Seneca
SCHOOL OF INFORMATION AND COMMUNICATIONS TECHNOLOGY

BTI425

Web Programming for Apps and Services

Week 7 in-class hands-on exercise

Goal or objective: To an existing app, add the ability to work with a web service

Overview

Last week, you completed a hands-on exercise. This week, we build upon that work. Fetch your work from last week, or fetch the solution in the repo folder.

Today, the goal is to add the ability to work with a web service (your Assignment 1 web service).

Fetch the app

Fetch your work from last week, or fetch the solution in the repo's *Templates and solutions* folder.

Open it for editing. Then, start/run it:

```
ng serve --open
```

Leave it running, and check it as you make changes. If the display goes blank/white, there's an error, so show the browser dev tools, and its console.

Make some components

In this section, you will generate two new components:

1. Person list (for "get all")
2. Person detail (for "get one")

Remember, creating a component is best done as follows. First, make sure you're in the project folder. Then create three components.

```
ng g c xxx-name-of-new-component-xxx --flat -S

# If your ng version is 9 or later, omit -S...
ng g c xxx-name-of-new-component-xxx --flat
```

> *Remember the tip from the notes:*
> *If you're creating a component that should have a multi-word name, use Pascal Case. For example:*
> `ng g c AboutMe --flat -S`

Configure the router module, by adding route objects for the three new components. Regarding the existing nav menu, add choices for the "get all" use case. Test your work.

Overview, work with a web service

Here's an overview of the work we will do in this section:

1. Add support for making HTTP requests
2. Write an interface
3. Write a service
4. Write the component

We write an TypeScript "interface" to describe the shape of the data from the web service.

We write an Angular "service" to interact with the web service.

We write the component code, which fetches from the Angular service, and renders the results in the UI.

Add HTTP request bits

We must add, to the app, support for making HTTP requests. While we can use the Fetch API, the Angular team recommends that we use HttpClient.

Open `app.module.ts` for editing. Add an import statement, and edit the decorator:

```
// Add this after the BrowserModule import
import { HttpClientModule } from '@angular/common/http';

// In the decorator's "imports" array, add it
  imports: [
    BrowserModule,
    HttpClientModule,
    // etc.
  ],
```

Write an interface

In an Angular app, we MUST know the shape (i.e. the *type*) of the data we're working with, including data from a web service.

To implement this requirement, TypeScript offers both a class and an interface. How to decide?

- Type-checking only? Use an interface
- Type-checking AND implementation/behaviour? Use a class
- Must create an instance? Use a class

> *Source: Post by Todd Motto*

In terminal, create a source code file named `data-classes.ts` (in the `src/app` folder obviously). Alternatively, use the generator:

```
ng g class DataClasses --flat
```

We will use this source code file to hold interfaces and classes for ALL data model classes in our app that are used in multiple components.

Open and study the shape of the responses from your web service. Open the new file for editing. Carefully write the interface code that describes the shape of one “person” object. It will look something like this:

```
export interface Person {
  _id?: string;
  firstName: string;
  lastName: string;
  // other properties go here
}
```

> *Note: The question mark that follows the property name indicates that the value is optional. In other words, it can be null.*

Data classes can obviously have properties. The syntax form above declares the name and type of a property. It is possible to assign an initial value to a property. In addition, data classes can include a constructor. For example:

```
export class UserAccount {

  constructor() {
    let now = new Date();
    this.dateCreated = now.toISOString();
  }

  _id?: string;
  username: string;
  password: string = 'abc123';
  dateCreated: string;
}
```

> *Reminder - The TypeScript Class docs are here.*

Write a service

Use the Angular CLI to generate a service. For example:

```
ng g s DataManager --flat -S
```

Open it for editing. Add these to the existing import statement(s):

```
import { HttpClient } from "@angular/common/http";
import { Person } from "./data-classes";
import { Observable } from 'rxjs';
```

Update the constructor signature, by adding an argument:

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

Add a new string property to the class, to hold the URL:

```
private url: string = 'https://your-web-api.herokuapp.com/api/persons
```

Add a new “get all” method to the class:

```
personsGetAll(): Observable<Person[]> {
  return this.http.get<Person[]>(this.url);
}
```

What is this code doing?

- It does not need arguments, because it does not have passed-in data
- Its return type is an “Observable” of type “Person[]” (i.e. a collection of Person objects)

What’s an “Observable”? For us, and for now, and our web service interaction, you can think of it as a kind of *Promise* that better serves the needs of Angular apps. This note has more information.

> *FYI - There is an active proposal to add Observable to the next version of the JavaScript language.*

Write component - class code

Open the person list component - class code - for editing. Add these to the existing import statement(s):

```
import { Person } from "./data-classes";
import { DataManagerService } from './data-manager.service';
```

Declare a property that will hold the collection of Person objects. This propery will be accessible in the (HTML) template code.

```
persons: Person[];
```

Update the constructor signature, by adding an argument:

```
constructor(private m: DataManagerService) {
```

In React, we added code to a `componentDidMount()` method, which fetched data from a web service. Here, we do someting similar.

Add code to the `ngOnInit()` method:

```
this.m.personsGetAll().subscribe(p => this.persons = p);
```

What is this code doing?

- It is calling the `subscribe()` method of the `personsGetAll()` function's return value
- This actually executes the "get" (fetch)
- The (callback) function that we pass into the `subscribe()` method tells it how to handle the results (which is a collection of Person objects)

> *If you want to see a bit of feedback in the console, replace that single-line statement with the following multi-line statement:*
>
> ```
> this.m.personsGetAll().subscribe(p => {
>    console.log(p);
>    this.persons = p;
> });
> ```

Write component - template code

Open the person list component - template code - for editing. Add a table structure with enough columns to make you happy.

Then, dereference the contents of the "persons" collection (in the class code) with the following `*ngFor` directive, which makes a new `tr` element for each "person" object:

```
<tr *ngFor='let p of persons'>
  <td>{{ p.firstName + ' ' + p.lastName }}</td>
  <td>{{ p.birthDate }}</td>
  <td>{{ p.creditScore }}</td>
```

```
  <td>{{ p.rating }}</td>
</tr>
```

Test. It should work. Yay!

Can we do something about the appearance of the date? Yes, and an easy fix is really easy. Angular has a feature built in, pipes, which can transform or format values. For us, the date pipe is the one we want. Edit the cell:

```
<td>{{ p.birthDate | date }}</td>
```

Implement the “get one” use case

Now that we have learned the essentials, let’s implement the “get one” use case. Parts of it will be familiar to the work done above, and also with the work you did in your React app.

Setup tasks

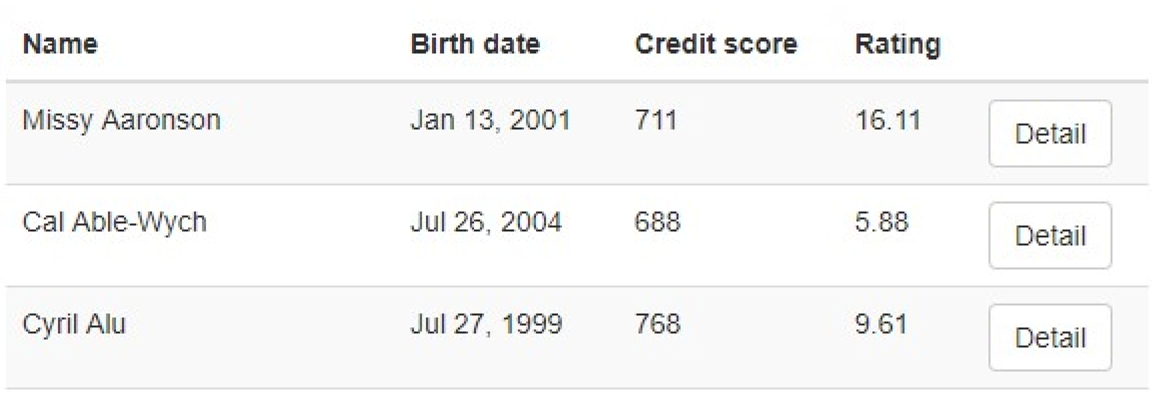
Eearlier, when you configured the router module, it’s likely that you configured a route for the “get one” (detail) component, maybe using a path value that looked similar to what you had done in your React app:

```
{ path: 'persons/:id', component: PersonDetailComponent},
```

Now, edit the “get all” template code, and add another column in the table to hold a details link (which is styled to look like a button):

```
<td><a class="btn btn-default" routerLink="/persons/{{ p._id }}">Deta
```

Test, and it should look something like the following. When you hover over a button, it should render a link that matches a route object:



Name	Birth date	Credit score	Rating	
Missy Aaronson	Jan 13, 2001	711	16.11	Detail
Cal Able-Wych	Jul 26, 2004	688	5.88	Detail
Cyril Alu	Jul 27, 1999	768	9.61	Detail

Work with the "get one" class code, initial

Our initial goal is simply to read and display the identifier that's part of the URL. Open the "get one" class code file for editing. We'll do these tasks:

1. Add the ability to read the URL
2. Extract the data we need from the URL
3. Display it

We must import a module to get the ability to read the URL. Add this import statement:

```
import { ActivatedRoute } from "@angular/router";
```

Update the constructor signature, by adding an argument:

```
constructor(private route: ActivatedRoute) { }
```

Add a string property to the class, so that we can save/store the identifier data we will get from the URL:

```
id: string;
```

Next, in the `ngOnInit()` method, extract the identifier from the URL, and save it:

```
this.id = this.route.snapshot.paramMap.get("id");
```

Finally, let's display it. You can add a simple `console.log()` statement, or a better way is to open the template code for editing, and adding a new element:

```
<p>The identifier from the URL is {{ id }}</p>
```

Now, when you click/tap a button on the list of items, it will navigate to the “get one” component, and display the identifier of the item.

Refine the functionality

Now, our goal is to use the value of the identifier, and send another request to the web service.

Open the data model manager service for editing. We want to add a method that will do a “get one” request:

```
personsGetById(id: string): Observable<Person> {
  return this.http.get<Person>(`${this.url}/${id}`);
}
```

What is this code doing?

- It has an argument for the item identifier
- Its return type is an “Observable” of type “Person” (i.e. one single Person object)

Now, open the “get one” class code for editing. Add these to the existing import statements:

```
import { Person } from "./data-classes";
import { DataManagerService } from './data-manager.service';
```

Declare a property that will hold the Person object.

```
person: Person;
```

Update the constructor signature, by adding an argument:

```
constructor(private route: ActivatedRoute, private m: DataManagerSer
```

Add the code to the `ngOnInit()` method:

```
this.m.personsGetById(this.id).subscribe(p => this.person = p);
```

What is this code doing?

- It is calling the `subscribe()` method of the `personsGetById()` function's return value
- This actually executes the "get" (fetch)
- The (callback) function that we pass into the `subscribe()` method tells it how to handle the results (which is a single Person object)

> *If you want to see a bit of feedback in the console, replace that single-line statement with the following multi-line statement:*
>
> ```
> this.m.personsGetById(this.id).subscribe(p => {
>    console.log(p);
>    this.person = p;
> });
> ```

Now, you're ready to edit the template code, to show or display the item's data. Open the template code for editing. Add suitable markup to render the values of the person object (e.g. a description list is often a good choice).

> *Is your Person coded as an interface?*
> *Does your console show error messages like this?*
> `ERROR TypeError: Cannot read property 'blah' of undefined`
> *The reason is that the person property is declared but does not have data when the component first loads, so the error messages appear. The person property gets its data after a successful call to the service.*
> *We can fix this by making Person a class, and then creating a new empty instance in the component's constructor.*