Table of Contents

[Part 1 Project Setup 2](#_Toc153451126)

[Part 2 NgRx 6](#_Toc153451127)

[Part 3 Implementing the NgRx Store 9](#_Toc153451128)

[Part 4 Using Profile Service with NgRx Store 12](#_Toc153451129)

[Part 5 Dispatching Actions 14](#_Toc153451130)

[Part 6 Signals - Intro 16](#_Toc153451131)

[Part 7 Signals – Objects and Mutate 18](#_Toc153451132)

[Part 8 Signals – Converting the Profile Service to use Signals 20](#_Toc153451133)

[Part 9 Signals – Completing Profile Service to use Signals 22](#_Toc153451134)

[Appendix A Logging Out with Actions 25](#_Toc153451135)

[Appendix B – Signals 29](#_Toc153451136)

Day02 State Management

# Part 1 Project Setup

There is a starter project for today. It is called ng16-day02-starter-skills.zip. Unzip that file, rename the folder to just skills and run the npm install command against that folder. After that run and ng build or ng serve. You must use the name *skills*. The app does not do anything at the moment.

In this boot camp, we will end up with several project folders. It won’t be just one folder called skills, there will be at least three others.

The project folder contains a basic boilerplate app, three components and a not-found single component. The project folder also contains a Profile service with a constructor and three methods. The Subject class has been imported from the RxJS library and is being used to return a value. The isLoggedIn property is of this Subject type that handles Boolean data. This property can now be used to emit a piece of data, *true* or *false*, based on whether the login or logout methods are called.

The loginStatus()method simply returns the isLoggedIn property as an Observable type.

This service has already been injected into the app component and a property userLoggedIn has been declared and initialized with the false value.

The future plan is to use the profile service to call an external API service to see if the user is on file and if she can be logged into the site. For now, we will NOT call an API but pretend that login/logout is being carried out.

Please note that there are several ways to perform authorization and authentication including using the local storage of the browser and third party services.

In Part 1 of the bootcamp, I am trying to show that we need some kind of mechanism to keep the state of the logged in user. That mechanism currently the Observable. With the observable, we can set and determine state.

1. In the ngOnInit() method of the app component, first we use the Profile service to login the user. Then we subscribe to the isLoggedIn method and assign the returned value to our local property userLoggedIn. Finally we print the local property to see what value we get:

|  |
| --- |
| ngOnInit(): void {  this.pService.login();  this.pService.isLoggedIn.subscribe(status => {  this.userLoggedIn = status;  });  console.log(this.userLoggedIn);  }  loginCheck(){ |

This should return false in the Console window. These changes are being done in the app.component.ts file. The Profile Service was injected via the constructor. Of course we should be using the login component to do this, but we will get to that shortly.

1. Now toggle the pService call from login() to logout() and check the Console window of your browser:

|  |
| --- |
| ngOnInit(): void {  this.pService.logout();  this.pService.isLoggedIn.subscribe(status => {  this.userLoggedIn = status;  });  console.log(this.userLoggedIn);  }  loginCheck(){ |

This should also return false in the Console window.

1. So, both methods produce the same result, *false*. However, if we change our strategy and use a BehaviorSubject in the service, the result is quite different. In the profile.service.ts file, change Subject to BehaviorSubject and point userLoggedIn property to this new class:

|  |
| --- |
| import { Injectable } from '@angular/core';  import { BehaviorSubject } from "rxjs";  @Injectable({ providedIn: 'root' })  export class ProfileService {  userLoggedIn = new BehaviorSubject<boolean>(false);  constructor() { } |

A BehaviorSubject takes an initial value and it remembers the last emission. Remember to import that class from the **rxjs** library.

1. With this new Observable and the Boolean property on our component class, we can restructure the app template to reflect login/logout. Basically if the user is logged in already, then show *logout* otherwise show *login,* so we add a new <li>:

|  |
| --- |
| <li class="nav-item">  <a  class="nav-link"  routerLinkActive="active"  routerLink="/logout"  [hidden] = "!userLoggedIn"  >  logout  </a>  </li>  <li class="nav-item">  <a  class="nav-link"  routerLinkActive="active"  routerLink="/login"  [hidden] = "userLoggedIn"  >  login  </a>  </li> |

Note, we do not yet have a logout view, so the app will default to the not-found component if you tried to go to /logout in the browser.

1. Add the hidden attribute and bind it to the userLoggedIn Boolean property on the class. Add a second <li> to handle the logout mode. In this way if userLoggedIn is false, it means that the user logged out or there is no user. In this case show *login* button. The opposite if a user is logged in.
2. At start of the app, no user is logged in so we show the login button. Once we click that button we are taken to the login component. We need the profile service here to actually login a user. So in the login.component.ts file, import the ProfileService service and inject the service into the class:

|  |
| --- |
| import { Component } from '@angular/core';  import { ProfileService } from "./../profile.service";  @Component({ |

1. In the component, add a function to login the user, using the Profile service:

|  |
| --- |
| constructor(private pService : ProfileService){  }  loginUser(): void {  this.pService.login();  } |

Also, create an object based on the ProfileService class via the constructor.

1. In the login.component.html file add a button to

|  |
| --- |
| <p>login works!</p>  <button (click)="loginUser()">Login</button> |

Clicking the button does call the Profile login function and it updates the menu logout button. If the login() method of the service is called, it updates the userLoggedIn property in the app component, which updates the template. In #2 above, we had initially called login()/logout(). Comment these lines or remove them. You may also add this line in the login component, just to confirm that the isLoggedIn property of the service is bein updated: this.pService.isLoggedIn.subscribe(x => console.log(x));

**The next two steps are Optional.**

It is a good idea to remove the subscription to the pService once the component is destroyed.

1. A good place to do this is in the parent component, app. This is a multi-step process, first import the OnDestroy and Subscription modules:

|  |
| --- |
| import { Component, OnInit, OnDestroy } from '@angular/core';  import { ProfileService } from "./profile.service"; import { Subscription } from 'rxjs'; |

Note, rxjs is NOT part of Angular core, it is a third party library.

1. Next, adjust the class declaration to include the OnDestroy implementation:

|  |
| --- |
| export class AppComponent implements OnInit, OnDestroy{  title = 'profile'; |

1. Next, create a subscription property of the class. This property must be of the Subscription type:

|  |
| --- |
| userLoggedIn : boolean = false;  private loginSubscription! : Subscription;  constructor(private pService : ProfileService){ |

1. Next, re-architect the way the subscription is created:

|  |
| --- |
| ngOnInit(): void {  this.loginSubscription = this.pService.isLoggedIn  .subscribe(status => {  this.userLoggedIn = status;  });  console.log(this.userLoggedIn); |

1. Finally, create an ngOnDestroy() method and run the unsubscribe() method against the loginSubject you created in #10 above:

|  |
| --- |
| ngOnDestroy(){  this. loginSubscription.unsubscribe();  } |

Note, there are other ways of doing this. For example it may be possible to use the takeUntil() operator piped through via the subscription.

Also, the login status will not change as we have not printed the status after calling the login() method. So what you see in the console window is the default false value.

# Part 2 NgRx

Using the RxJS library to manage state in larger and more complex applications is not the most efficient method. Angular has partnered with NgRx for a more robust solution. NgRx not only manages state in Angular (and React) applications, but it also manages asynchronous operations. It offers a consistent data flow and most importantly a **centralized** store.

There are several ways to implement NgRx in Angular, this is just a simple demo. The main parts of an NgRx solution involve actions, reducers, selectors and effects. This demo does not use effects. Effects are mainly used to interact with a database.

NgRx is more of a pattern than a platform, it’s a pattern backed up by a library. It dispatches actions to affect change on a state and utilizes reducers, selectors and a centralized store to help with state changes.

Note Angular 17 has now introduced Signals that may be an alternative to NgRx in most cases. However we may continue to use a combination of NgRx, RxJS and Signals in the short term.

1. Continue with the app from Part1 and Install the NgRx package via a terminal window using npm:

|  |
| --- |
| npm install @ngrx/store |

If you get this error while trying to install this package: *ERESOLVE unable to resolve* *dependency tree*, then install with a flag: npm install @ngrx/store --legacy-peer-deps

1. There are several starting points but I will start with the reducer. Reducers will help us decide what actions we want to perform in our app. Create a .ts file inside of the app folder called profile.reducer.ts:

|  |
| --- |
|  |

Reducers are just functions that affect change to some object, usually called a store. There is a createReducer class in @ngrx/store, but we wont be using that class here. There is also an on() method that can be utilized.

1. In the reducer file, start with the initial state which should be an object:

|  |
| --- |
| const initialState = {  loginStatus : false  }; |

1. Next add a function called profileReducer:

|  |
| --- |
| export function profileReducer(){  } |

1. That function takes two arguments, the first parameter gets the initial state:

|  |
| --- |
| export function profileReducer ( state = initialState, action ) {  } |

Note, this function gets called by the NgRx package and will therefore supply both parameters. The second parameter is responsible for changing the initial state.

1. The second parameter is of type Action. We can import this Action class via the @ngrx/store package:

|  |
| --- |
| import { Action } from "@ngrx/store";  const initialState = { |

This should also return false in the Console window.

1. Now our action parameter is going to be of this Action type:

|  |
| --- |
| export function profileReducer( state = initialState, action:Action ){  } |

This should also return false in the Console window.

1. The reducer performs its tasks based on Actions. We have the ability to define our actions. For now, we have just the one action, login the user:

|  |
| --- |
| export function profileReducer(state = initialState, action : Action){  if(action.type == "LOGIN\_USER"){  return {  loginStatus : true  }  } else {  return { loginStatus : false }  }  } |

1. Create a new file called profile.actions.ts. This will eliminate the future typos when it comes to actions:

|  |
| --- |
| A screenshot of a computer program  Description automatically generated |

Note: there is a createAction class in @ngrx/store, but we are not using that here.

1. In this new file, import the Action class like you did in #6 and at the same time export a const value of the type of action we wish to perform:

|  |
| --- |
| import { Action } from "@ngrx/store"; export const LOGIN\_USER = 'LOGIN\_USER'; |

1. Adjust the reducer file to use the actions file:

|  |
| --- |
| import { Action } from "@ngrx/store";  import { LOGIN\_USER } from "./profile.actions";  const initialState = { loginStatus : false };  export function profileReducer( state = initialState, action : Action ) {  if ( action.type == LOGIN\_USER ) {  return { |

The LOGIN\_USER is a constant. We will build a class in the actions file shortly.

# Part 3 Implementing the NgRx Store

In order to use the files we created so far, we should start with the app.module.ts file. We need to configure this driver file, to accommodate our changes and allow the use of this different type of state management (NgRX).

Please note that Part 3 will be used again in Part 8. Save this folder and this step.

1. Inside of the app.module.ts file, import the StoreModule from @ngrx/store:

|  |
| --- |
| import { NgModule } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { StoreModule } from "@ngrx/store";  import { AppRoutingModule } from './app-routing.module'; |

1. Also import the reducer TS file:

|  |
| --- |
| import { StoreModule } from "@ngrx/store";  import { profileReducer } from "./profile.reducer"; |

1. Move to the imports array of @NgModule and add the StoreModule as an element:

|  |
| --- |
| ],  imports: [  BrowserModule,  AppRoutingModule,  StoreModule.forRoot( ),  ],  providers: [], |

1. The forRoot() method takes an object in which you supply the key. That key must point to the profileReducer we imported in #2:

|  |
| --- |
| imports: [  BrowserModule,  AppRoutingModule,  StoreModule.forRoot( {loginUser : profileReducer} ),  ],  providers: [], |

The use of the forRoot() method here forces the singleton pattern for loginUser.

1. To use our new feature, in the profile service component, import the Store class from @ngrx/store, and the Observable class from rxjs:

|  |
| --- |
| import { Injectable } from '@angular/core';  //import { BehaviorSubject } from "rxjs";  import { Store } from "@ngrx/store";  import { Observable } from "rxjs"; |

We don’t need the BehaviorSubject anymore. Remember this is in the profile.service.ts.

1. Comment all methods except the constructor method and the loginStatus() method. We will focus on one method at a time. In the constructor inject the Store class as before:

|  |
| --- |
| export class ProfileService {  //isLoggedIn = new BehaviorSubject<boolean>(false);  constructor(private store : Store) { }  //login(){ |

Also comment the isLoggedIn property for now. Store can also be used to dispatch actions.

1. The Store now needs to indicate what type of data it is handling. We use generics for this. Now, in this case the data must conform to the shape we established in the app.module.ts file:

|  |
| --- |
| export class ProfileService {  //isLoggedIn = new BehaviorSubject<boolean>(false);  constructor(private store : Store < { loginUser : { loginStatus : boolean } } > ) { }  //login(){ |

This type was declared in #4 above.

1. In the loginStatus() method we will now return the status stored in the Store, **not** the isLoggedIn property value:

|  |
| --- |
| loginStatus(){  return this.store.select();  //return this.isLoggedIn.asObservable();  }  //register(){ |

In addition to reducers and actions there are selectors, as shown above.

1. The select() method acts like a *getter* method. It returns whatever Store property we setup and it’s value. In this case there is only one, the loginUser, so pass that into the select() method and return everything as an Observable of the same type as in the constructor:

|  |
| --- |
| loginStatus(){  return this.store.select('loginUser') as Observable < { loginStatus : boolean } > ;  //return this.isLoggedIn.asObservable();  }  //register(){ |

What we are really saying here is that we know there is a piece of data in storage called loginUser. The select() method will go find it and return its value as an Observable. However the value that loginUser points to, is itself an object with a key/value pair. The key is loginStatus and whatever value it turns out to be, that value is either *true* or *false*. The as Observable part here is redundant.

If you try to run the app, it may fail. You can continue on or comment out certain lines such as the login() method in the LoginComponent. Also comment the entire ngOnInit() method in the app.component.ts file.

# Part 4 Using Profile Service with NgRx Store

In order to use the Store we created in parts 2 and 3 we must use it via the Profile service. We implemented the profile service in the app.component.ts file to check the status of the user. Now we get the same result but with a different architecture.

1. One benefit of this architecture is that we do not have to manage subscriptions to Observables. We can remove the onDestroy and Observable classes we imported into the app component. Also you can completely remove the ngOnDestroy() method we implemented. Now the NgRx Store will manage our observables:

|  |
| --- |
| import { Component, OnInit, ~~OnDestroy~~ } from '@angular/core';  import { ProfileService } from "./profile.service";  import { ~~Observable~~, Subscription } from 'rxjs';  …  ~~ngOnDestroy(){~~  ~~this.loginSubject.unsubscribe();~~  ~~}~~ |

1. The userLoggedIn property no longer points to boolean value, now it points to an object that contains a boolean value:

|  |
| --- |
| export class AppComponent implements OnInit{  userLoggedIn! : { loginStatus : boolean };  private loginSubject = new Subscription();  constructor(private pService : ProfileService){ |

Actually it could remain as just a boolean property and we can assign status.loginStatus from our Store. But the object works better.

1. We do not have to store the Subscription in loginSubscription since NgRx Store will manage our Observables. Everything else in this method stays mainly the same except for the method we call on our pService object. Now we call the loginStatus() method. That method will now return an object, NOT a boolean value. Also, remember from part3 #9, that the *status* is being returned via an Observable. We have to subscribe to get the value from that Observable. I added in a log() here just to demonstrate the object that gets returned from our service:

|  |
| --- |
| constructor(private pService : ProfileService){  }  ngOnInit(): void {  this.pService.loginStatus()  .subscribe( status => {  this.userLoggedIn = status ;  console.log(status);  }); |

Remember userLoggedIn is of type Object NOT just a boolean. If we wanted to use a boolean, we would do this assignment: this.userLoggedIn = status.loginStatus;

1. Since our app component will execute the ngOnInit() method every time the component gets rendered, we can check the console window to see the result of the console.log():  
   A screenshot of a computer error

   Description automatically generated
2. There is one last piece to this store puzzle. Remember that the template for the basic app is directly dependant on the value in userLoggedIn. But now since we changed everything, this property now returns an object. We therefore cannot use this property as is because it will yield the incorrect value. Instead change it as shown below:

|  |
| --- |
| <li class="nav-item">  <a  class="nav-link"  routerLinkActive="active"  routerLink="/logout"  [hidden] = "!userLoggedIn.loginStatus"  >  logout  …  routerLinkActive="active"  routerLink="/login"  [hidden] = "userLoggedIn.loginStatus"  >  login  </a> |

It is loginStatus that will yield the Boolean value. The property userLoggedIn refers to an object, so that by itself will always return true.

1. The app will not work fully as we do not yet have a login method on the service. For now just comment out the loginUser() method in the login.component.ts file.

# Part 5 Dispatching Actions

Up to this point, we can work with our Store. A Store is a place where we can keep several items, not just the status of the user logging in or out. This means that it should behave just like a database. The Boolean value of loginStatus is initially supplied the false value due to defensive programming and security. But what happens if we want to change that value. Well that’s where *dispatching actions* come in. Dispatching an action is just a fancy way of calling a method.

We started the action file in Part 2 of today’s bootcamp.

1. The obvious place to dispatch an action is the place where the change originates. Well in our case the change originates when the user logs in via the login component. However it is the Profile service that does the heavy lifting. So lets dispatch our action from there:

|  |
| --- |
| constructor(private store : Store<{loginUser : {loginStatus : boolean}}>) { }  login(){  this.store.dispatch( );  }  logout(){ |

Another reason to use the Profile service is that the Store is already set up there.

1. In Part2 #10 when we created the actions file, we imported the Action class from @ngrx/store but did not use it. Let’s now export a class based on this imported file. In profile.actions.ts file, add these lines:

|  |
| --- |
| import { Action } from "@ngrx/store";  export const LOGIN\_USER = 'LOGIN\_USER';  export class LoginUser implements Action {  readonly type = LOGIN\_USER;  } |

1. Back to the service, import LoginUser from #2 it at the top of our Profile class:

|  |
| --- |
| import { Store } from "@ngrx/store";  import { Observable } from "rxjs";  import { LoginUser } from "./profile.actions";  @Injectable({ |

1. There is only one action at the moment, LoginUser, the name of our class. When we dispatch an action, it should therefore be an object of this type:

|  |
| --- |
| constructor(private store : Store<{loginUser : {loginStatus : boolean}}>) { }  login(){  this.store.dispatch(new LoginUser());  }  logout(){ |

1. It seems strange to have an entire Class for a single action, but this is how it is now.

This new object will be created upon dispatch, but we still have not changed the value of our Store property. For that we should change the Store a bit. Let’s add a constructor to the profile.action.ts file that accepts a property:

|  |
| --- |
| export class LoginUser implements Action {  readonly type = LOGIN\_USER;  constructor(public payload : boolean) {  }  } |

The constructor’s parameter must be public in order to pass a value into it. The name to the left of the colon can be anything, but payload is conventional.

1. Now in the service class, we can safely pass the value of *true* to our LoginUser() class, when it is called by dispatch():

|  |
| --- |
| constructor(private store : Store<{loginUser : {loginStatus : boolean}}>) { }  login(){  this.store.dispatch(new LoginUser(true));  } |

1. Check the login.component.ts file, if you had the loginUser() commented out, remove the comments. This is what the loginUser() method should look like:

|  |
| --- |
| constructor(private pService : ProfileService){}  loginUser(): void {  this.pService.login();  } |

You may now test the application by clicking on the login menu item at the top of the browser. Once the component loads, click the login button on that view. You should now see the logout button in the menu, indicating that login was successful.

# Part 6 Signals - Intro

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.

In this boot camp you are given starter files. The signalz-starter zipped file has to be un-zipped in a folder of your choice and renamed to just signalz. After that, remember to run the npm install command from that folder. Once that is done, run ng serve from the command line and go to localhost:4200. There you should see the only visible line as of right now: *…to be changed*. For the first part of this section on Signals, we will just use the boilerplate application provided. In the second part we use the application from earlier in this bootcamp.

1. In app.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.

*computed*.

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 7 Signals – Objects and Mutate

1. Let us work with an interface now. We will create an Employee interface in the app.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. Remove or comment the getFullName() method and the constructor. Begin to code the mutate() method of a Signal object inside of the ngOnInit() method:

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

Any component method can be used, I chose ngOnInit() for this example.

1. The mutate() 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 mutate part or all of that object. Here I will change just the lName part of the original object:

|  |
| --- |
| ngOnInit(): void {  this.Axle.mutate( o => o.lName = "Barr" );  } |

Here lower case ‘o’ represents the original object.

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. In order to get the value from this object, you first have to execute the object, then access it’s properties:

|  |
| --- |
| <div>  Hello {{ Axle().lName }}  </div> |

# Part 8 Signals – Converting the Profile Service to use Signals

In this section we try to convert the Profile Service we completed in Part 3 above to use Signals. Please use the code at the **end** of Part 3. At the end of Part 3 we saw the service being used by the parent, app component to check the status of the user. In that approach we created a new property and assigned it whatever the service responded with.

One of the child components, login, was used to access this service and call the login() method on the service. In this part we did not yet have a logout() method. That’s ok, we will use these files for our Signal journey.

Continue using the skills folder for this part of the boot camp, so close the signalz folder.

1. We start with the profile.service.ts file, import the signal package from @angular/core:

|  |
| --- |
| import { Injectable, signal } from '@angular/core';  import { BehaviorSubject } from "rxjs"; |

You could also remove the BehaviorSubject or comment out that import. Note, if you have not seen the Injectable class being used this way, it is simple. It is a replacement for constructor based class injection.

1. The local property isLoggedIn can now become our signal. The signal is of the Boolean type and at the same time pass in a default value of false:

|  |
| --- |
| export class ProfileService {  isLoggedIn = signal<boolean>(false);  constructor() { } |

1. Instead of firing the next() method, we update the signal:

|  |
| --- |
| login(){  this.isLoggedIn.update( prevValue => true );  }  logout(){  this.isLoggedIn.update( prevValue => false );  } |

At the moment we do not need loginStatus() or register(). The parameter prevValue can be anything. It holds whatever the current value of isLoggedIn was. Note, if you see any tutorials using mutate, just know that update is now used instead and mutate will be deprecated if not already.

1. The first place we might use this service is in the app component. All we need to do is point the existing userLoggedIn property to the injected ProfileService:

|  |
| --- |
| export class AppComponent implements OnInit{  title = 'profile';  userLoggedIn = inject(ProfileService);  //private loginSubject = new Subscription(); |

Also we don’t need the loginsSubject property either.

1. Remove all of the existing methods including the constructor, and replace all of them with just the logout() method for now:

|  |
| --- |
| userLoggedIn = inject(ProfileService);  //  logout(){  this.userLoggedIn.logout();  } |

Also, we will not be implementing the OnInit method, so clean up that part of the code.

1. Here is the entire app.component.ts file so far:

|  |
| --- |
| import { Component, inject } from '@angular/core';  import { ProfileService } from "./profile.service";  @Component({ …  export class AppComponent {  title = 'profile';  userLoggedIn = inject(ProfileService);  logout(){  this.userLoggedIn.logout();  }  } |

1. Next we handle the login component. It currently has a loginUser() function that is being called from the template. At the moment, the Profile service is being injected in the traditional way. We could leave it like this but there is more work involved. Lets transform it like we did for app.component.ts. Basically, inject the service, then use the property of that service which is of signal type:

|  |
| --- |
| export class LoginComponent {  userLoggedIn = inject(ProfileService);  loginUser(): void {  this.userLoggedIn.login();  } |

1. Switch the userLoggedIn binding in the app template, remove the status part if you see one:

|  |
| --- |
| routerLinkActive="active"  routerLink="/logout"  [hidden] = "userLoggedIn"  >  …  routerLinkActive="active"  routerLink="/login"  [hidden] = " ! userLoggedIn" |

1. At this point, you will see that it does not work. If you add a console.log() statement in the loginUser() method of the loginComponent class, you will see that the service signal is being changed. The problem is that the *view* is not being updated. We will handle that in the next section.

|  |
| --- |
| export class LoginComponent {  userLoggedIn = inject(ProfileService);  constructor(){ }  loginUser(): void {  console.log("before " + this.userLoggedIn.isLoggedIn());  this.userLoggedIn.login();  console.log("after " + this.userLoggedIn.isLoggedIn());  } |

# Part 9 Signals – Completing Profile Service to use Signals

In this section we complete the Profile Service using Signals and RxJS features. Although there are ways to force a view refresh, in the end using Observables seem to be a more elegant way to get this done for now. With this approach in Part 9, the Signal itself becomes an Observable. This triggers the Angular’s *change detection* mechanism.

1. We start with the profile.service.ts file, import the Observable and of packages from rxjs:

|  |
| --- |
| import { Injectable, signal } from '@angular/core';  import { Observable, of } from "rxjs"; |

For the new programmers, the of package is a name and it does exist.

1. Create a private Observable of the Boolean type above of the isLoggedIn property:

|  |
| --- |
| export class ProfileService {  private userStatus$ = new Observable<boolean>( );  isLoggedIn = signal<boolean>(false);  ~~constructor(private store : Store<{loginUser : {loginStatus : boolean}}>) { }~~  login(){ |

Also remove the constructor if you still have it.

1. Now the signal will be created using this Observable:

|  |
| --- |
| private userStatus$ = new Observable<boolean>( );  isLoggedIn = signal ( this.userStatus$ );  login(){ |

1. With this different approach, the Signal itself is now an Observable. Remember isLoggedIn is our Signal, it is writable. This means that we can use the set() operator on it and fire a Boolean value:

|  |
| --- |
| constructor() { }  login(){  this.userLoggedIn.set ( of ( true ) );  }  logout(){  this.userLoggedIn.set(of(false));  } |

1. Moving on to the app.component.ts file, remove the ngOnInit() method and its imports. Also remove the implements part of the class creation. Do this only if you have this part of the code.
2. Make sure that the inject class is imported and then use it to inject the ProfileService. Also, create a local property of status which will be of the Boolean type:

|  |
| --- |
| export class AppComponent {  title = 'profile';  pService = inject(ProfileService);  status : boolean = false; |

The constructor will be used in this example, see below.

1. If you do not have a constructor, create one now and immediately call the logout() method of the service we injected in #5 above:

|  |
| --- |
| status : boolean = false;  constructor(){  this.pService.logout();  }  logout(){ |

I could not find a better way to make sure that at class creation time, the user is logged out.

1. The final piece of this file is of course the logout process. Remember this component is the one used to *log out* the user:

|  |
| --- |
| constructor(){  this.pService.logout();  }  logout(){  this.pService.logout();  } |

1. Now in the login.component.ts file, the final file so far in this change, rename userLoggedIn to just pService:

|  |
| --- |
| export class LoginComponent {  pService = inject(ProfileService);  ~~constructor(){}~~  loginUser(): void {  this.pService.login();  } |

There is no need for a constructor in this file.

1. Up to this point we have not touched the template. With this new architecture we have to change the template. We are not now dealing with a true/false property. We have to handle an Observable. Luckily Angular has given us the async pipe for just this purpose:

|  |
| --- |
| <li class="nav-item">  <a  class="nav-link"  routerLinkActive="active"  routerLink="/logout"  [hidden] = "!(pService.isLoggedIn() | async)"  >  logout  </a>  </li>  <li class="nav-item">  <a  class="nav-link"  routerLinkActive="active"  routerLink="/login"  [hidden] = "(pService.isLoggedIn() | async)"  >  login |

There is no need for a constructor in this file.

1. Also, we could now hookup the logout menu link to a function on the .ts file:

|  |
| --- |
| <a  class="nav-link"  routerLinkActive="active"  routerLink="/home"  [hidden] = "!(pService.isLoggedIn() | async)"  (click)="logout()"  > |

Notice that after the user logs out, they are immediately redirected to the home view

1. The app should now work as designed.

# Appendix A Logging Out with Actions

If you want to see how this code works, you must first get the code from the end of Part 5. This code is a continuation of using Store to keep state. It has nothing to do with Signals.

So, continuing from Part 5, Now that we can dispatch an action and turn the initial value to *true* from *false*, we need to turn it back to *false* if needed. As you might have already guessed, this can be done from the service where we already have the infrastructure to do this.

1. Import the LogoutUser from the actions file at the top of the service file:

|  |
| --- |
| import { Store } from "@ngrx/store";  import { Observable } from "rxjs";  import { LoginUser, LogoutUser } from "./profile.actions";  @Injectable({ |

1. Complete the logout() function in the profile.service.ts file and perform the opposite of the login() function:

|  |
| --- |
| this.store.dispatch(new LoginUser(true));  }  logout(){  this.store.dispatch(new LoginUser(true));  }  loginStatus(){ |

It may seem strange to pass the value of *true* here, but remember the reducer is evaluating which code to execute based on a switch/case structure. If we pass *false* here, the default case will prevail.

1. The next question then is where to call this logout service method from. We could mirror the login process and create a new component. However I think that it is more efficient to do this from the parent component, app.component.ts file:

|  |
| --- |
| this.userLoggedIn = status;  console.log(status);  });  }  logout(){ } |

1. Since we already have access to our Profile service, just call the logout() method on that service:

|  |
| --- |
| });  }  logout(){  this.pService.logout();  } |

1. Now, just calling the logout() method will not affect the template view. We have to now extract this new value and pass it on to the local property userLoggedIn:

|  |
| --- |
| logout(){  this.pService.logout();  this.pService.loginStatus()  .subscribe(status => {  this.userLoggedIn = status;  }); |

1. So far, we have a service that contains a function that calls our Store. But what calls the function inside of the service? For that we can use event binding. Since it is the user who initializes this, we hook into the click event of the menu link on the template:

|  |
| --- |
| <li class="nav-item">  <a  class="nav-link"  routerLinkActive="active"  routerLink="/logout"  [hidden] = "!userLoggedIn.loginStatus"  (click)="logout()"  >  logout  </a> |

1. Following the chain, the service then dispatches an action to logout the user:

|  |
| --- |
| logout(){  this.store.dispatch(new LogoutUser(true));  } |

1. In the reducer, you might be tempted to do something like this:

|  |
| --- |
| export function profileReducer(state = initialState, action : Action){  if(action.type == LOGIN\_USER){  return {  loginStatus : true  }  } else {  return { loginStatus : false }  }  ~~if(action.type == LOGOUT\_USER){~~  ~~return {~~  ~~loginStatus : false~~  ~~}~~  ~~} else {~~  ~~return { loginStatus : false }~~  ~~}~~ |

This will NOT work!

1. We should instead create a new action. So in the profile.actions.ts file, add a new class to handle logging out:

|  |
| --- |
| import { Action } from "@ngrx/store";  export const LOGIN\_USER = 'LOGIN\_USER';  export const LOGOUT\_USER = 'LOGOUT\_USER';  export class LoginUser implements Action {  readonly type = LOGIN\_USER;  constructor(public payload : boolean) { }  }  export class LogoutUser implements Action {  readonly type = LOGOUT\_USER;  constructor(public payload : boolean) { }  } |

Yes, it is possible to have multiple classes in one ts file.

1. Then in the reducer, import this new property from the actions file:

|  |
| --- |
| import { Action } from "@ngrx/store";  import { LOGIN\_USER, LOGOUT\_USER } from "./profile.actions"; |

1. In the profileReducer() function, begin a switch() method based on the action.type that comes with the NgRx library:

|  |
| --- |
| export function profileReducer(state = initialState, action : Action){  switch ( action.type ) {  } |

1. The first case can be the login case:

|  |
| --- |
| switch ( action.type ) {  case LOGIN\_USER:  return { loginStatus : true };  break; |

1. The second can be logout:

|  |
| --- |
| case LOGIN\_USER:  return { loginStatus : true };  break;  case LOGOUT\_USER:  return { loginStatus : false };  break; |

1. This type of switch architecture requires a default in order to function properly:

|  |
| --- |
| break;  default:  return { loginStatus : false };  break;  } |

The default case can return true or false depending on the rest of your application.

1. Here is the entire function:

|  |
| --- |
| export function profileReducer(state = initialState, action : Action){  switch ( action.type ) {  case LOGIN\_USER:  return { loginStatus : true };  break;  case LOGOUT\_USER:  return { loginStatus : false };  break;  default:  return { loginStatus : false };  break;  }  } |

1. One more thing to consider here. In the template, if we now execute this logout procedure, we can re-direct the user to wherever we want. For this simple example I will re-direct the user back to the home page:

|  |
| --- |
| <a  class="nav-link"  routerLinkActive="active"  routerLink="/home"  [hidden] = "!userLoggedIn.loginStatus"  (click)="logout()"  > |

# Appendix B – Signals

A **signal** is a wrapper around some value that broadcasts to users of that value, when it changes. Signals can contain any value that JavaScript understands. So the same concept as JSON.

You access a signal's value by invoking its getter function. Angular will track the most recent value and provide that back to you in a return.

Signals may be either *writable* or *read-only*. Signals can also be computed from other signal values. These computed signals are read-only, cached and can be considered a form of lazy evaluation.

Signals can have an **effect.** There is an effect function that can be implemented whenever one or more signal values change. This is used mostly for logging purposes.

I mentioned that signals work nicely with observables. Well Angular's @angular/core/rxjs-interop package contains several utilities to integrate [Angular Signals](https://angular.dev/guide/signals) with RxJS Observables seamlessly.

For example there is the toSignal() function that will create a signal to watch an observable. In the background a subscription to that observable is formed and gets destroyed when the component is destroyed.

On the other hand there is the [toObservable](https://angular.dev/api/core/rxjs-interop/toObservable)() utility function that will create an Observable to track the value of a signal. The signal's value is watched using an [effect](https://angular.dev/api/core/effect). That effect is used to emits a value back to the Observable whenever the signal’s value changes.