WEB422 Assignment 5

Submission Deadline:

Friday, November 15, 2019 @ 11:59pm

Assessment Weight:

9% of your final course Grade

Objective:

To work with an existing static HTML documents & refactor them into Angular components with Routing. Additionally, introduce Services in Angular & display information from our Teams API.

Specification:

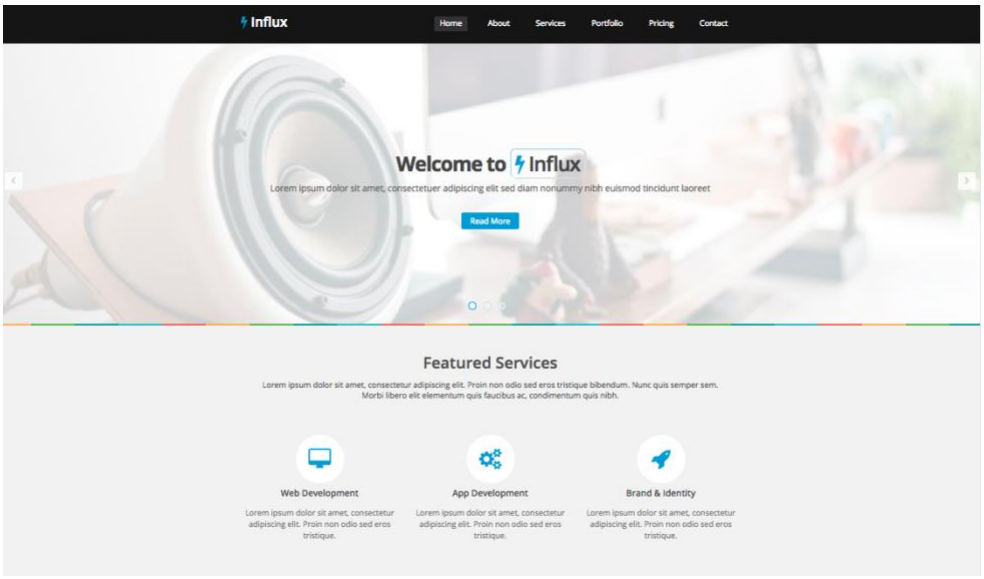
For this application, we will be using a premium html Bootstrap template (influx) and incorporating it into an application with routing enabled. Once this is complete, we will add services to load information from our Teams API into our components.

Getting Started:

To begin this assignment, download the template - https://scs.senecac.on.ca/~patrick.crawford/shared/winter-2019/web422/A5/influx.zip

IMPORTANT NOTE: - this template may only be used within the context of the WEB422 course for Winter 2019 at Seneca College

Open the index.html file from the "influx" template - you should see:



Our job is to use the design provided in this template to kick-start a new (routing enabled) angular application.

Part 1 - Creating the App / Adding Views & Routing

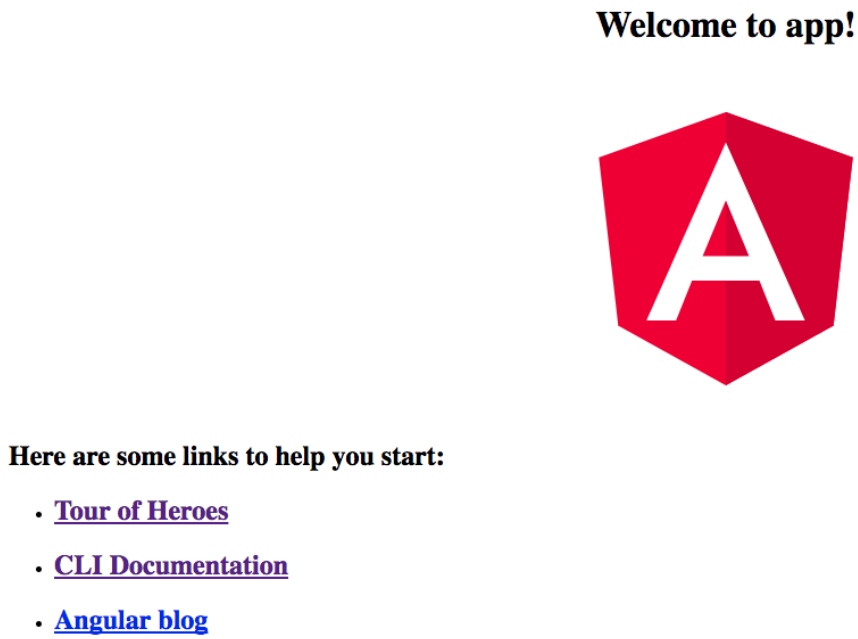
Step 1: Creating a new Angular App

Now that we have the premium template downloaded, we can create a fresh (routing enabled) angular application (Refer to the Week 7 notes under: **Quick Review - Creating an Angular Project from Scratch**).

Once the application is created, we need to incorporate the content from the "influx" template into our new application:

Step 2: Updating our new app with "Influx" template content

If we run our new Angular application, we should see something that looks like the following:



What we need to do, is ensure that the <body> content from the Influx "index.html" file is carried over to our "index.html" file within our Angular application. However, first we need to bring over the resources from the influx template:

- In Visual Studio code, right-click on the "assets" folder within the "src" folder and choose "Reveal in Finder" (OSX) or "Reveal in Explorer" (Windows).
- Open the "assets" folder - we will be copying content from our Influx template into this folder, ie:
    - **css** folder - (from the Influx template)
    - **fonts** folder - (from the Influx template)
    - **images** folder - (from the Influx template)
    - **js** folder - (from the Influx template)

- Once the files are copied, you can **remove main.js** from the **js** folder - we will not be using it.
- Lastly, refresh the file listing in Visual Studio Code - you should see the new files there

The next step is to copy the **content of the <body> element** (but not the <body></body> tags) from the "index.html" file from the "Influx" template, into the content of our new Angular app's **app.component.html** file, effectively replacing the default (above).

However, you will notice that there's some links to some JavaScript files (the ones recently added to our "assets" folder. They don't belong in the component - they really belong in the "index.html" file:

- Move the first four (4) <script></script> elements that you just pasted into app.component.html (there should be 4 lines at the end of the file - we will **not** be including main.js) into the "index.html" file at the end of the <body> element (beneath the <app-root></app-root> element)
- Modify the "src" attributes to accommodate the new location of the files, ie: change src="js/jquery.js" to src="assets/js/query.js" for all 4 of the script tags

Additionally, you will notice that inside the "index.html" file of our Influx template, we have references to the CSS files - these are currently missing from our Angular app:

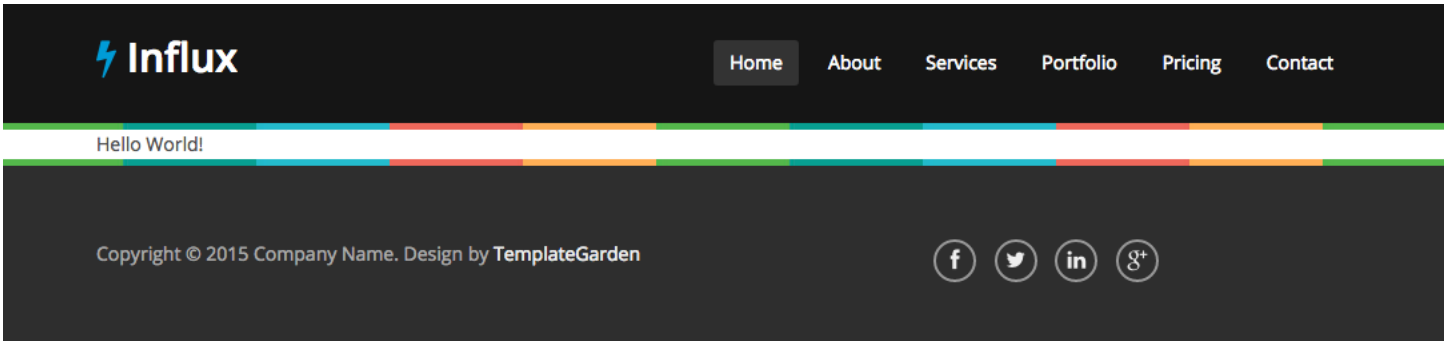
- Copy the 5 <link href="…css" rel="stylesheet"> elements at the top of the "index.html" file (from the Influx template) to the top of the "index.html" file in our Angular app - just **below** the line: <meta name="viewport" content="width=device-width, initial-scale=1">
- Modify the 5 <link> elements to include the new "assets" path, ie: <link href="css/bootstrap.min.css" rel="stylesheet"> should read <link href="assets/css/bootstrap.min.css" rel="stylesheet">
- Save all the changes

Finally, open the app.component.html file again and remove all the content **except the following:**

- <header>...</header>
-
-
- …

Next, between the two "color-border" elements, you can place a new … element for testing. Feel free to add some content inside the div: "Hello World".

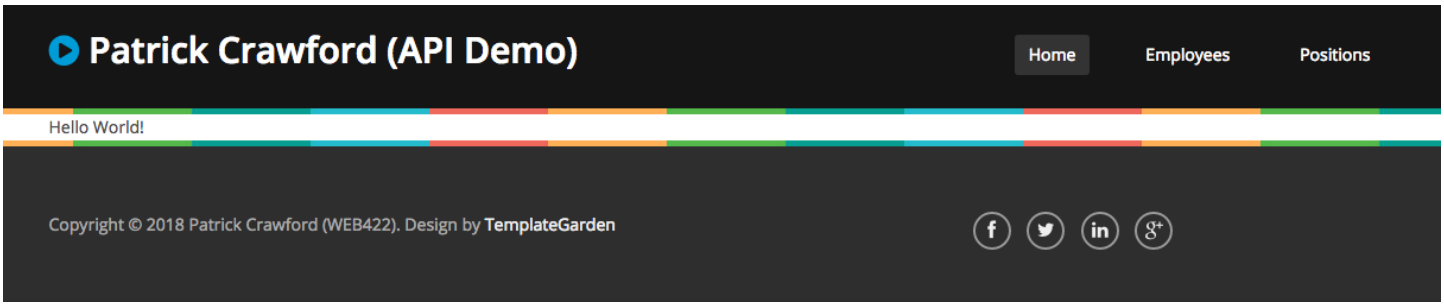
With all the changes saved, you should now be able to run the app (webpack should automatically compile it) and you should see the following (if you used "Hello World!" as placeholder text within your … element)



This isn't especially personal and we won't be using all the above routes, so we need to make a few edits to our app.component.html file:

- Modify the … to include your name followed by the text "(API Demo)" *instead* of the text "Influx" and choose a different font-awesome icon. Choosing a different font-awesome icon will involve changing the class "fa-bolt" into something from [this list of font awesome 4.0.3 icons](), for example "fa-play-circle"
- Change the footer in the copyright, so that instead of reading "2015 Company Name.", it will instead read "**2019 [student name] (WEB422)**", where [student name] is your full name.
- Remove the all the links **except for "Home"** in the <header>…</header> element (ie "About", "Services", "Portfolio", etc.) and replace them with "Employees" and "Positions" ("Employees" and "Positions" can link to "#" for the time being - we will be adding routing later).

With all of the changes in place, your app should look something like this



Step 3: Creating Components - Part I:

Since our entire "home" page (including the navbar, content & footer) exists within the <app-root></app-root> component, we need to start to think about dividing the page up into sections (Components):

Using the Angular CLI, create the following new components (Recall: the CLI automatically appends the text "Component" to your component name):

- NavComponent (selector: app-nav)
- ContentComponent (selector: app-content)
- FooterComponent (selector: app-footer)

Next, we need to update our **app.component.html** file to use the above 3 components, ie (back up / comment out the contents of this file first, and change it to the following):

```
<app-nav></app-nav>
<app-content></app-content>
<app-footer></app-footer>
```

This will break our layout and if you look in the browser after running "ng serve" and saving your changes, you should see 3 lines:

```
nav works!
content works!
footer works!
```

To get our content back, we need to divide our **old app.component.html** contents into the above 3 components, ie:

NavComponent:

ContentComponent:

This component simply shows the … element that we added for testing

FooterComponent:

Step 4: Creating Components - Part II:

You will have noticed that we still have left out all our major "views", ie "Home", "Employees" and "Positions".

The next part of this assignment requires you to create high-level components for each of these views, ie:

- HomeComponent (selector: app-home)
- EmployeesComponent (selector: app-employees)
- PositionsComponent (selector: app-positions)

Once You have created each of the above components, edit their respective .html template files and replace the … element with the following:

```
<div class="container">
```

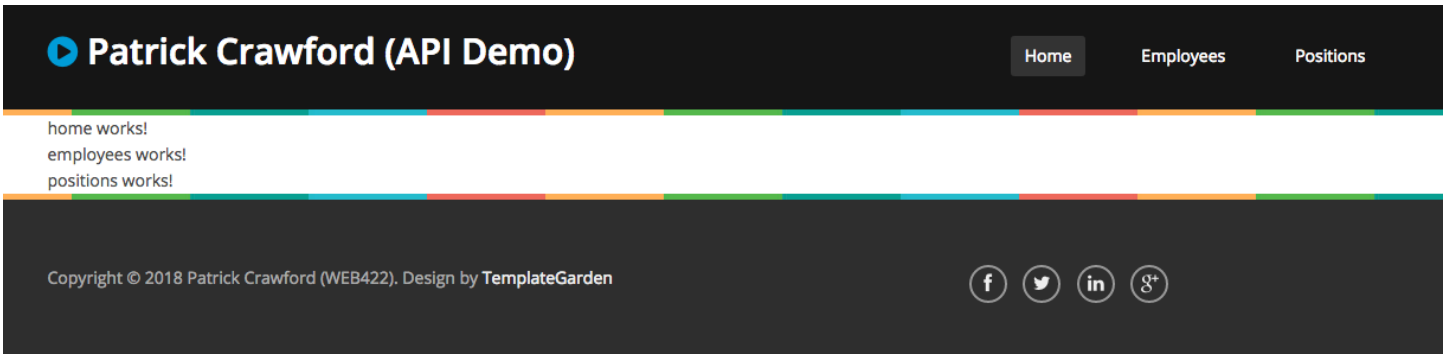
```
        <div class="row">
            <div class="col-md-12">
                ...
            </div>
        </div>
</div>
```

This will place each of these components on their own single column "container".

Now that we have content for each of these components, place them inside the ContentComponent template, ie:

```
<app-home></app-home>
<app-employees></app-employees>
<app-positions></app-positions>
```

If we view our app now, we should see:



Step 5: Adding Routing:

Once all our main view components are complete (ie: HomeComponent, EmployeesComponent, PositionsComponent), we can "wire up" our application to respect predefined routes within our application.

Before we begin however, we should make a 404 "Not Found" component, to use if a route is not matched. This can contain any content you wish, but try to design it to fit within the theme of the rest of the site. The only requirements are that:

- It must be nested within a single column container, ie: …
- It must show the text "Not Found" in the template
- The component must be called PageNotFoundComponent (selector: app-page-not-found)

Updating app-routing.module.ts

We should now have all the components we need to create a functioning site, we just have to do a little bit of maintenance work to ensure that all of our routes work correctly. The first task is to update the "app-routing.module.ts" file:

- Import all 4 of your main "view" components, ie: HomeComponent, EmployeesComponent, PositionsComponent and PageNotfoundComponent at the top of the file, beneath the existing import statements using the syntax (for example):

  import { HomeComponent } from "./home/home.component";

- Add all of your routes to the Routes = [ ] array according to the following rules:

Path	**Component / Redirect**
'home'	HomeComponent
'employees'	EmployeesComponent
'positions'	PositionsComponent
''	Redirects to "/home", pathMatch: 'full'
'**'	PageNotFoundComponent

Adding a "Router Outlet" Component

With all the routes defined, the last thing that we need to do to "Activate" the routes, is to remove the 3 components our **content.component.html** and replace them with a single "Routing" component:

<router-outlet></router-outlet>

Updating the Navigation Bar / Internal links ("Contact us", etc)

If we wish to make use of the router correctly, we must update all the anchor (<a>) links in the app use Angular's "routerLink" directive instead of the "href" attribute (Hint: Do not forget the navbar-brand), ie:

<a **routerLink="/home"**>Home

Highlighting Current Navigation Menu Items

An important usability feature of most websites, is to **highlight** the navigation bar to match the current "route" the user is currently on. Fortunately, using Angular's Component router, we have a very simple directive we can use: **routerLinkActive**.

Using this directive, we can conditionally add/remove the "active" class to the correct <li> element within the navigation menu by simply adding the directive: **routerLinkActive="active"**, ie:

```
<li routerLinkActive="active"><a routerLink="/home">Home</a></li>
<li routerLinkActive="active"><a routerLink="/employees">Employees</a></li>
<li routerLinkActive="active"><a routerLink="/positions">Positions</a></li>
```

Part 2 - Updating the Views / Adding Services & Rendering Data

Step 1: Updating the Home view:

Recall, our original template had some very cool features, including a nice slider on the home page. To restore the look and feel, open the original index.html file from the influx template and copy the entire block of html starting from (including):

```
<section id="main-slider" class="no-margin">
```

and ending with

```
</section>
<!--/#feature-->
```

This should be 96 lines of html code in total. Use this code to replace the existing … element that we currently have

If you refresh the page now, you should see something that very closely resembles the original index.html page from our influx template!

The next step is to make this page our own, by making the following changes:

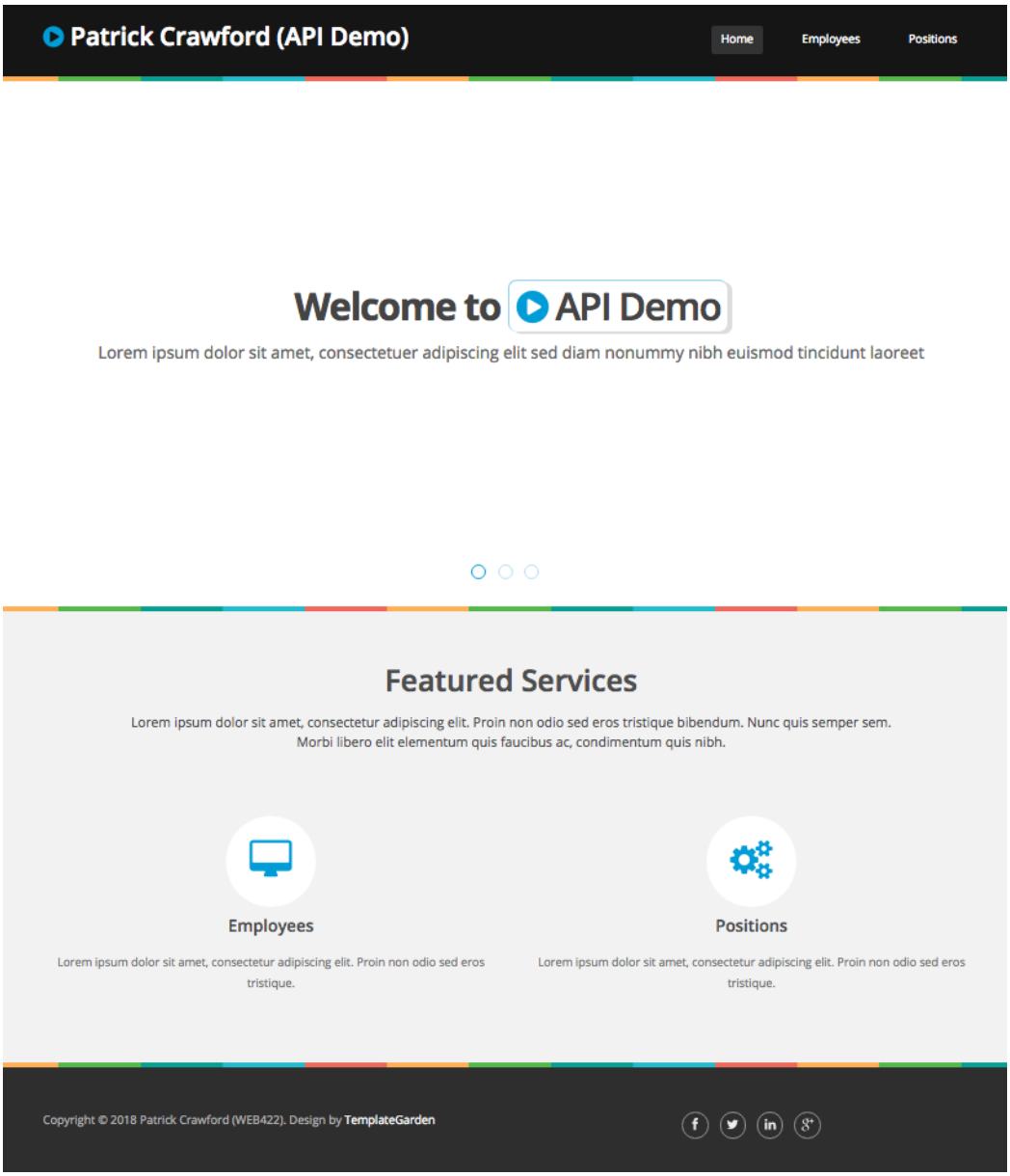
- Add the following css to **home.component.css** to ensure the correct spacing between sections:

```
section {
    padding: 70px 0;
}

.no-margin {
    margin: 0;
    padding: 0;
}
```

- Change the "Welcome to" text to match your navbar-brand, ie: "API Demo" instead of "Influx" (Note: be sure to use the matching font-awesome icon as well)
- Remove the "Read More" links (we do not have an "about-us" route)
- In the "feature" section, remove the last column ("col-md-4") and change the other features (ie: "Web Development" and "App Development" to read "Employees" and "Positions" and change their column width *from* "col-md-4" to "col-md-6" (since there are only two of them now, instead of three).

Once all the changes are made, your "home" route should resemble the following:



Step 2: Creating Classes & Services to Fetch Data:

Updating the "Home" page was easy, we just had to tweak some html / css. The Employees & Positions are a little more difficult, however. Here, we will need to define classes & services to connect to our "Teams API" on Heroku and update our components with the resultant data.

To get started, add the following services and classes (to match our incoming data). Once the files are created, place their .ts files in a new folder called **data** within the **src/app** directory (**src/app/data**).

NOTE: For a refresher on Creating / Working with Services in Angular, see the example from the course notes here: https://sictweb.github.io/web422/notes/angular-services-example

Classes:

Position (position.ts)

Property	Type
_id	string
PositionName	string
PositionDescription	string
PositionBaseSalary	number
__v	number

Employee (employee.ts)

Note: For this class definition, we need to import the "Position" class from './position' so that our typed class is valid.

Property	Type
_id	string
FirstName	string
LastName	string
AddressStreet	string
AddressState	string
AddressCity	string
AddressZip	string
PhoneNum	string
Extension	number
Position	**Position**
HireDate	string
SalaryBonus	number
__v	number

Services:

The following 2 services need to be added to our application. Recall: you can generate a service using the Angular CLI with the command "ng g s *serviceName* --spec false". They must also be made available on the root (app) component level, so **do not forget** to **move them into the "data" folder** add them to the **providers** array of **@ngModule** in **app.module.ts.**

PositionService (position.service.ts)

This service needs to expose the following method:

- **getPositions()**

  This method must make a "**get**" request (using the HTTPClient module) to your Teams API running on heroku with the path: "/positions". It will return an Observable of type Position[], ie: Observable<Position[]>

EmployeeService (employee.service.ts)

This service needs to expose the following method:

- **getEmployees()**

  This method must make a "**get**" request (using the HTTPClient module) to your Teams API running on heroku with the path: "/employees". It will return an Observable of type Employee[], ie: Observable<Employee[]>

Step 3: Updating Our Components to Use the Services / Add New Methods

The next step is to use the services to fetch data from our Teams API and store it in the correct component. This will involve adding the following for each of the main "data" components"

HINT: Do not for get to **import** the **HTTPClientModule** in **app.module.ts** and include it in the **imports** array of @NgModule:

EmployeesComponent (employees.component.ts)

Properties:

- employees (type Employee[])
- getEmployeesSub (type any)
- loadingError (type boolean, default value: false)

Methods:

- constructor()
  In this method, we must inject the following Service:
  - EmployeeService from employee.service
- ngOnInit()
  In this method, we will use Injected services / modules to perform the following task:
  - populate the "employees" property using the EmployeeService service (Note: a reference to the subscription should be stored using "getEmployeesSub" so that it can be disposed of later)
  - If an error was encountered, set the value of this.loadingError to true (we can use this in the view to indicate that there was an error by showing an error message)
- ngOnDestroy()
  In this method we call the "unsubscribe()" methods on any saved subscriptions within the component (ie: getEmployeesSub) - Note: we must make sure they are not "undefined" before we call "unsubscribe()"

PositionsComponent (positions.component.ts)

Properties:

- positions (type Position[])
- getPositionsSub (type any)
- loadingError (type boolean, default value: false)

Methods:

- constructor()
  In this method, we must inject the following Services:
    - PositionService from position.service
- ngOnInit()
  In this method, we will use Injected services / modules to perform the following task:
    - populate the "positions" property using the PositionService service (Note: a reference to the subscription should be stored using "getPositionsSub" so that it can be disposed of later)
    - If an error was encountered, set the value of this.loadingError to true (we can use this in the view to indicate that there was an error by showing an error message)
- ngOnDestroy()
  In this method we call the "unsubscribe()" methods on any saved subscriptions within the component (ie: getPositionsSub) - Note: we must make sure they are not "undefined" before we call "unsubscribe()"

Step 4: Quick Sanity Check

Now that our components have access to their respective services and everything ***should*** be working properly, why don't we do a quick sanity check?

- In your positions.component.html file, next to "position works!", add the line: **{{positions | json}}**
- In your employees.component.html file, next to "employees works!", add the line: **{{employees | json}}**

If you have written / used your services correctly in the components per the specification above, you should see all the data loaded into the correct view! The next (final) piece is rendering it in a table:

Step 5: Rendering the Employee / Position Data

To give our Employees / Positions views the same look and feel as the rest of the site, let's recycle some code from the "influx" layout, specifically:

```
<div class="center">
  <h2>Title</h2>
  <p class="lead">Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin non odio sed eros tristique bibendum.</p>
</div>
```

Place this code **above** the … in both the employees.component.html file **and** the positions.component.html file, ensuring to update the "Title" to read "Employees" or "Positions" depending on the template.

If we look at our views in the browser now, we'll see that they do indeed match the theme, however the positioning is off (too close to the header). To remedy this (and add some spacing for our upcoming table), add the following lines of CSS to both the employees.component.css file **and** the positions.component.css file:

```
.center{ margin-top:40px; }
.table-responsive{margin-bottom:60px;}
```

Next, we must update each file (employees.component.html & positions.component.html) to render our data in a table, instead of displaying all the data as JSON in the view. To achieve this we will instead use the following table structure to display formatted data (within our … element):

```
<div class="table-responsive">
  <table class="table table-condensed table-hover">
    <thead>
      ...
    </thead>
    <tbody>
      ...
    </tbody>
  </table>
</div>
```

For the specific columns in each table (Employees & Positions), see below:

"Employees" Table

- Full Name: This is the employees First Name & Last Name
- Address: This is the employees Street, State, City & Zip Address Values, formatted as: "Street. State, City. Zip"
- Phone Number: This is the employees Phone Number & Extension Values, formatted as "Phone Number ext: Extension"
- Hire Date: This is the employees hire date, formatted using the **date:'longDate'** DatePipe (https://angular.io/api/common/DatePipe)

"Positions" Table

- Position Title: This is simply the title of the position, ie: "Front End Developer"
- Position Description: This is the description of the position (currently lorem ipsum text)
- Salary: This is the position's Base Salary, formatted as $xx,xxx.00 (HINT, you can use the **number:'.2'** DecimalPipe (https://angular.io/api/common/DecimalPipe) to format the number after the "$"

Finally, refresh your app and ensure that all the data is rendering correctly. Once this is complete, you're ready to submit your assignment!

Assignment Submission:

- Add the following declaration at the top of your app.component.ts file:

```
/*********************************************************************************
* WEB422 – Assignment 05
* I declare that this assignment is my own work in accordance with Seneca  Academic Policy.  No part of this
* assignment has been copied manually or electronically from any other source (including web sites) or
* distributed to other students.
*
* Name: _____ Student ID: _____ Date: _____
*
*********************************************************************************/
```

- Compress (.zip) the all files in your Visual Studio code folder **EXCEPT the node_modules folder** (this will just make your submission unnecessarily large, and all your module dependencies should be in your package.json file anyway).
- Submit your compressed file (without the node_modules folder) to My.Seneca under **Assignments** -> **Assignment** 5

Important Note:

- **NO LATE SUBMISSIONS** for assignments. Late assignment submissions will not be accepted and will receive a **grade of zero (0)**.
- After the end (11:59PM) of the due date, the assignment submission link on My.Seneca will no longer be available.
- Submitted assignments must run locally, ie: start up errors causing the assignment/app to fail on startup will result in a **grade of zero (0)** for the assignment.