

Contents

PART 01 – API CALLS	2
PART 02 – OBSERVABLES	4
PART 03 – DISPLAYING DATA	5
PART 04 – POST REQUEST	7
PART 05 – LOGGING IN	10
PART 06 – PERSIST LOGIN	13
PART 07 – MOVING TO A SERVICE	16
PART 08 – ADAPT LOGIN TO USE SERVICE	20
PART 09 – (OPTIONAL) USING AN INTERFACE	22
APPENDIX A – SUBJECT	24
APPENDIX B – AUTH SERVICE EXPLAINED	24

Day02 Introduction to NG 13

PART 01 – API CALLS

1. Create a new component like we did on Day01 Part01 section 5. So:

```
ng g c employeelist
```

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

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

```
import { RegisterComponent } from "../register/register.component";
import { employeeListComponent } from
  "../employeelist/employeelist.component";

const routes: Routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'register', component: RegisterComponent },
  { path: 'employees', component: employeeListComponent }
];
```

Note: Angular may change the name of this component to employee-list

3. 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';
```

4. Add this module to the imports section:

```
],
imports: [
  BrowserModule,
  AppRoutingModule,
  ReactiveFormsModule,
  HttpClientModule
],
providers: [],
```

(remember to insert a comma at the line above)

5. Back in the child component, so employeelist.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({
```

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

```
export class employeeListComponent implements OnInit {  
  constructor(private http:HttpClient) {  
  }  
}
```

7. Within that class, so the employeeListComponent class, create a new property call it *employeeList*, it will be of the *any* data type:

```
export class employeeListComponent implements OnInit {  
  employeeList: any;  
  constructor(private http:HttpClient) { }  
  
  ngOnInit(): void {  
  }  
}
```

8. 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

```
employeeList: any;  
constructor(private http: HttpClient) { }  
  
ngOnInit() {  
  this.http.get('http://localhost:3000/employees');  
}
```

9. Assign the *observable* returned from the `get()` to *employeeList*. Do this in the `ngOnInit()` method.

```
ngOnInit() {  
  this.employeeList = this.http.get('http://localhost:3000/employees');  
}
```

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

```
ngOnInit(): void {  
  this.employeeList = this.http.get('http://localhost:3000/employees');  
  console.log(this.employeeList);  
}
```



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

PART 02 – OBSERVABLES

1. For now we can subscribe to the `employeeList` observable. 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/employees')  
        .subscribe();  
    console.log();  
}
```

2. Now inside of the subscribe method, we have access to the employees data, so provide a variable to store it:

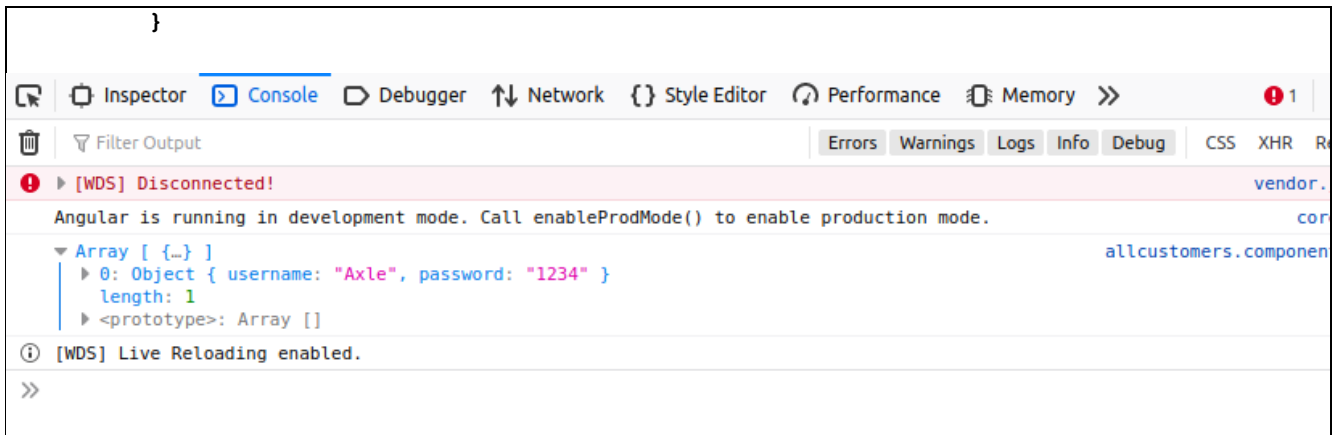
```
ngOnInit(): void {  
    this.http.get('http://localhost:3000/employees')  
        .subscribe(data => this.employeeList = data);  
    console.log();  
}
```

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/employees')  
        .subscribe(function(data) {  
            this.employeeList = data;  
        });  
}
```

3. 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/employees')  
        .subscribe(data => {  
            this.employeeList = data;  
            console.log(this.employeeList);  
        });  
}
```



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

PART 03 – DISPLAYING DATA

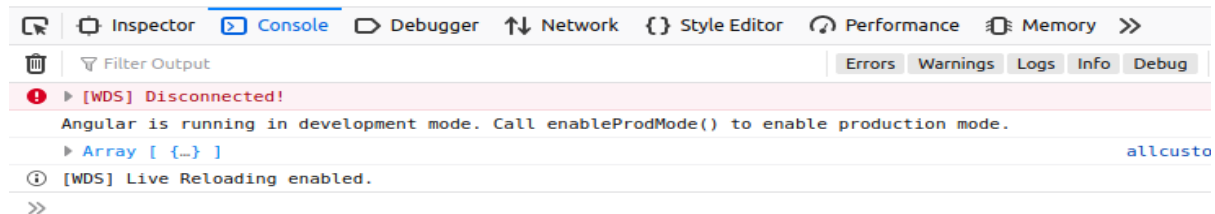
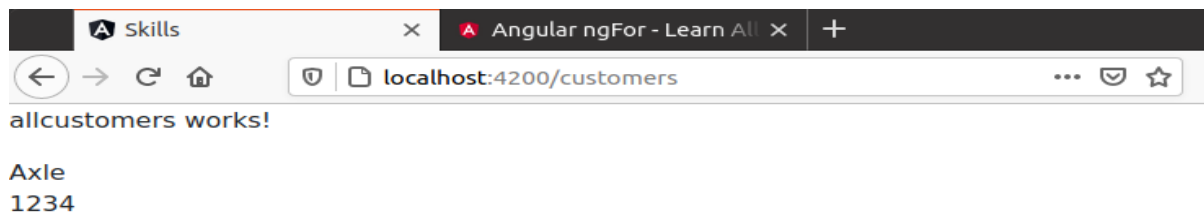
1. On the template, so on [employeeList.component.html](#) begin to add a few html tags to handle the array of data coming shortly:

```
<p>employeeList 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 the individual parts of the array, username and password

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

```
<p>employeeList works!</p>
<div *ngFor="let employee of employeeList">
  <div>{{employee.username}}</div>
  <div>{{employee.password}}</div>
</div>
```



3. All the code so far for the class:

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-employeeelist',
  templateUrl: './employeeelist.component.html',
  styleUrls: ['./employeeelist.component.css']
})
export class EmployeeelistComponent implements OnInit {
  employeeList: any;
  constructor(private http:HttpClient) { }

  ngOnInit(): void {
    this.http.get("http://localhost:3000/employees")
      .subscribe(data => {
        this.employeeList = data
        console.log(this.employeeList);
      });
  }
}
```

- Now let's do this a different way, using observables from the RxJS library. Return the import statement of `Observable` and re-create the `employeeList` 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-employeeList',
  templateUrl: './employeeList.component.html',
  styleUrls: ['./employeeList.component.css']
})
export class employeeListComponent implements OnInit {
  employeeList$: Observable<any>;
  constructor(private http:HttpClient) { }
```

Note, in order to prevent TS2564 error code, in VS Code, add the **!** after the `employeeList$` variable, so it should be **`employeeList$!`**

- Within the `ngOnInit()` method, all you have to do is pass the observable returned from the `get()` method straight onto the `employeeList$` property:

```
ngOnInit(): void {
  this.employeeList$ = this.http.get('http://localhost:3000/employees');
}
```

- Then, over in the template, just pipe the data through an `async` method:

```
<p>employeeList works!</p>
<div *ngFor="let employee of employeeList$ | async">
  <div>{{employee.username}}</div>
  <div>{{employee.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.

- 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"
  }
]
```

2. 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({
```

3. 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();
  }
}
```

4. Change the `onSubmit()` function to this:

```
onSubmit() {
  //console.log(this.frmRegister.value);
  this.http.post(
    'http://localhost:3000/employees',
    this.frmRegister.value
  );
}
```

5. 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/employees',
    this.frmRegister.value
  ).subscribe();
}
```






6. 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:


```
onSubmit(): void {
  //console.log(this.frmRegister.value);
  this.http.post<any>('http://localhost:3000/employees',
    this.frmRegister.value).subscribe(data => console.log(data));
}
```


Register in the Program

User name

Password

 Inspector  Console  Debugger  Network  Style Editor

 Filter Output Errors Warnings

► Object { username: "Johnny", password: "12345", id: "4poI8M0" }

Notice the new id showing up.

7. But this is not the recommended way, also we need to handle any errors. So since Angular used the RxJS library and there are only *three* ways to handle an observable, we will use two of those ways here, **next()** and **error()**. First the next() method will be implemented. First remove everything from between the subscribe method and replace it with a pair of curly braces:

```
onSubmit(): void {  
  //console.log(this.frmRegister.value);  
  this.http.post<any>('http://localhost:3000/employees',  
    this.frmRegister.value).subscribe({ } );  
}
```

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

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

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

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

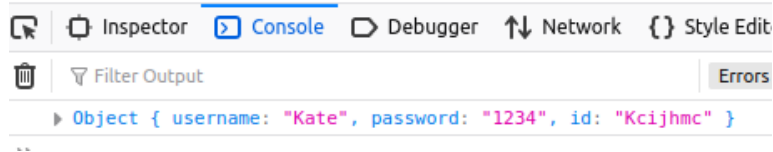
```
this.http.post<any>('http://localhost:3000/employees',  
  this.frmRegister.value).subscribe({  
    next: data => console.log(data),  
    error: err => console.log(err)  
  });
```

At this point, you may try inserting a new employee:

Register Form

User name

Password



Verify if the employee was added by going to localhost:3000/employees

The entire onSubmit() function

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

PART 05 – 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`
 - b. Import the HttpClient from `@angular/common/http` as well as all the Form modules from `@angular/forms`
 - c. copy all the code between the class definition and the end of the `register.component.ts` file
 - d. change the FormGroup property from `frmRegister` to `frmLogin`
 - e. remove everything from the `onSubmit()` function
 - f. remove the Observable object if you have one

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 {

  }
}
```

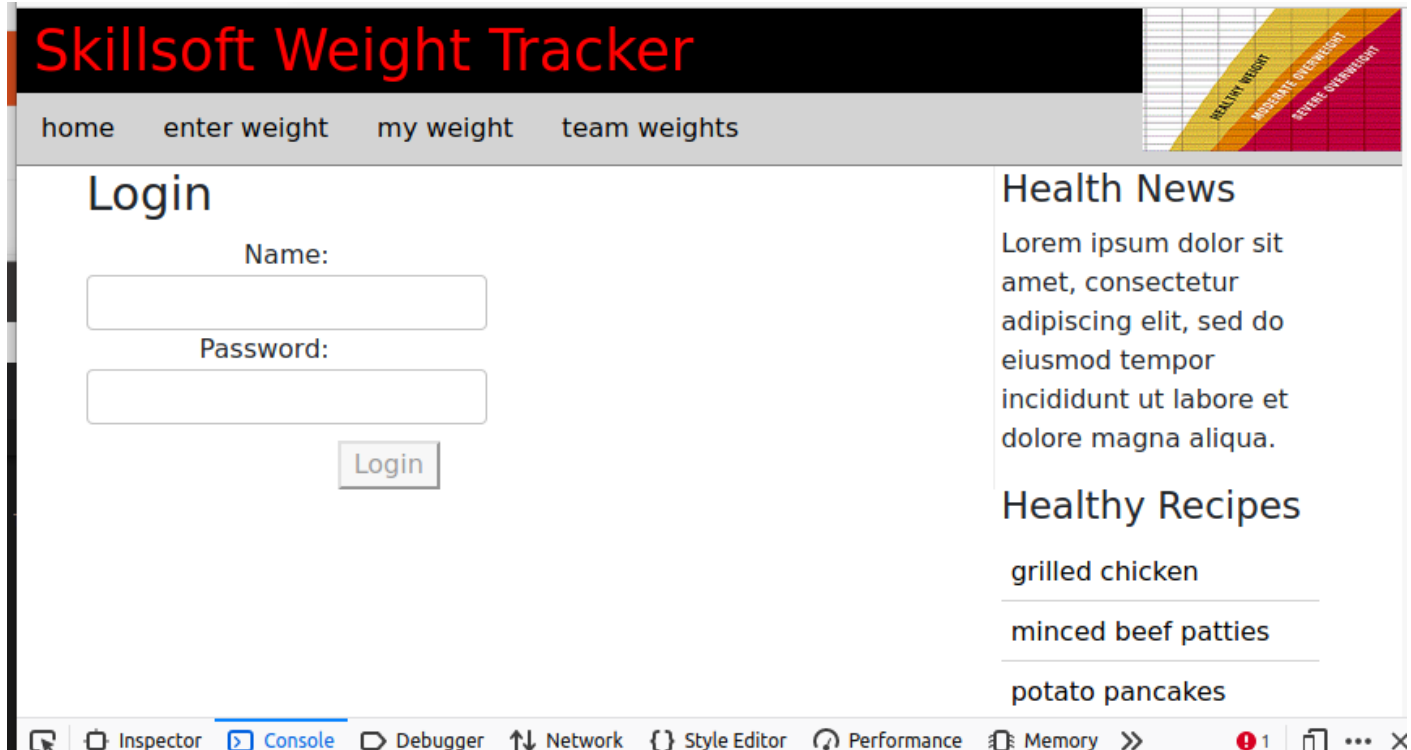
2. Now in the template, copy all the code from [register.component.html](#) to login.component.html, just change the formGroup name to frmLogin. Change all occurrences of frmRegister to frmLogin. Also change the <h2> tag to something appropriate for logging in.

3. Create a path for this component in the routing ts file:

```
import { employeeListComponent } from "../employeeList/employeeList.component";
import { LoginComponent } from "../login/login.component";

const routes: Routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'register', component: RegisterComponent },
  { path: 'employees', component: employeeListComponent },
  { path: 'login', component: LoginComponent }
];
```

4. 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.



The screenshot shows a web application titled "Skillsoft Weight Tracker" in a red font. Below the title is a navigation bar with links: "home", "enter weight", "my weight", and "team weights". The main content area is divided into two columns. The left column is titled "Login" and contains a form with two input fields labeled "Name:" and "Password:", and a "Login" button. The right column contains two sections: "Health News" with a paragraph of Lorem ipsum text, and "Healthy Recipes" with a list of three items: "grilled chicken", "minced beef patties", and "potato pancakes". The bottom of the image shows a browser's developer tools interface with tabs for Inspector, Console, Debugger, Network, Style Editor, Performance, and Memory.

Skillsoft Weight Tracker

home enter weight my weight team weights

Login

Name:

Password:

Login

Health News

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Healthy Recipes

- grilled chicken
- minced beef patties
- potato pancakes

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$` in the `login.component.ts` file 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

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

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

Remember to import the Observable module from rxjs

3. Next step, 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/employees');  
}
```

4. When we hit the `/employees` endpoint, if we just go with what we had in `employees` 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>

5. 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();
}
```

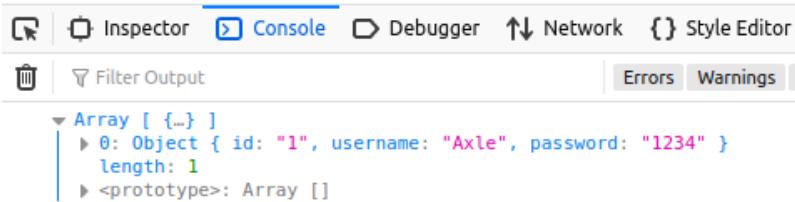
6. (Optional) check the data being returned using the console window:

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

Login

User name

Password



>>

7. 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");
    }
});

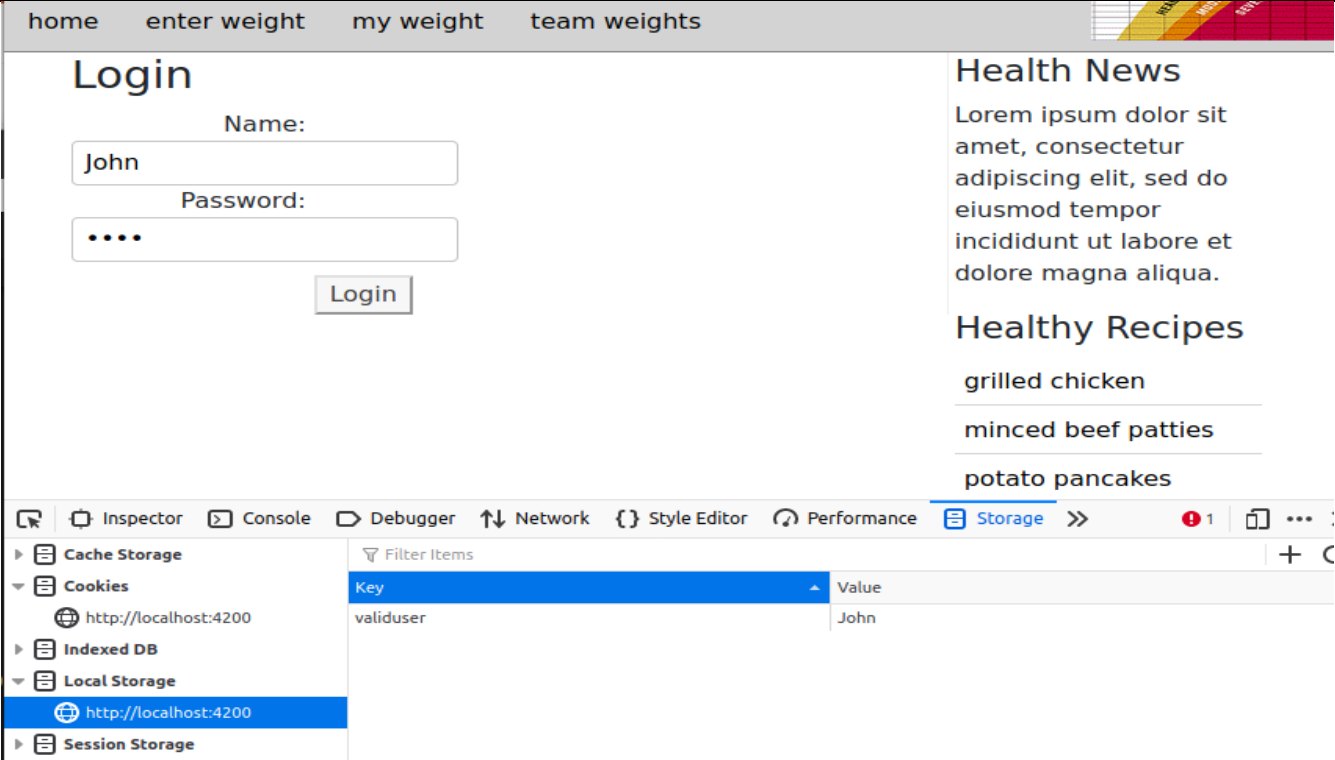
ngOnInit(): void {
```

8. 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!");
    }
});
```

9. 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");
})
```



home enter weight my weight team weights

Login

Name:

Password:

Login

Health News

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Healthy Recipes

- grilled chicken
- minced beef patties
- potato pancakes

Inspector Console Debugger Network Style Editor Performance Storage

Cache Storage Cookies Indexed DB Local Storage Session Storage

Filter Items

Key	Value
validuser	John

10. 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');
  }
})
```

11. 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` decorator that takes a metadata object. This tells Angular to inject this service where necessary and also to perform garbage collection.

2. 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) {

  }
}
```

3. 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 its `next()` method.

4. 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.

5. 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> {
```

6. 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>();
```

7. 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}  
    }  
  );
```

8. 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=>{
  });
```

9. 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);
  }
});
```

10. 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;
    }
  }
});
```

11. 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.

12. 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 where we use the `next()` method of the subject, we should execute `subject.complete()` but I am still experimenting with this.

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 {
  }
}
```

2. 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";
```

3. 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

4. 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.

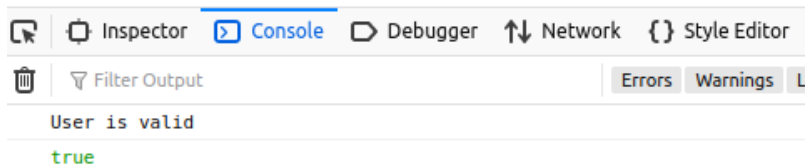
5. In the onSubmit() method, we can implement that service like we did in similar situations, remember to pass the user data contained in frmLogin:

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

Login

User name

Password



6. 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.

7. 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');  
  });  
}
```

8. 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.

PART 09 – (OPTIONAL) USING AN INTERFACE

1. In the employeeList.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 Employee {  
  username: string;  
  password: string;  
}  
  
@Component({}
```

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

```
export class employeeListComponent implements OnInit {  
  employeeList$: Observable<Employee>;
```

```
constructor(private http:HttpClient) {
```

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

```
export class employeeListComponent implements OnInit {  
  employees: Employee[] = [];  
  constructor(private http:HttpClient) {
```

4. In the ngOnInit() method we return the Employee[] type, which is an array:

```
  ngOnInit(): void {  
    this.employeeList$ = this.http.get<Employee[]>('http://localhost:3000/employees');  
  }
```

5. In this case we do not need the employeeList\$, just subscribe to the get() method. In the ngOnInit() method we return the Employee type, which is an array:

```
  ngOnInit(): void {  
    this.http.get<Employee[]>('http://localhost:3000/employees')  
      .subscribe(data => this.users=data);  
  }
```

In this

6. In the template we no longer have to use async also we use **employees** instead of the **observable**:

```
<div *ngFor="let employee of employees">  
  <div>Name: {{employee.username}}</div>  
  <div>Password: {{employee.password}}</div>  
</div>
```

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

```
<div id="container">  
  <main>  
    <div *ngFor="let employee of employees ">  
      <div>Name: {{employee.username}}</div>  
      <div>Password: {{employee.password}}</div>  
    </div>  
  </main>  
  <aside>  
    <section>
```

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

```
<main>  
  <div>  
    <h2>Our Employees</h2>  
  </div>  
  <div class="container" *ngFor="let employee of employees ">  
    <div class="row">  
      <div class="col-md-4 col-sm-6 col-xs-12">Name: {{employee.username}}</div>  
      <div class="col-md-4 col-sm-6 col-xs-12">Password: {{employee.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.

2. 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.

3. 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;
        }
      }
    });
```

4. 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');
};
```

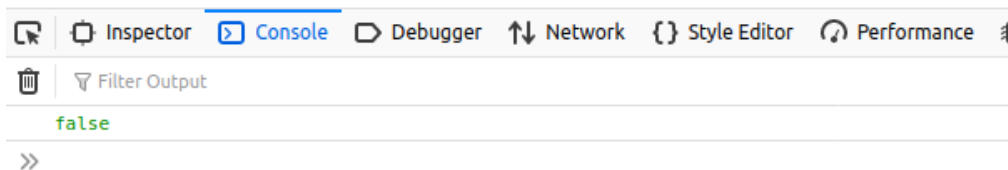
5. The problem here is that we get *false* on the first try, and may occasionally get a *true*, the values are inconsistent:

Login

Name:

Password:

Login



6. 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

```
8 export class AuthService {
9   isLoggedIn:boolean = false;
10  constructor(private http: HttpClient) { }
11
12  login(userData:any):boolean {
13    let currentUser = userData.username;
14    let currentPassword = userData.password;
15    this.http.get(
16      'http://localhost:3000/customers',
17      {
18        params:{username:currentUser} 2
19      }
20    ).subscribe(data=>{
21      if(data[0]){
22        if(currentUser == data[0].username && currentPassword == data[
23          localStorage.setItem('validuser', currentUser);
24          this.isLoggedIn = true;
25        }
26      } else{
27        this.isLoggedIn = false; 1
28      }
29    });
30    return this.isLoggedIn;
31  }
32 }
33
```

This code never got a chance to finish

This line will always execute first, returning with false

property has not yet gotten the chance to change based on the credentials we sent.

7. 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

```
25     this.user$.subscribe(data=>{
26         if(data[0]){
27             if(currentUser == data[0].username && currentPassword == data
28                 localStorage.setItem('validuser', currentUser);
29                 isLoggedIn = true;
30                 subject.next(isLoggedIn);
31             }
32         } else{
33             isLoggedIn = false;
34             subject.next(isLoggedIn);
35         }
36     });
37     return subject.asObservable();
38 }
39 }
```

that Subject.

8. Since the login() function must return an observable, it has to wait for the new observable to be created, `subject.next(isLoggedIn);` then return. This forces the entire function to wait until the Subject is resolved `return subject.asObservable();`

APPENDIX C – 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";
```

2. In the class LoginComponent add a new Subscription object:

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

3. 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.ISubscription = this.loginStatus$.subscribe(status=>{  
    if (!status)import { AuthService } from "../auth.service";  
  });  
}
```

4. Finally use the ngOnDestroy() method to clean up:

```
ngOnDestroy(): void {  
  this.ISubscription.unsubscribe();  
}
```