WEB422 Assignment 4 (Ver1.1)

Submission Deadline:

Friday, July 17th 2020 @ 11:59pm

Assessment Weight:

9% of your final course Grade

Objective:

For this assignment, we will be working with a professionally designed Bootstrap 4 "Blog" theme from https://bootstrapious.com/. (**Important Note:** we are using the "free" version of this template, which means we must keep the footer link back to Bootstrapious, ie: "Template By Bootstrapious" intact in the footer)

The goal of this assignment is to use our knowledge of Angular to begin development and prototyping of a useable and scalable Blogging system User Interface (UI). We must use the provided HTML & CSS to help guide our design decisions and develop custom components to work with and display data. We will not have any "live" data for this blog, but we will have some test data that we can use to ensure that all of the components function properly.

Specification:

Step 1: Creating an Angular App & Downloading the Files

To begin this project we must create a new Angular app using the @angular/cli command line tool: "ng". If you do not have it installed yet, you can pull it from npm using the command:

npm install -g @angular/cli

Once the Angular CLI has been installed, you should be able to kickstart a new Angular application with the following command (**Note:** The options here enable routing, skip testing and skip the initialization of a "git" repository, respectfully)

ng new web422-a4 --routing -S -g

With our app in place, we must now download the template files. You will not have to download anything directly from "bootstrapious" – instead, we have provided you with the main files that you will need here:

https://ict.senecacollege.ca/~patrick.crawford/shared/winter-2020/web422/A4/static.zip

Step 2: Static Content & Main "View" Components

Before we start adding the HTML & CSS, we have to make a few updates to our index.html file, ie: Add the following HTML just before the closing <head> tag to the "**index.html**" file. The paths won't make any sense yet, but we'll fix that in the next step:

```
<!-- Bootstrap CSS-->
<link rel="stylesheet" href="/assets/vendor/bootstrap/css/bootstrap.min.css">
<!-- Font Awesome CSS-->
<link rel="stylesheet" href="/assets/vendor/font-awesome/css/font-awesome.min.css">
<!-- Custom icon font-->
<link rel="stylesheet" href="/assets/css/fontastic.css">
<!-- Google fonts - Open Sans-->
<link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Open+Sans:300,400,700">

<!-- theme stylesheet-->
<link rel="stylesheet" href="/assets/css/style.default.css" id="theme-stylesheet">
<!-- Custom stylesheet - for your changes-->
<link rel="stylesheet" href="/assets/css/custom.css">
<!-- Favicon-->
<link rel="shortcut icon" href="/assets/img/favicon.ico">
<!-- Tweaks for older IEs--><!--[if lt IE 9]>
   <script src="https://oss.maxcdn.com/html5shiv/3.7.3/html5shiv.min.js"></script>
   <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script><![endif]-->
```

Next, we must copy the following folders (**from** our download in Step 1) **into** the "**src/assets**" folder within our newly created Angular application:

- css
- vendor
- img
- fonts

Finally, wipe out all of the content from **app.component.html** and replace it with **<p>App Works!</p>** (just to render something for the time being).

Creating new "view" Components

Now that all of our styles are in place, we must create the primary components that will be rendered using client side routing. These can be thought of as the "view" components. Within most of these components, we will be creating smaller components. However, before we do that, let's get the boilerplate code for each "view" added in its own component:

For the next steps, use the Angular CLI (ng) to create the following components with boilerplate templates from the files provided (step 1)

HeaderComponent (header.component.ts)

- Once you have created this component, copy the contents of the file "**header.html**" into the contents of "header.component.html" (replacing the automatically-generated paragraph element) and save the file.

- If you wish to test it at this time, you can add the **<app-header></app-header>** component to the **app.component.html** file

FooterComponent (footer.component.ts)

- Once you have created this component, copy the contents of the file "**footer.html**" into the contents of "footer.component.html" (replacing the automatically-generated paragraph element) and save the file.

- If you wish to test it at this time, you can add the **<app-footer></app-footer>** component to the **app.component.html** file

HomeComponent (home.component.ts)

- Once you have created this component, copy the contents of the file "**home.html**" into the contents of "home.component.html" (replacing the automatically-generated paragraph element) and save the file.

- If you wish to test it at this time, you can add the **<app-home></app-home>** component to the **app.component.html** file (ideally between the header and footer)

BlogComponent (blog.component.ts)

- Once you have created this component, copy the contents of the file "**blog.html**" into the contents of "blog.component.html" (replacing the automatically-generated paragraph element) and save the file.

- If you wish to test it at this time, you can add the **<app-blog></app-blog>** component to the **app.component.html** file (ideally between the header and footer – you may want to comment out <app-home></app-home> if it was previously added)

PostComponent (post.component.ts)

- Once you have created this component, copy the contents of the file "**post.html**" into the contents of "post.component.html" (replacing the automatically-generated paragraph element) and save the file.

- If you wish to test it at this time, you can add the **<app-post></app-post>** component to the **app.component.html** file (ideally between the header and footer– you may want to comment out <app-blog></app-blog> if it was previously added)

PageNotFoundComponent (page-not-found.component.ts)

- You are free to put whatever you like in this template – it's the "404" view for our application

Step 3: Main Route Configuration

Now that our main components are in place, it's a good idea to get "routing" up and running to make it easier to test and develop each individual view. To begin, configure the following routes in your **app-routing.module.ts** file by adding the configurations to the **Routes** array:

- **Path: "home" -** Shows the **HomeComponent**
- **Path: "blog" -** Shows the **BlogComponent**
- **Path: "post" -** Shows the **PostComponent**
- **Path: "" -** Redirects to the "/home" Route
- **No Route Found -** Shows the **PageNotFoundComponent**

Once this is complete, place the "router outlet" component **(<router-outlet></router-outlet>)** between the <app-header></app-header> and <app-footer></app-footer> components in your **app.component.html** file.

As a final task before we start configuring our individual "views", let's make the "header" work with our Angular Routing components:

- Replace all 3 **href** properties with **routerLink** – this will ensure that whenever they are clicked, it will use our Angular routing instead of the default browser routing.

- Next, to ensure that our menu items are correctly **highlighted**, add the following "directive" to the **same 3 anchor elements**:
  - **routerLinkActive="active"**

Step 4: Mock "Blog" Data

Before we start dividing our views into reusable components (you probably have spotted all of the "widgets" in the side bar – these are perfect candidates), let's add a few fake blog posts.

Since we're using typescript, it makes sense for our data to be "typed" – ie: we must have a "class" to define the "shape" of each post. To achieve this, add the following 2 files / content

Comment.ts

```
export class Comment{
  _id: string
  author: string;
  comment: string;
  date: string;
}
```

BlogPost.ts

```
import { Comment } from "./Comment";
```

```
export class BlogPost{
    _id: string
    title: string;
    postDate: string;
    featuredImage: string;
    post: string;
    postedBy: string;
    comments: Array<Comment>;
    category: string;
    tags: Array<string>;
    isPrivate: Boolean;
    views: number;
}
```

Next, we must copy the **blogData.json** file (from our files included with the download in step 1) and place it within the "app" folder. This will serve as our sample data for the blog.

Unfortunately, if we want to actually "import" this json file in any of our components, we must add a couple of "compiler options" to our typescript configuration, ie: add the following two properties to the "**compilerOptions**" property in the "**tsconfig.base.json**" file

- "resolveJsonModule": true
- "esModuleInterop": true

Now, if we want to ever pull in the data, we can add the following "import" lines in our component files:

import blogData from '../blogData.json';
import { BlogPost } from '../BlogPost';

In fact, since our "BlogComponent" will need the blog data, let's proceed to add the above two "import" lines to the top of the **blog.component.ts** file now.

Inside the **BlogComponent** class, add the following property declaration:

blogPosts: Array<BlogPost> = blogData;

This will ensure that our data is loaded as "blogPosts" before we start the next part of the assignment (in the future, this would come from a service which accesses a WEB API endpoint for data).

Step 5: "Blog" view components

If you open your app to the "blog" route, you will notice that we have quite a few "containers" in the UI that manage data. In the next part of this assignment, we will build out components for these containers

PostCardComponent

This component will be responsible for rendering the above chunk of UI (**Hint**: all the code *inside* <div class="post col-xl-6">…</div>, so use that as your template starting point).

To get started, create the component using the ***ng g c*** (ng generate component) command, then proceed to implement the component according to the following specifications:

- It must receive a single property **post**. The type of this property will be **BlogPost** (therefore, { BlogPost } must be imported from '../BlogPost' and the @Input() decorator must also be used. Do not forget to import "Input" from @angular/core
- It must be rendered using the class property: "class="post col-xl-6", ie: "<app-post-card class="post col-xl-6">"
- Its "post-thumbnail" element must show the post's "featuredImage"
- Its "date meta-last" element must show the post's "postDate"
- Its "cateogory" element must show the post's "category"
- Its "<h3 class="h4"> element must show the post's "title"
- Its " <p class="text-muted"> element must be changed to a <div> element instead (ie: "<div class="text-muted">…</div>". Also its "innerHtml" property must be set to the post's "post"
- Its "avatar" image must be changed to "/assets/img/user.svg"
- Its "title" element must show the post's "postedBy"
- Its "date" element (next to the clock) must show the post's "postDate"
- Its "meta-last" element must show the number of comments, ie comments.length

Once this is done, use the new "app-post-card" in place of the existing elements in the "blog.component.html" template. You should be able to show one app-post-card per "post" in the "blogPosts" array using *ngFor.

Using our sample data, your "blog" page should now look like the below screenshot:

SearchWidgetComponent

This component simply renders the "widget-search" element, so use that as your template starting point.

To get started, create the component using the *ng g c* (ng generate component) command. We will use this to render the template only - we not be implementing any logic at this time.

LatestPostsComponent

This component will be responsible for rendering the above chunk of UI (**Hint**: this is the <div class="widget latest-posts"> element, so use that as your template starting point).
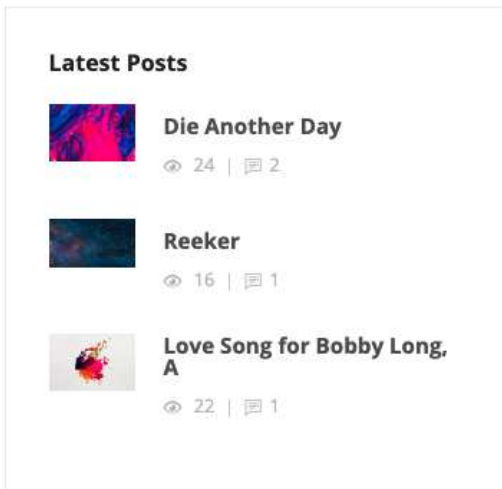
To get started, create the component using the ***ng g c*** (ng generate component) command, then proceed to implement the component according to the following specifications:

- It must receive a single property **posts**. The type of this property will be **Array<BlogPost>** (therefore, { BlogPost } must be imported from '../BlogPost' and the @Input() decorator must also be used. Do not forget to import "Input" from @angular/core
- Inside the template, you will notice three "item" elements, each nested inside an "<a href="#">" element. The idea is that we must render one "item" link for every "post" in the "posts" array according to the following:
  - Its "image" element must show the post's "featuredImage"
  - Its "title" element must show the post's "title"
  - Its "views" element must show the post's "views"
  - Its "comments" element must show the number of comments, ie comments.length

Note: When you include the component, you can limit the number of "blogPosts" that you provide to the component using the code:

<app-latest-posts [posts]="blogPosts.slice(0,3)"></app-latest-posts>

When complete, your component should look like:

CategoriesComponent

This component will be responsible for rendering the above chunk of UI (**Hint**: this is the <div class="widget categories"> element, so use that as your template starting point).

To get started, create the component using the ***ng g c*** (ng generate component) command, then proceed to implement the component according to the following specifications:

- In the future, this component would have to worry about fetching the category data on its own. In the meantime, hardcode the following array as the component's single property:
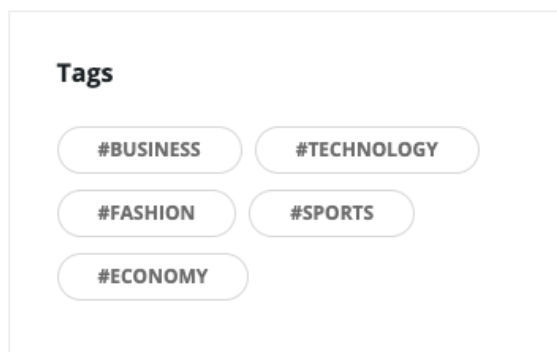
  categories: Array<**any**> = [
    {cat: "Crime", num: 2},
    {cat: "Comedy", num: 1},
    {cat: "Musical", num: 1},
    {cat: "Adventure", num: 2},
    {cat: "Drama", num: 2},
    {cat: "Action", num: 2},
    {cat: "Documentary", num: 1},
    {cat: "Thriller", num: 1}
  ];

- In the template, you will notice that <div class="item d-flex justify-content-between">…</div> is repeated once per category. We will use our "categories" array to accomplish the same output, ie: looping through the array and outputting one "item" per element.

When complete, your component should look like:

Categories

Crime	2
Comedy	1
Musical	1
Adventure	2
Drama	2
Action	2
Documentary	1
Thriller	1

TagsComponent

**Tags**

#BUSINESS   #TECHNOLOGY
#FASHION   #SPORTS
#ECONOMY

The **TagsComponent** functions very much like the CategoriesComponent in that it doesn't accept any props and (for now) we will hardcode the existing tags into a "tags" array, ie:

```
tags: Array<string> =[
  "#funny",
  "#dramatic",
  "#rental",
  "#seeagain",
  "#spooky",
  "#worththecost",
  "#lovedIt",
  "#scary",
  "#silly",
  "#good4kidz"
];
```

In the template, you will notice that <li class="list-inline-item">… is repeated once per category. We will use our "tag" array to accomplish the same output, ie: looping through the array and outputting one "list-inline-item" per element.

When complete, your component should look like:
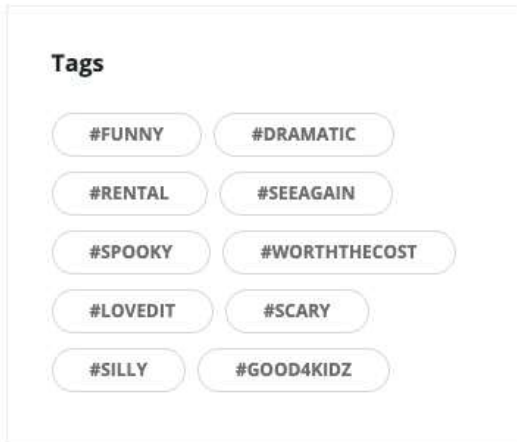
Step 6: "Post" view components

Now that we have completed all of the "Blog" view components, we can move on to the "Post" view components. The good news here is that that we can reuse most of the components from the "Blog" view

To get started, we will follow along with the same procedure as the "Blog" component, ie adding in the "blogPosts" array:

First, add the following "import" lines in our post.component.ts file:

import blogData from '../blogData.json';
import { BlogPost } from '../BlogPost';

Next, inside the **PostComponent** class, add the following property declaration:

blogPosts: Array<BlogPost> = blogData;

This will ensure that our data is loaded as "blogPosts" (once again - in the future, this would come from a service which accesses a WEB API endpoint for data).

Reusing the <aside> from blog.component.html

If you take a look at your completed blog.component.html, you should notice that your <aside> element is now much smaller than it used to be, ie:

<aside class="col-lg-4">
   <!-- Widget [Search Bar Widget]-->

```
   <app-search-widget></app-search-widget>
   <!-- Widget [Latest Posts Widget] -->
   <app-latest-posts [posts]="blogPosts.slice(0,3)"></app-latest-posts>
   <!-- Widget [Categories Widget]-->
   <app-categories></app-categories>
   <!-- Widget [Tags Cloud Widget]-->
   <app-tags></app-tags>
</aside>
```

You will also notice that we have the exact same <aside> element in our post.component.html file (starting at line 100). For this next step, simply copy your full <aside> element from your blog.component.html and paste it in place of the existing <aside> within your post.component.html file!

PostDataComponent (The last one!)

This Final Component is basically the entire "post single" div from the sample html, ie: "<div class="post-single">…</div>" (everything from line 6 to line 94 inclusive). So, to get started wiring up this component, simply cut all of that HTML including the <div class="post-single">…</div> container element, and **paste** it into your post-data.component.html. Then, in its place add back:

<app-post-data [post]="blogPosts[0]"></app-post-data>

Note: We will simply be hard-coding the first blog post for now.

Next, once your html is in the right place (post-data.component.html) we can start working with the template according to the following specifications:

- It must receive a single property **post**. The type of this property will be **BlogPost** (therefore, { BlogPost } must be imported from '../BlogPost' and the @Input() decorator must also be used. Do not forget to import "Input" from @angular/core
- The "post-thumbnail" must show the post's "featuredImage"
- The "category" must show the post's "category"
- The <h1>…</h1> element must show the post's "title"
- The "avatar" must be changed to show "/assets/img/user.svg"
- The "title" (directly beneath avatar) must show the post's "postedBy"
- The "date" element (next to the clock) must show the post's "postDate"
- The "views" element (next to the eye) must show the post's "views"
- The "comments" element must show the number of comments, ie comments.length
- The "post-body" innerHTML property must be set to the post's "post"
- The "post-tags" must show one "…" element for every tag in the post's "tags" array such that the content of the <a> shows the value of a tag used in the post
- The "no-of-comments" must show the number of comments, ie comments.length

The last piece of this component implementation is the "comments" list. To complete this, we must iterate over the "comments" array and display a <div class="comment">…</div>" for every comment in the array. This will involve updating the following "comment" related elements:

- The "title" should show the current comment's "author"
- The "data" should show the current comment's "date"
- The "comment body" innerHTML property must be set to the current comment's "comment"

Step 7:Updating Links and Testing

You will notice that there are a few things that are missing from our solution. Most notably the out of date, static blog posts on the Home page and in the footer. If you require additional practice with components please feel free to use the knowledge gained from updating the "Blog" and "Post" views to refactor these views as well.

Also, you will notice that there's a lot of "href" links still in the solution. Unless the app requires the user to navigate away (ie: heading to twitter, etc.) then "href" should be replaced with "routerLink", ie:

- **href="#"** should be replaced with **routerLink**
- **href="post.html"** should be replaced with **routerLink="/post"**

Once this is complete, you should be ready to submit. However have a look at the full-page screenshots of the app and compare your results with what you see. Everyone is using the data, so it should look identical (unless you added some additional styling).

- Home View (Sample)
- Blog View (Sample)
- Post View (Sample)
- 404 View (Sample)

Assignment Submision:

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

```
/*****************************************************************************
 * WEB422 – Assignment 04
 * 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** 4

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.