| --- |
| **this.http.post<any>('http://localhost:3000/employees',**  **this.frmRegister.value).subscribe({**  **next:data => console.log(data),  error: err => console.log(err)**  **});** |

At this point, you may try inserting a new employee. Notice the comma between the two methods. The third method attached to an Observable’s subscription is the complete() method. I discuss this more in-depth in the Asynchronous JavaScript bootcamp, but see below.

The entire **onSubmit**() function

|  |
| --- |
| **onSubmit() {**  **this.http.post(**  **'http://localhost:3000/employees',**  **this.registerForm.value**  **).subscribe({**  **next:data => console.log(data),**  **error: err => console.log(err)**  **});**  **}** |

The third method that can be used is the complete method. It is not necessary here but below I show how it is implemented:

|  |
| --- |
| **onSubmit() {**  **this.http.post(**  **'http://localhost:3000/employees',**  **this.registerForm.value**  **).subscribe({**  **next:data => console.log(data),**  **complete:()=> console.log("Data returned"),**  **error: err => console.log(err)**  **});**  **}** |

Usually the **complete**() method goes **before** the error method.

# part 04 – Logging In

1. Repeat all the steps for creating 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 –-skip-tests -s**

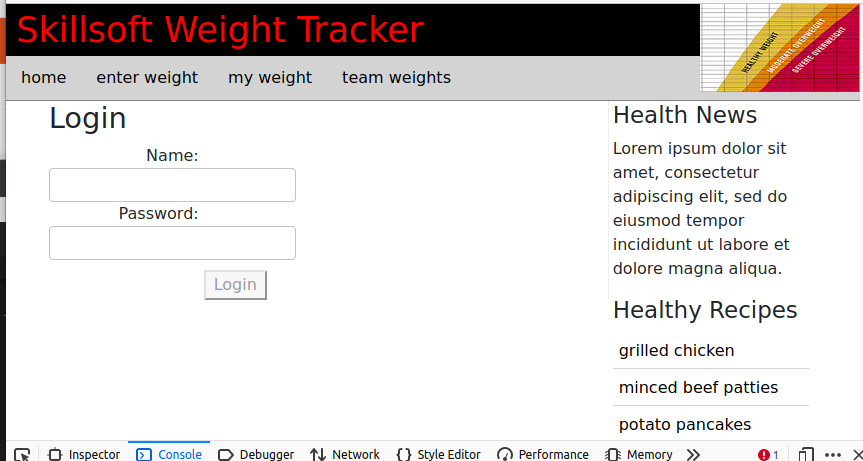
1. Import the **HttpClient** from **@angular/common/http** as well as all the Form modules from **@angular/forms** (in login.component.ts)
2. In the imports array, within the @Component decorator function, add the **ReactiveFormsModule**
3. Copy all of the code between the class of **Register** and past it between the class definition of Login
4. change the FormGroup property from **registerForm** to **loginForm**
5. remove everything from the **onSubmit**() function
6. remove the **Observable** object if you have one

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

|  |
| --- |
| **import { Component } from '@angular/core';**  **import { FormGroup,FormControl, ReactiveFormsModule, Validators } from '@angular/forms';**  **import { HttpClient } from "@angular/common/http";**  **@Component({**  **selector: 'app-login',**  **standalone: true,**  **imports: [ReactiveFormsModule],**  **templateUrl: './login.component.html',**  **styleUrl: './login.component.css'**  **})**  **export class LoginComponent {**  **loginForm = new FormGroup({**  **username : new FormControl(null, Validators.required),**  **password : new FormControl(null, Validators.required)**  **});**  **constructor(private http:HttpClient) {};**  **onSubmit() { }**  **}** |

1. Now in the template, copy all the code from register.component.html to login.component.html, just change the **formGroup** name to **loginForm**. Change all occurrences of **registerForm** to **loginForm**. Also change the <**h2>** tag to something appropriate for logging in.
2. Create a path for this component in the app.routes.ts file:

|  |
| --- |
| **import { EmployeesComponent } from "./employees/employees.component";**  **import { LoginComponent } from "./../app/login/login.component";**  **//**  **export const routes: Routes = [**  **{ path: '', redirectTo: '/home', pathMatch: 'full' },**  **…**  **{ 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 05 – 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$** in the login.component.ts file to hold the data being returned once we find our user:

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

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 **employees** component:

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

Remember to import the **Observable** module from **rxjs**

1. Next step, create two local variables to hold the current user and password

|  |
| --- |
| **export class LoginComponent {**  **user$! : Observable<any>;**  **loginForm = new FormGroup({**  **username : new FormControl(null, Validators.required),**  **password : new FormControl(null, Validators.required)**  **});**  **currentUser = this.loginForm.value.username;**  **currentPassword = this.loginForm.value.password;**  **constructor(private http:HttpClient) {};** |

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

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

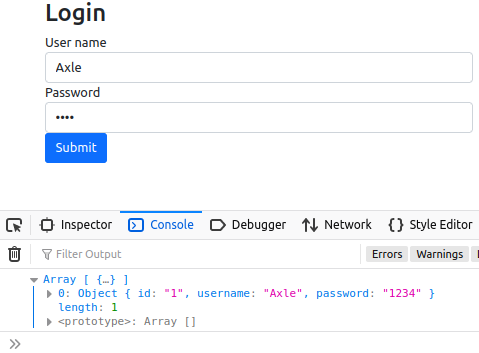
Notice the params object, this is so that we can achieve something like this:  
[http://localhost:3000/employees/?username=John](http://localhost:3000/customers/?username=John). Also since TS thinks that the object may be null, we add a ! to the end.

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/employees',**  **{**  **params:{username:currentUser!}**  **}**  **);**  **//**  **this.user$.subscribe();**  **}** |

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

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



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

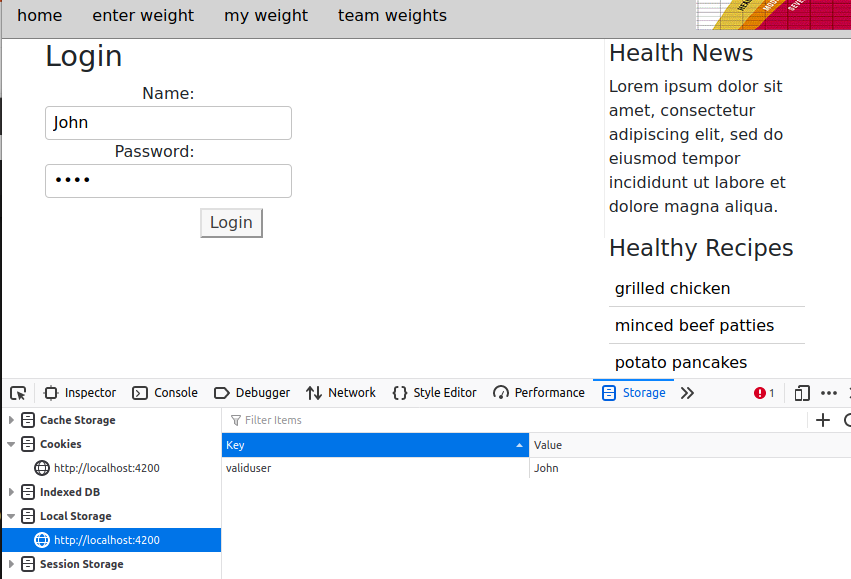
|  |
| --- |
| this.user$.**subscribe**(data=>{  **if(currentUser == data[0].username && currentPassword == data[0].password){**  **//we have a match**  **console.log("User is valid");**  **}**  **});**  **}**  **ngOnInit(): void {** |

1. We can also prepare for an invalid user code:

|  |
| --- |
| **this.user$.subscribe(data=>{**  **if(currentUser == data[0].username && currentPassword == data[0].password){**  **console.log("User is valid")**  **} else {**  **console.log("Invalid User!");**  **}**  **});** |

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 06 – Signals an Introduction

Signals behave on the publish/subscribe pattern of web development. Once you define a variable as a signal, other components will know about this signal and are updated when the value of that signal changes.

Once a signal is defined, use methods on that variable, even in the template. Signals work better if defined in a service the use them anywhere. Angular signals is supposed to improve data flow, change detection and component. It can make the app more reactive. A signal behaves just like a behavior subject.

1. In home.component.ts file, import the signal and computed classes. We will also use OnInit, so import that as well:

|  |
| --- |
| import { Component, OnInit, computed, signal } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css'] |

1. Declare the first Signal. In this case fName is the signal and it has a string value:

|  |
| --- |
| styleUrls: ['./app.component.css']  })  export class AppComponent {  fName = signal('Axle');  constructor(){ |

1. In order to use that signal, we use like a function. For example, in the template we could do this:

|  |
| --- |
| <div>  Hello {{ fName() }}  </div> |

1. So a Signal is just a variable, or piece of data, but it behaves like a function. As with all data, the value can be changed. Changing a Signal involves treating the Signal as an object which has an update() method associated with that object. But then we must pass a function into the update’s constructor. Not only that, we have to acknowledge the initial state as well:

|  |
| --- |
| fName = signal('Axle');  constructor(){  this.fName.update( prevValue => "AxleB" );  }; |

The parameter prevValue can be any valid variable name. You will see why it is important to have the previous state available shortly. However, you could probably see that Signals involve mutating the original state rather than just replacing values.

1. Add a new Signal and implement the NgOnInit() method:

|  |
| --- |
| export class AppComponent implements OnInit {  fName = signal('Axle');  lName = signal('');  constructor(){  this.fName.update(prevValue => "AxleB")  };  ngOnInit(): void {    }  } |

1. Notice that in #5 above, I did not specify a value for lName, we will do it in the ngOnInit() method using yet another Signal method, set() :

|  |
| --- |
| ngOnInit(): void {  this.lName.set('Barr');  } |

You could now comment out the one line of code in the constructor:

1. We can now use both Signals in the template like this:

|  |
| --- |
| <div>  Hello {{ fName() + " " + lName() }}  </div> |

1. Here we see how to use Signals in a custom function. I will create a new function, and a new signal that is *computed* from the other two existing Signals:

|  |
| --- |
| ngOnInit(): void {  this.lName.set('Barr');  }  getFullName(){  } |

1. This is one way to do the computation:

|  |
| --- |
| getFullName(){  let fullName = computed(() => this.fName() + ' ' + this.lName());  return (fullName());  } |

At the time of writing, it is possible to declare an empty (or null) Signal then use it afterwards. This technique only allowed me to *mutate* the original Signal, I could not use *computed*.

1. Another way to return a computed Signal:

|  |
| --- |
| getFullName(){  return (computed(() => this.fName() + ' ' + this.lName())());  } |

If you did this, you would have to execute the computed function. Notice the extra pair of parenthesis at the end of the computed statement.

1. Then in the template:

|  |
| --- |
| <div>  Hello {{ getFullName() }}  </div> |

Again, notice the extra pair of parenthesis at the end of the computed statement. This is in addition to #10.

# Part 07 - Signals Objects and Update (formally Mutate)

When I first developed this document, the **mutate()** method was suggested as the best way to update an object, including arrays. However that method has since been deprecated and the option now is to use **update**().

1. Let us work with an interface now. We will create an Employee interface in the home.component.ts:

|  |
| --- |
| import { Component, OnInit, computed, signal } from '@angular/core';  interface Employee {  fName : string;  lName: string;  }  @Component({ |

Remember an Interface must be declared above the @Component decorator

1. Create a Signal property in the AppComponent class based on this new interface:

|  |
| --- |
| export class AppComponent implements OnInit{  Axle = signal<Employee>({  });  fName = signal('Axle'); |

1. With the property in place, just move the two original signals into that property. However they are no longer Signals on their own, they are now part of a parent Signal property called Axle:

|  |
| --- |
| export class AppComponent implements OnInit{  Axle = signal<Employee>({  fName : 'Axle',  lName : ' '  });  constructor(){ |

Each member of this new Axle signal are just primitive strings now, not Signals themselves.

1. We will use the ngOnInit() method (or any other method) to update our Axle object:

|  |
| --- |
| ngOnInit(): void {  this.Axle.update(  );  }; |

1. The update() method, takes a function. The parameter of that function represents the original object. In this case it is the one we defined in #3 above. As it turns out, we can update part or all of that object. Here I will change just the lName part of the original object:

|  |
| --- |
| ngOnInit(): void {  this.Axle.update( o => {  o.lName='Barr';  return o }  );  }; |

Here lower case ‘o’ represents the original object, which must be returned from this update() method.

1. In order to render this on the template, you might think that this will work:

|  |
| --- |
| g<div>  Hello {{ ~~Axle()~~ }}  </div> |

This will NOT work, see below

1. Any method can be used to bind to the template interpolation code but here I use the getFullName() method. You might think at first that this will work:

|  |
| --- |
| getFullName(){  return this.Axle.fName + " " + this.Axle.lName;  }; |

1. As the IDE is indicating, fName and lName are not part of the signal of Employee type. However if we first execute each part of the object, the values will be revealed:

|  |
| --- |
| getFullName(){  return this.Axle().fName + " " + this.Axle().lName;  }; |

1. In order to render this on the template side, just execute the getFullName() function:

|  |
| --- |
| <div>  Hello {{ getFullName() }}  </div> |

1. Alternatively you could just return the signal from the getFullName() function:

|  |
| --- |
| getFullName(){  return this.Axle();  }; |

Remember to execute the signal!

1. Then on the template:

|  |
| --- |
| <div>  Hello {{ getFullName().fName }}  </div> |

Remember to execute the signal!

# Appendix A –Signal Sharing

Although it is possible to export a signal, it is safer to share signal via a service.

1. Create the service:

|  |
| --- |
| **ng g s employee** |

1. Create a new .ts file that will hold our employee interface:

|  |
| --- |
| **export interface Employee {**  **id: Number;**  **username: string;**  **password: string;**  **};** |

This is the same code from #8 so you can now remove the interface from the employee.component.ts file. Instead import the Employee interface from the newly created employee.ts file.

1. Here are all the imports from the service file:

|  |
| --- |
| **import { Injectable, inject, signal } from '@angular/core';**  **import { Employee } from "./employee";**  **import { HttpClient } from '@angular/common/http';** |

1. Inject the HttpClient and create the signal:

|  |
| --- |
| **export class EmployeeService {**  **http = inject(HttpClient);**  **constructor() { };**  **employees = signal<Employee[]>([]);** |

You did this in #5 and #7 from part 1

1. Start with either setting, getting or updating. Eventually you will also need an edit method. Here is the set method where we execute the API and load our signal with the array that is returned:

|  |
| --- |
| **employees = signal<Employee[]>([]);**  **setEmployees(): void {**  **this.http.get<Employee[]>('http://localhost:3000/employees')**  **.subscribe(empData => {**  **this.employees.set(empData);**  **})**  **};**  **getEmployees(){** |

1. For the getter, simply return the signal with the parameters:

|  |
| --- |
| **this.employees.set(empData);**  **})**  **};**  **getEmployees(){**  **return this.employees();**  **};**  **updateEmployees(newEmployee:Employee){** |

1. And for the update method, simply pass the form object to the API endpoint:

|  |
| --- |
| **return this.employees();**  **};**  **updateEmployees(newEmployee:Employee){**  **//this.employees.update(data => [...data, newEmployee]);**  **this.http.post(**  **'http://localhost:3000/employees',**  **newEmployee**  **).subscribe()**  **}** |

Remember that the code at this point, works with the database via the API, therefore there is no need to mess with the signal. In other cases, you change the signal and then update the API if any.

1. In both the Employee and Register components, import the service and call the appropriate methods. We start with the Employee component. First remove anything that had to with calling the API. Import the Employee interface, the EmployeeService and effect:

|  |
| --- |
| **import { Component, OnInit, effect } from '@angular/core';**  **import { EmployeeService } from "./../employee.service";**  **import { Employee } from "./../employee";**  **@Component({**  **selector: 'app-employees',** |

1. In the class, create a property of the Employee type and inject the EmployeeService:

|  |
| --- |
| **export class EmployeesComponent implements OnInit{**  **allEmployees : Employee[] = [];**  **constructor(private employeeService:EmployeeService){**  **}**  **ngOnInit() {}**  **}** |

1. Inside of an effect callback function, load the property you created in #9 with the array being returned from the service:

|  |
| --- |
| allEmployees : Employee[] = [];  constructor(private employeeService:EmployeeService){  effect(()=>{  this.allEmployees=this.employeeService.getEmployees();  });  }  ngOnInit() {}  } |

Note, this is just one of several ways to get data stored in a signal.

1. For the Register component, import all the same file as with the employee component and perform the same injection of the service. The only change here is to remove all the code from the onSubmit() function and replace it with this line:

|  |
| --- |
| constructor(private employeeService:EmployeeService){};  onSubmit() {  this.employeeService.updateEmployees(this.registerForm.value as Employee);  } |

Remember we are NOT manipulating the signal directly, we are still working with the API

# Appendix B – Signals in Forms

The use of Signals in forms is not advisable at this time and the techniques below will change in the near future. The following code and explanations are provided just for your information.

1. Import the **toSignal** class from **rxjs-interop** library, into the current login component:

|  |
| --- |
| **import { Observable } from 'rxjs';**  **import { Router } from "@angular/router";**  **import { toSignal } from "@angular/core/rxjs-interop";**  **@Component({**  **selector: 'app-login',** |

1. Now create two signals using the **toSignal**() method:

|  |
| --- |
| **password : new FormControl(null, Validators.required)**  **});**  **userName = toSignal();**  **passWord = toSignal();**  **constructor(private http:HttpClient, private router:Router) { };** |

Since signals must have an initial value, you could add a pair of single quotation marks inside of the parenthesis to indicate an empty string.

1. Now **userName** and **passWord** will turn into signals by the **toSignal**() method. However we need to pass **the value that changes:**

|  |
| --- |
| **password : new FormControl(null, Validators.required)**  **});**  **userName = toSignal(this.loginForm.controls.username.valueChanges);**  **passWord = toSignal(this.loginForm.controls.password.valueChanges);**  **//**  **constructor(private http:HttpClient, private router:Router) { };** |

Make sure that these two lines appear after where the loginForm was created.

1. In the **onSubmit**() method, we simply retrieve the value in the signal by calling the appropriate method:

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

Notice that the signal value is called with parenthesis, so just like a getter function.

1. Then in the **subscribe**() method use the signal values in the **if()** statement:

|  |
| --- |
| **this.user$.subscribe(data=>{**  **if(this.userName() == data[0].username && this.passWord() == data[0].password){**  **console.log("match");**  **localStorage.setItem('validuser', this.userName()!);**  **this.router.navigateByUrl('/home');**  **} else {** |

Note, you don’t have to use the signal for the localStorage, the currentUser value is good enough. Once you introduct signals, there is no need for currentUser or currentPassword.

# Appendix C – API Calls using Signals

This is the exact code from Part01 and Part02 except I used signals instead of Observables.

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

We will use this component to show all employees in our database, nothing special here.

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

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

1. We would need to export Angular’s **provideHttpClient()** method, so do this in app.config.ts:

|  |
| --- |
| **import { routes } from './app.routes';**  **import { provideHttpClient } from '@angular/common/http';**  **export const appConfig: ApplicationConfig = {**  **providers: [provideRouter(routes), provideHttpClient()]**  **};** |

You should be prompted to do the import. Note, this can also be done in main.ts.

1. Now to the consumer file, **employes**. In employees.component.ts file import the helper modules

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

1. Like in V.16, we can inject the **HttpClient** via the constructor since we do still have a class or we can use the inject() function to do it. Here I use the latter but you must remember to import it at the top of the file:

|  |
| --- |
| **export class EmployeesComponent implements OnInit{**  **http = inject(HttpClient);**  **constructor() { };**  **ngOnInit(): void {** |

1. Since we already know the shape of our data we can create an interface to match that shape. This allows us to avoid using <any>:

|  |
| --- |
| **import { HttpClient } from '@angular/common/http';**  **interface Employee {**  **id: Number;**  **username: string;**  **password: string;**  **};**  **@Component({** |

1. Use the interface to create a signal of that type:

|  |
| --- |
| **http = inject(HttpClient);**  **constructor() { };**  **employees = signal<Employee[]>([]);**  **ngOnInit(): void {** |

1. Since we already know the shape of our data we can create an interface to match that shape. This allows us to avoid using <any>:

|  |
| --- |
| **import { HttpClient } from '@angular/common/http';**  **interface Employee {**  **id: Number;**  **username: string;**  **password: string;**  **};**  **@Component({** |

1. Use the interface to create a signal of that type:

|  |
| --- |
| **http = inject(HttpClient);**  **constructor() { };**  **employees = signal<Employee[]>([]);**  **ngOnInit(): void {** |

1. Update the signal once get() is executed, and everything can be done inside ngOnInit():

|  |
| --- |
| **http = inject(HttpClient);**  **constructor() { };**  **employees = signal<Employee[]>([]);**  **ngOnInit(): void {**  **this.http.get<Employee[]>('http://localhost:3000/employees')**  **.subscribe(empData => {**  **this.employees.set(empData);**  **})**  **}** |

1. On the template side, start with a basic structure:

|  |
| --- |
| <div>  <ul>  </ul>  </div> |

1. Use the newer @for syntax with an else that might be empty:

|  |
| --- |
| <div>  <ul>  @for () {  <li>{{ }}</li>  } @empty {  <span>No employees found!</span>  }  </ul>  </div> |

1. Just like the older @for directive, select each employee from the array of employees and display:

|  |
| --- |
| <div>  <ul>  @for (employee of employees(); track employee.id) {  <li>{{ employee.username }}</li>  } @empty {  <span>No employees found!</span>  }  </ul>  </div> |

Remember to expose the signal value using ()

# appendix D – 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**

A screen shot of a computer

Description automatically generatedAngular 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. A subject is a special **Observable**, we have full control over how it emits data using it’s **next**() method.

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:any):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. Note **userData** is also of the any type.

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> { |

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>(); |

The rest of the code is almost the same as in the original login 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/employees',  {  params:{username:currentUser}  }  ); |

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/employees',  {  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/employees',  {  params:{username:currentUser}  }  );  this.user$.subscribe(data=>{  if(currentUser == data[0].username && currentPassword == data[0].password){  localStorage.setItem('validuser', currentUser);  }  }); |

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

|  |
| --- |
| this.user$ = this.http.get(  'http://localhost:3000/employees',  {  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/employees',  {  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.

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 just true or false because we would return from this function even before the user was checked. Also after the line were we use the next() method of the subject, we should execute subject.complete() but I am still experimenting with this.

# Appendix E – 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 } from '@angular/core';  import { FormGroup, FormControl, ReactiveFormsModule, Validators } from '@angular/forms';  import { Router } from '@angular/router';  //  @Component({  selector: 'app-login',  standalone: true,  imports: [ReactiveFormsModule],  templateUrl: './login.component.html',  styles: ``  })  //  export class LoginComponent {  //  loginForm = new FormGroup({  username: new FormControl('', Validators.required),  password: new FormControl('', Validators.required)  });  //  constructor(private router: Router) { };  onSubmit() {  let currentUser = this.loginForm.value.username;  let currentPassword = this.loginForm.value.password;  }  } |

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, private 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. Also delete the **user**$ if you have it still.

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

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

A screenshot of a computer

Description automatically generated

Change the variable ‘data’ to ‘status’ instead. Also move the **log()** method into a pair of curly braces:

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

Now we can check the status and redirect to the proper paths.

1. If NOT status, meaning if status is NOT positive, it means there was login failure, so keep user on login screen:

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

1. If status is ok, then login is ok and we can send them to home page:

|  |
| --- |
| 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');  });  } |

Note: generally whenever we subscribe to something, we should unsubscribe from that something. Take a look at Appendix C to see how I did it.

# Appendix F – 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 G – 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/employees',  {  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;  } |

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. A computer screen with text and images

   Description automatically generatedThe problem here is that we get false on the first try, and may occasionally get a true, the values are inconsistent:A screenshot of a computer

   Description automatically generated

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.

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**.

A screen shot of a computer code

Description automatically generated

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

# Appendix H – Subscribing to the Subject

When a subscription is created, a potential for memory leak is created. It is better to destroy subscriptions whenever they are created. For this simple example it is not necessary but here is how to handle it anyway.

1. In the login.component.ts file import the Subscription class from rxjs:

|  |
| --- |
| import { Observable } from 'rxjs/internal/Observable';  import { Subscription } from 'rxjs';  import { AuthService } from "./../auth.service"; |

In the class **LoginComponent** add a new Subscription object:

|  |
| --- |
| loginStatus$!:Observable<boolean>;  private lSubscription: Subscription = new Subscription;  constructor(private auth:AuthService, private router:Router) {  this.frmLogin = this.createFormGroup();  } |

In the **onSubmit**() method, store the Subscription into the variable you created in #2 above:

|  |
| --- |
| onSubmit(): void {  this.loginStatus$ = this.auth.login(this.frmLogin.value);  this.lSubscription = this.loginStatus$.subscribe(status=>{  if (!status)import { AuthService } from "./../auth.service"; |

Finally use the **ngOnDestroy**() method to clean up:

|  |
| --- |
| ngOnDestroy(): void {  this.lSubscription.unsubscribe();  } |