WEB422 Assignment 6

Submission Deadline:

Friday, August 14th 2020 @ 11:59pm

Assessment Weight:

9% of your final course Grade

Objective:

The purpose of this assignment is to add additional functionality to our ongoing "Blog" project. This will include the ability to create new posts as well as edit / delete existing posts, allow users to comment on posts and track the number of "views" per post.

Unfortunately, implementing the complete list of features for this blog will take too much time and this is our final assignment. We will however, end with a ton of functionality in place and it will serve as an excellent starting point if you wish to build out the complete project. This might include implementing the search functionality, updating the "Home" page components to show real blog data and/or ensuring users must be logged in before modifying blog data (either directly through the app or indirectly through the API).

NOTE: If you are unable to start this assignment because there was an issue with your Assignment 5, please email your professor for a fresh (working) copy.

Specification:

To begin this assignment, you will have to make a copy of your (now complete) assignment 5. Once this is complete, open up the copied folder in Visual Studio Code and start working from there.

Step 1: Creating new PostService Methods

One of the major features of this assignment is to enable the creation, modification and deletion of blog posts using with a UI within the app. To facilitate this, we must create the following new methods within our PostService (post.service.ts):

getAllPosts():Observable<BlogPost[]>

- Fetches all blog posts using the path: YOUR API/api/posts?page=1&perPage=**Number.MAX_SAFE_INTEGER** where "Number.MAX_SAFE_INTEGER" is a constant value of: $2^{53} - 1$ (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/MAX_SAFE_INTEGER).

  This will ensure that all of the blog posts are returned using a single AJAX request.

newPost(data: BlogPost): Observable<any>

This function must invoke the "post" method on the injected "http" service with the data parameter as the body of the request, ie:

return this.http.post<any>(`YOUR API/api/posts`, data);

updatePostById(id: string, data: BlogPost): Observable<any>

This function must invoke the "put" method on the injected "http" service with the data parameter as the body of the request, ie:

return this.http.put<any>(`YOUR API/api/posts/${id}`, data);

deletePostById(id: string): Observable<any>

This function must invoke the "delete" method on the injected "http" service with the data parameter as the body of the request, ie:

return this.http.delete<any>(`YOUR API/api/posts/${id}`);

Step 2: Creating an "Admin" Section (New Components)

To help manage our blog posts, we will create a new "admin" section (ie: all routes responsible for adding, editing or removing blog posts will begin with "admin/".

Before we begin to add our new "admin" components, we must be sure to **add the FormsModule** to the **imports** array in **app.module.ts** (enabling our app to use template-driven forms within the components).

Next, create 3 new components and add them to the routing configuration as follows:

PostsTableComponent (route: admin)

This component will be responsible for showing all of the current posts within the blog in a single table, acting as our primary interface to access existing posts. This means that the component must:

- Import & inject the "PostService" service
- Import & inject the "Router" service from "@angular/router"
- Create a property blogPosts of type: Array<BlogPost> with a default value of [] (empty array)
- Set the value of "blogPosts" in the ngOnInit() method using the "getAllBlogPosts()" method of the "PostService" (defined above)

The component's template must consist of a single table with the fields: "Title" "Post Date" & "Category" as well as a button with the text "+ New Post". Each row of the table will be a "blogpost" from the "blogPosts" array.

The following HTML can be used as a starting point:

```
<br />
<div class="container">
  <div class="row">
    <div class="col-md-12">
      <a class="btn btn-success btn-sm pull-right">+  New Post</a><br /><br />
      <table class="table table-hover">
        <thead>
          <tr>
            <th>Title</th>
            <th>Post Date</th>
            <th>Category</th>
          </tr>
        </thead>
        <tbody>
          <tr>
            <td>title</td>
            <td>post Date (formatted using the "date: 'mediumDate'" pipe</td>
            <td>post category</td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
</div>
<br />
```

With the template in place and rendering the blog data, the rest of the component's functionality can be implemented, including:

- Ensuring that the "+ New Post" links to "/admin/newPost" (using client-side routing)
- Ensuring that when a specific row is clicked, the user will be redirected to the route "admin/post/**id**" where **id** is the "_id" value of the post in the row clicked. This will involve the following steps:
    - Writing a click handler function in the component, ie "rowClicked(e, id)" where e is $event, and "id" is the post id of the row clicked. This function will use the "**id**" value with the "router" service to navigate to the path: "admin/post/**id**" using the "navigate" method.

      (see "Linking to Parameterized Routes from the notes here: https://web422.ca/notes/angular-routing-contd
    - Invoking the "rowClicked" event for each <tr> element in the table of posts. This should ensure that the user can navigate to a specific post by clicking on it in the row.

EditPostComponent (route: admin/post/:id)

This component will show the post information for the post with an _id value of the route parameter ":id" in an editable form with the ability to "remove" the post (**Note**: We will not be editing any of the post "comments" for this assignment).

Editing an exist post will primarily involve editing the "Title", "Featured Image", "Category", "tags" as well as the "Post" itself. This means that the component must:

- Import & inject the "PostService" service
- Import & inject the "Router" and "ActivatedRoute" services from "@angular/router"
- Create a property blogPost of type: BlogPost
- Create a property tags of type string
- Set the value of "blogPost" in the ngOnInit() method using the "getPostById(**id**)" method of the "PostService" (defined above) such that the value of **id** is the value passed in using the route parameter "id" (this value should be something like: "5e5ffe8b0f706590aae79d92". **HINT:** the "snapshot" method of the "ActivatedRoute" service can be used here (https://web422.ca/notes/angular-routing-contd)
- Set the value of the component's tags property to this.blogPost.tags.toString(); once the blogpost is loaded (previous step). This will allow us to show the tag values as a single string in the UI.

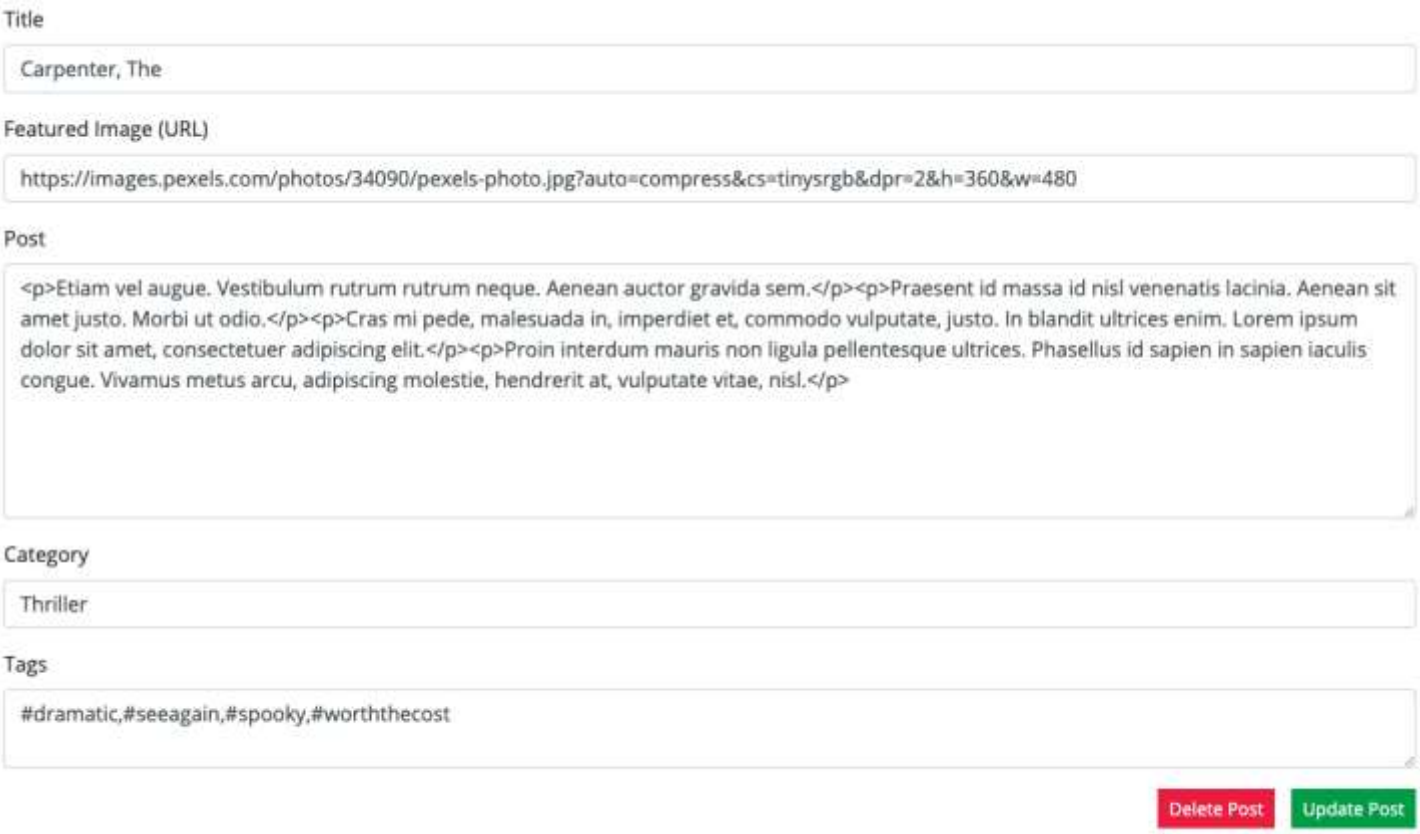
Once this is complete, the template must be created. The following HTML can be used as a starting point (note: all properties can bind directly to the "blogPost" properties, with the exception of "tags", which much bind to the component's tags property:

```
<br />
<div class="container">
  <div class="row">
    <div class="col-md-12">
      <form>
        <div class="form-group">
          <label>Title</label>
          <input type="text" class="form-control">
        </div>
        <div class="form-group">
          <label>Featured Image (URL)</label>
          <input type="text" class="form-control">
        </div>
        <div class="form-group">
          <label>Post</label>
          <textarea class="form-control" style="min-height:200px"></textarea>
        </div>
        <div class="form-group">
          <label>Category</label>
          <input type="text" class="form-control">
        </div>
        <div class="form-group">
          <label>Tags</label>
          <textarea class="form-control"></textarea>
        </div>

        <div class="pull-right"><button class="btn btn-danger btn-sm">Delete Post</button>   <button type="submit" class="btn btn-success btn-sm">Update Post</button></div>
      </form>
    </div>
```

```
  </div>
</div>
<br />
```

Once this template has been implemented correctly, you should be able to see information for a specific post, ie:



The remaining tasks involve implementing the functionality for the "Update Post" and "Delete Post" buttons. Let's start with "Update Post":

- Create a "submit" handler, ie: "formSubmit()" that executes when the form is submitted. It must perform the following tasks:
    - Set the value of blogPost.tags to the value of the "tags" property, converted to an array (ignoring optional whitespace). This can be done using the code:

      this.tags.split(",").map(tag => tag.trim()); // convert the string to an array and remove whitespace

    - Invoke the "updatePostById(id)" method of the PostService using the value of blogPost._id and passing the value of blogPost. Once this method has completed, use the "router" service to redirect to the "/admin route using the "navigate" method, ie: navigate(['admin']);

With "Update Post" complete, let's move on the "Delete Post" button:

- Create a "click" handler, ie "deltePost()" that executes when the "Delete Post" button is clicked. It must perform the following tasks:

    - Invoke the "deletePostId(id)" method of the PostService using the value of blogPost._id. Once this method has completed, use the "router" service to redirect to the "/admin route using the "navigate" method, ie: navigate(['admin']);

NewPostComponent (route: admin/newPost)

The new Post component is very similar to our EditPost component, however we will not be populating the form with existing data and there will not be a "Delete Post" button.

To begin, update the NewPostComponent according to the following specification:

- Import & inject the "PostService" service
- Import & inject the "Router" service from "@angular/router"
- Create a new property "blogPost" of type "BlogPost" with a default value of **new BlogPost();**
- Create a property tags of type string
- Copy the same HTML template from EditPostComponent (edit-post.component.html) and paste it into the template for this component (new-post.component.html), making sure to:
    - Remove the "Delete Post" button
    - Change the text "Update Post" to "Add Post"
- Create a "formSubmit()" method and ensure that it's executed when the form is submitted. This function must adhere to the following specifications:
    - Set the value of blogPost.tags to the value of the "tags" property, converted to an array (ignoring optional whitespace). This can be done using the code:

      this.tags.split(",").map(tag => tag.trim()); // convert the string to an array and remove whitespace
    - Set the value of blogPost.isPrivate to false
    - Set the value of blogPost.postDate to new Date().toLocaleDateString();
    - Set the value of blogPost.postedBy to "WEB422 Student"
    - Set the value of blogPost.views to 0
    - Invoke the "newPost(post)" method of the PostService using the value of blogPost. Once this method has completed, use the "router" service to redirect to the "/admin route using the "navigate" method, ie: navigate(['admin']);

Step 2: Commenting on Posts

The next missing piece of functionality is the ability for users to comment on specific blog posts. To enable this functionality, we must work primarily with the PostDataComponent, since it encapsulates the form controls for adding new comments:

- Open the post-data.component.html file and locate the <form> element with class "commenting-form". Here, we must add an ngSubmit event handler that invokes a submit function, ie: "submitComment()" and remove the 'action="#"' property
- In the PostDataComponent, create two new fields:
    - commentName: string;
    - commentText: string;
- In the post-data.component.html file, locate the <input type="text"> element with the name "username" and use the template-driven forms binding syntax to sync its value to "commmentName"
- Similarly, locate the <textarea> element with the name "usercomment" and use the template-driven forms binding syntax to sync its value to "commentText"
- In the PostDataComponent add the previously mentioned "submitComment()" function with the following functionality:
    - "push" a new object onto the "comments" array for the current post with the properties:
        - author: this.commentName
        - comment: this.commentText
        - date: new Date().toLocaleDateString()
    - Invoke the "updatePostById(id,data)" method, passing the current post's _id attribute as the "id" and the current post as the "data" property
    - Once the updatePostById() has completed (ie: in the first "subscribe" callback method), reset the values of this.commentName & this.commentText

Step 3: Counting the "Views"

The last piece of functionality that we need to update before we publish our app, is to track the number of times each post has been viewed. Currently, each post has a "views" property, but it remains static, despite multiple viewings of the same post. To fix this, we must edit the "PostDataComponent" according to the following specification:

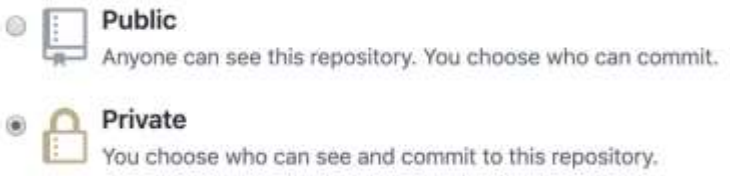
- Notice how, in the ngOnInit() method, we invoke getPostById(). When this is complete, we set the value of this.post to the returned data from our API. For us to track the number of times this component has been "viewed", we must add additional code in this callback (subscribe) method.
    - After the code to set the value of this.post (in the subscribe method), increase the value of "views" on this.post by one
    - Invoke the "updatePostById(id, data)" method on the injected "PostService" and provide the value of the current post._id and post as its parameters. Since we do not have to execute any code once its complete, we can invoke the "subscribe" method with no paramters, ie:

      this.postData.updatePostById(this.post._id, this.post).subscribe();

Step 4: Publishing the App

As a final step, we must publish the application online. This can be any of the methods discussed in class, ie: **Heroku**, **Netflfy** or Seneca's **Matrix** server.

However, if you choose to publish the application on Netflify, you must use a **private Github Repository:**



Assignment Submission:

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

```
/*********************************************************************************
* WEB422 – Assignment 06
* 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: _____
*
* Online Link: _____
*
********************************************************************************/
```

- 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 6**

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.