Contents

Day02 Introduction to NG 19

PART 01 – VALIDATION

1. In the **`Register`** component file, add the **`Validators`** module by importing it (if you do not already have it):

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule, Validators } from
'@angular/forms';

@Component({console.log(this.frmRegister.value);
  }
```

2. For each control, add the *required* **`validator`** in an array as the second argument to the **`FormControl()`** method. The first parameter to the method is empty for now, but you could pass in a prompt to the user using that method:

```
export class RegisterComponent {
  registerForm = new FormGroup({
    userName : new FormControl( '',[ Validators.required ] ),
    password : new FormControl( '',[ Validators.required ] )
  });
```

3. Move to the template and for our first validation we will disable the submit button unless something is entered in the fields:

```
          </div>
          <button
            type="submit"
            class="btn btn-primary"
            [disabled]="!registerForm.valid"
          >
            Submit
          </button>
        </form>
```

4. Let's add a message about this violation. This is the first real change for NG19. We get to use the **`@if`** construction. If you are on VS Code and it's updated, it will perform code completion for you:

```
 <input type="text" class="form-control" id="username" formControlName="username">
            @if (registerForm.controls['username'].errors) {
              <span>
                Invalid
              </span>
            }
          </div>
```

5. Do the same for password

```
          <div class="form-group">
            <label for="password">Password</label>
            <input type="password" class="form-control" id="password"
formControlName="password">
            @if (registerForm.controls['password'].errors) {
              <span>
                Invalid
              </span>
            }
          </div>
```

Test before moving to #6. Both fields should indicate *invalid* until a character is entered in the corresponding field.

6. The problem now is that the 'invalid' word appears as long as the form is on the screen. We can control the appearance of 'invalid' by implementing Angular's form control properties:

```
          <div class="form-group">
            <label for="username">User name</label>
            <input type="text" class="form-control" id="username"
formControlName="username">
            @if (
              (registerForm.controls['username'].errors)
              && (!registerForm.controls['username'].pristine)
            ) {
              <span>
                Invalid
              </span>
            }
          </div>
          <div class="form-group">
            <label for="password">Password</label>
            <input type="password" class="form-control" id="password"
formControlName="password">
            @if (
              (registerForm.controls['password'].errors)
              && (!registerForm.controls['password'].pristine)
            ) {
              <span>
                Invalid
              </span>
            }
          </div>
```

Notice that I wrapped each operand inside of parenthesis.

PART 02 – FIXING THE HTML AND COMPLETE ROUTING

1. Make the following changes to the home.component.html file:

```
</header>
<nav>
  <ul>
    <li><a href="index.html">home</a></li>
    <li><a href="enterweight.html">register</a></li>
    <li><a href="myweights.html">login</a></li>
  </ul>
</nav>
```

7. Also in home.component.html change the physical anchor link to use **routerLink**:

```
</header>
<nav>
  <ul>
    <li><a routerLink="/home">home</a></li>
    <li><a routerLink="/register">register</a></li>
    <li><a routerLink="/login">login</a></li>
  </ul>
</nav>
<div id="container">
```

Once you have made this change on the **home** component, do it to the other component i.e. *register*. We will do the login component tomorrow.

2. Make the following changes to the styles.css file:

```
header h1 a {
      color:red;
      text-decoration:none;
}
```

3. Comment out the following lines in the styles.css file:

```
/* label{
      display:inline-block;
      width:150px;
      text-align:right;
}

button{
      margin-left:155px;
      margin-top:10px;
}
 */
```

4. Make the following changes to the styles.css file:

```
nav{
	text-align:left;
	background-color:lightgray;
	border-bottom:1px solid gray;
	height: 46px;
}
…
#logo{
	float:right;
	width:160px;
	height:94px;
}
```

Feel free to change these numbers according to your browser and layout

5. For list items and anchor tags:

```
nav li, nav a {
	display:inline-block;
	text-decoration:none;
}}
```

One more thing…

It appears that we now have to include the RouterModule into the components where we are using routerLink. So far we have the home and register components so lets make this change there.

For example in the home component, add the highlighted line to your existing code:

```
import { Component } from '@angular/core';
import { RouterModule } from '@angular/router';
@Component({
  selector: 'app-home',
  imports: [RouterModule],
  templateUrl: './home.component.html',
  styleUrl: './home.component.css'
```

Remember to do this in each component that requires linking.

PART 03 – 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.

2. 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 }
];
```

3. 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: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    provideHttpClient()
  ]
};
```

   You should be prompted to do the import. Note, this can also be done in main.ts. Notice the providers[] array is now vertical instead of horizontal, this is for easy reading

4. 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';
```

5. 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`.
  }
```

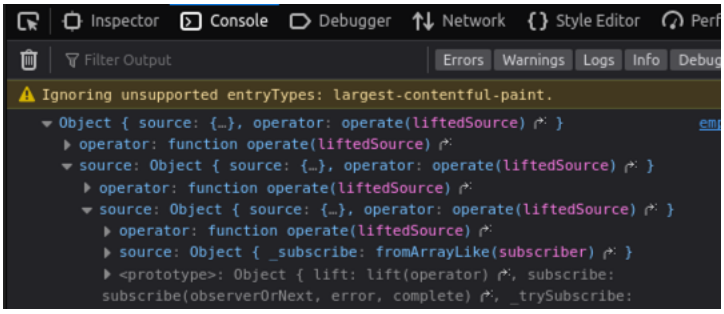
6. Since the **`get()`** method of the **`HttpClient`** returns an **`Observable`**, let us create a variable of that type and have it be of the any type:

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

Remember to import Observable from rxjs.

7. 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:4201/employees');
    console.log(this.employees$);
  }
```



Remember to implement the OnInit() method by importing OnInit from @Angular/core and writing the ngOnInit() method in the EmployeesComponent class.

8. You can also now add a path and a link for our new component. In app.routes.ts file add the EmployeesComponent and create a path in the Routes array:

```
export class EmployeesComponent {
import { HomeComponent } from "./home/home.component";
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 can now use the same observable, **employees$** for our template output. NG19 now has the **@if** and **@for** control flow available. First we must import the CommonModule from @angular/common and add it to the imports of the employees component:

```
import { Observable } from 'rxjs/internal/Observable';
import { CommonModule } from '@angular/common';
@Component({
  selector: 'app-employees',
  imports: [CommonModule],
```

Notice that only the CommonModule is in the imports array. This is part of Angular's dependency system, this module is importable as opposed to Observable which is declarable.

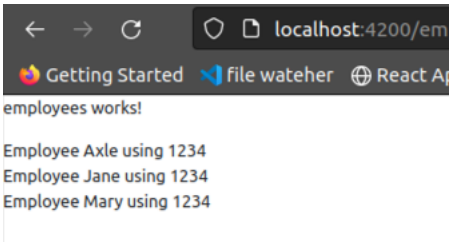
2. Add these lines of code in the template:

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

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

3. 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>
}
```



4. 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.

5. 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:4201/employees');
  }
```

PART 05 – POST REQUEST

So far 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. With no module support in NG19, 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 imported it in Part 01 #3.

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** in the register component:

```
import { FormGroup,FormControl, ReactiveFormsModule, Validators } from
'@angular/forms';
import { RouterLink } from '@angular/router';
import { HttpClient } from "@angular/common/http";
```

We do this in register.component.ts file

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

```
export class RegisterComponent {
  constructor(private http:HttpClient) {};
  registerForm = new FormGroup({
```

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

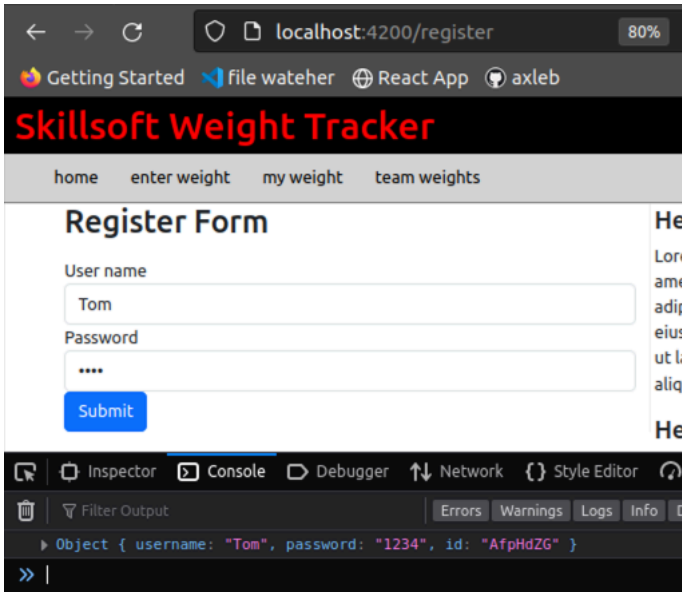
```
  constructor(private http:HttpClient) {};
  onSubmit() {
    this.http.post(
      'http://localhost:4201/employees',
      this.registerForm.value
    )
  }
```

5. 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:4201/employees',
      this.registerForm.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:

```
  constructor(private http:HttpClient) {};
  onSubmit() {
    this.http.post(
      'http://localhost:4201/employees',
      this.registerForm.value
    ).subscribe(response => console.log(response));
  }
```



Notice the new id showing up.

Also if you refresh localhost port 4201 you should see the new employee Tom showing up as the fourth employee.

7. 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, as a first parameter of the **subscribe()** method add a pair of curly braces:

```
    this.http.post(
      this.registerForm.value
    )
    .subscribe({}response => console.log(response));
  };
```

8. Now we can work with either of the three observable methods or any combination, so first the **next**() method, just cut and paste the **response** as the **value** part of **next():**

```
  onSubmit() {
    )
    .subscribe(
      {
        next : response => console.log(response)
      });
  };
```

If you run the app at this point, a similar result to what we got 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:4201/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:4201/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:

```
     'http://localhost:4201/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 06 – 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`**
   b. Import the **`HttpClient`** from **`@angular/common/http`** as well as all the FormGroup, FormControl, ReactiveFormsModule and Validators from **`@angular/forms`** (in login.component.ts). Also import RouterLink from @angular/router.
      In the imports array, within the @Component decorator function, add the **`ReactiveFormsModule`** and **`RouterLink`**
   c. Copy all of the code between the class of **`Register`** and past it between the class definition of Login
   d. change the FormGroup property from **`registerForm`** to **`loginForm`**
   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:

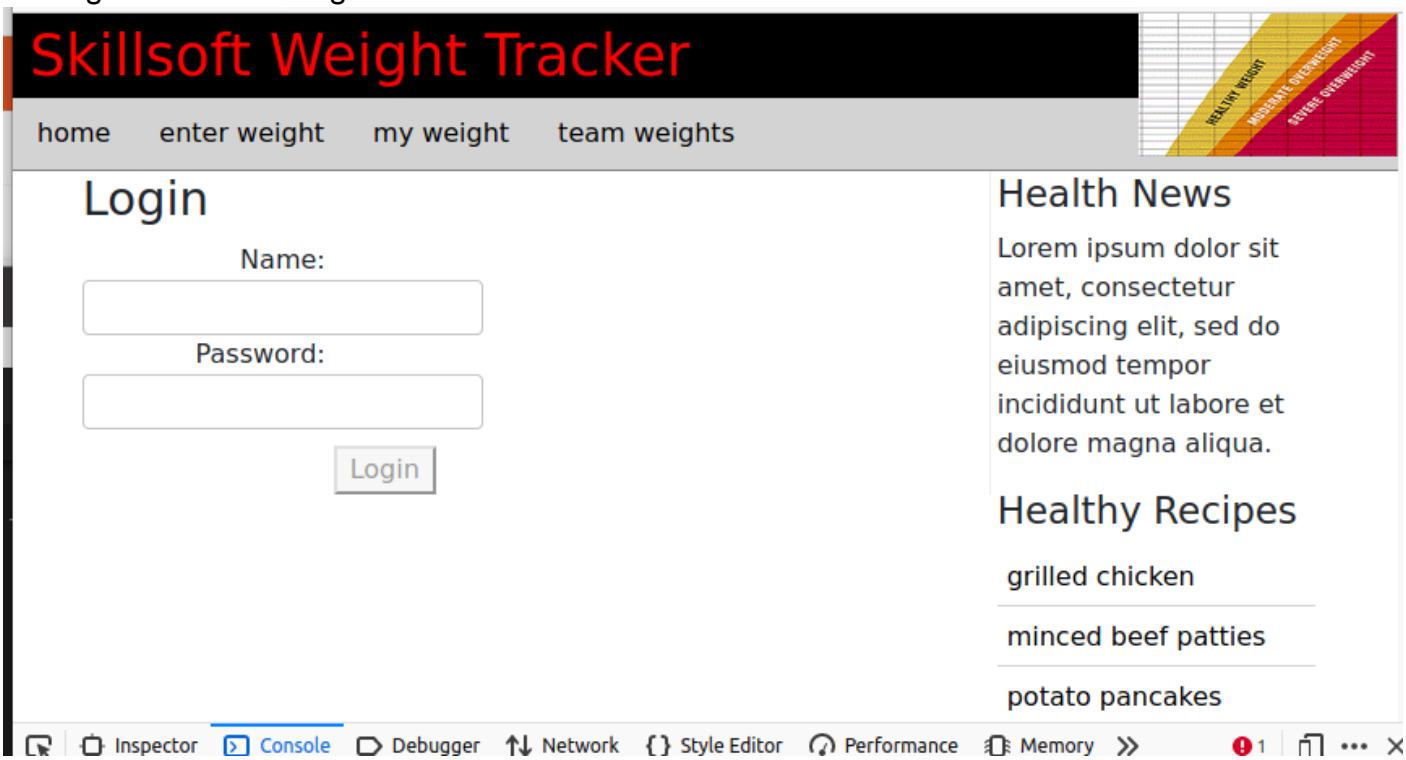
```
import { Component } from '@angular/core';
import { HttpClient } from "@angular/common/http";
import { RouterLink } from '@angular/router';
import { FormGroup,FormControl, ReactiveFormsModule, Validators } from '@angular/forms';
//
@Component({
  selector: 'app-login',
  imports: [ReactiveFormsModule, RouterLink],
  templateUrl: './login.component.html',
  styles: ``
})
//
export class LoginComponent {
 constructor(private http:HttpClient) {};
  //
  loginForm = new FormGroup({
    userName : new FormControl( '',[ Validators.required ] ),
    password : new FormControl( '',[ Validators.required ] )
  });
//
  onSubmit() { …
```

2. 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.

3. 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 }
];
```

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.



PART 07 (OPTIONAL) – 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

2. 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:4201/employees');
  }
```

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

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

```
export class LoginComponent {
  user$!: Observable<any>;
  currentUser = "";
  currentPassword = "";
  constructor(private http:HttpClient) {};
    //
    loginForm = new FormGroup({
```

4. 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:4201/employees',
      {
        params:{userName:currentUser!}
      }
    );
  }
```

Notice the params object, this is so that we can achieve something like this: http://localhost:4201/employees/?username=John. Also since TS thinks that the object may be null, we add a ! to the end. Finally, the key/value pair must be spelled to match JSO file.

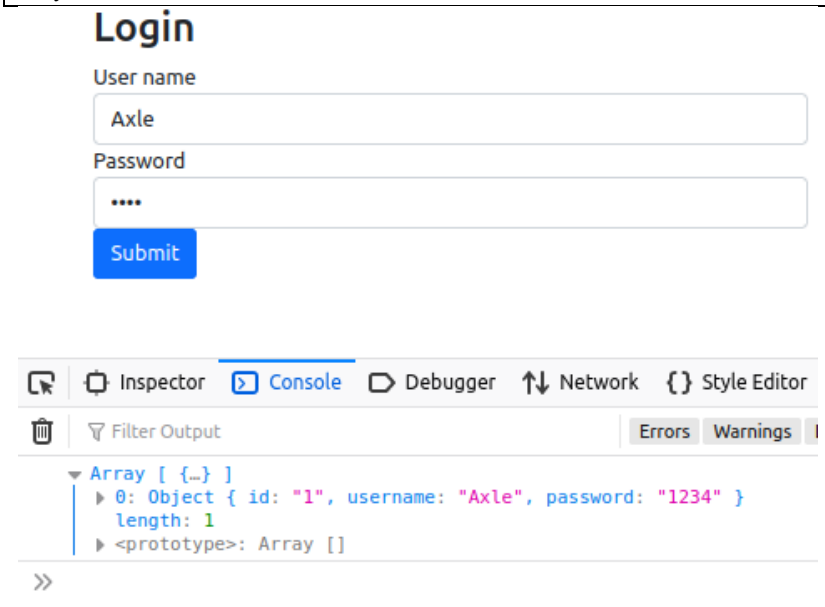
5. Now we subscribe to the **user$** in order to work with the return from our **get**() call

```
this.user$ = this.http.get('http://localhost:4201/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)});
  }
```



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

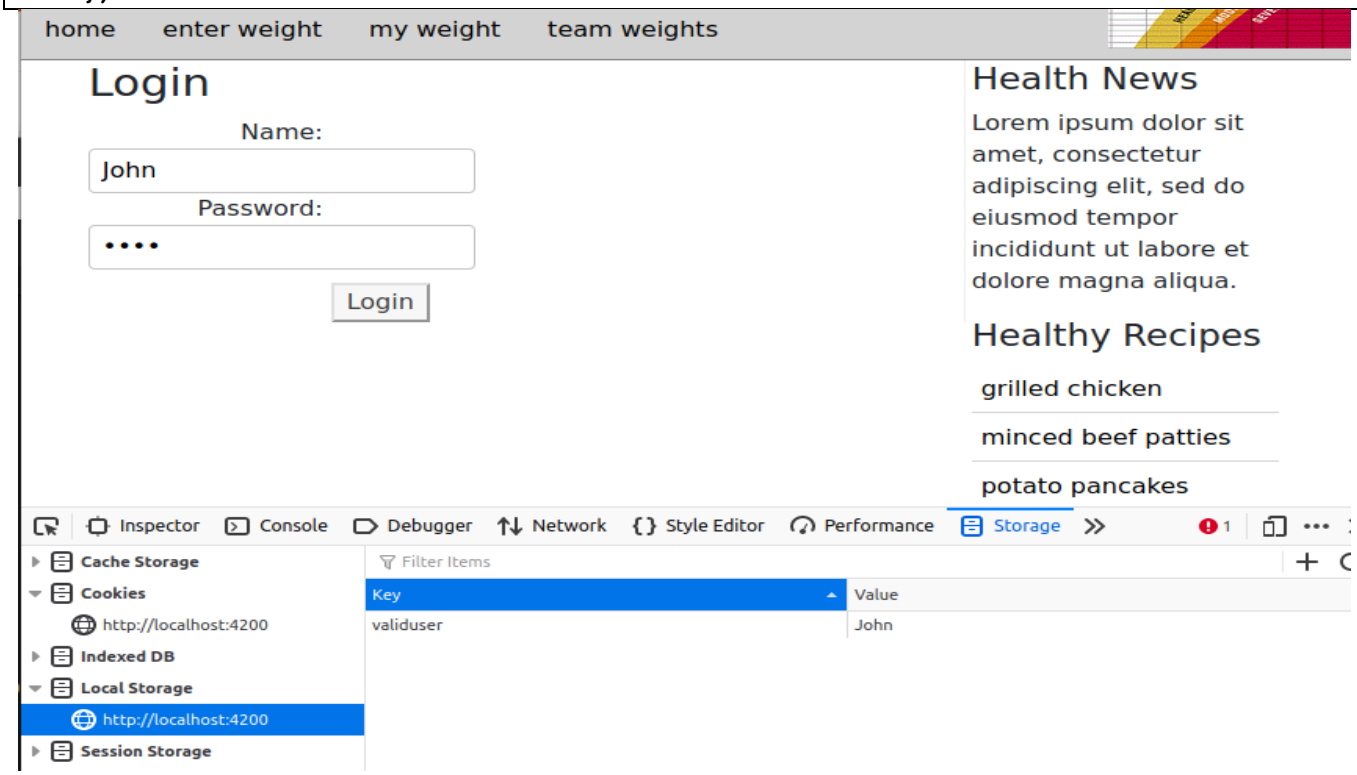
```
this.user$.subscribe(data=>{
  if(currentUser == data[0].username && currentPassword == data[0].password){
        //we have a match
        console.log("User is valid");
      }
    });
  }

  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");
    })
```



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 08 (OPTIONAL) – 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 needed 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 AuthService:

```
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 that can also emit data.

4. The first function in our service class, the **`login()`** function, should be 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.

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> {
```

Note if you get an error under user$ to the effect of *Property 'user$' has no initializer and is not definitely assigned in the constructor*, just add this key/value pair to your tsconfig.js file. { "strictPropertyInitialization":true} do this under **Compiler Options**. This is one of many ways of removing this error. Alternatively add an exclamation point next to the definition.

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 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;
          }
```

Now, how do we make 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 true or false because we would return from this function even before the user was checked

PART 09 (OPTIONAL) – ADAPT TO A 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 { RouterLink, Router } from '@angular/router';
import { FormGroup,FormControl, ReactiveFormsModule, Validators } from '@angular/forms';
import { Observable } from 'rxjs/internal/Observable';
@Component({
  selector: 'app-login',
  imports: [ReactiveFormsModule, RouterLink],
  templateUrl: './login.component.html',
  styles: ``
})
export class LoginComponent {
  currentUser = "";
  currentPassword = "";
  constructor( private router:Router) {};
```

```
  loginForm = new FormGroup({
    userName : new FormControl( '',[ Validators.required ] ),
    password : new FormControl( '',[ Validators.required ] )
  });
  onSubmit() {
    let currentUser = this.loginForm.value.userName;
    let currentPassword = this.loginForm.value.password;
  };
};
```

2. First import the service into the login component:

```
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, 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, private auth:AuthService) {
    this.frmLogin = this.createFormGroup();
  }
  createFormGroup() {
```

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

5. 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));
  }
```

6. 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');
    });
  }
```

APPENDIX A – FORM AND CONTROLS STATUS

In Angular, form control statuses are properties that automatically track the state of a form field or the form itself. You can access these form and form control properties from the Inspector tab in Mozilla Firefox.

pristine vs. dirty: This pair of statuses tracks the state of a form control's value. A control is pristine if it is just loaded in the browser and becomes dirty as soon as the user changes its value.

untouched vs. touched: This pair tracks whether the user has interacted with the control. A control starts as untouched but a click in or on the control changes that. It becomes touched when the user first focuses on the field and then moves the mouse to away from the control.

valid vs. invalid: This pair reports on the control's validation status. The control is valid if it satisfies all the validation rules set by the developer. If the value that the user enters does not comply with any rule of the control, it becomes invalid.

pending: This status is advanced and is used for asynchronous validators. When an async validation is in progress the control's status changes to pending.