Project 4: Markdown Editor Using Angular

Change History

1. 11/17/18 5:10AM: A paragraph in Part B is reworded to make it clear that a server-side code change is needed for `/editor/` authentication
2. 11/18/18 9:25AM: A paragraph has been added to the submission instruction to make sure that localhost is used in the submitted code as the server hostname, not 192.168.X.X
3. 11/18/18 9:50AM: Instead of suggesting the `jsonwebtoken` module for JWT decoding, we provide a custom function that can extract the payload from a JWT

Overview

In this project, you will learn and use Angular, a popular front-end Web-development framework, to develop a more advanced and dynamic version of markdown blog editor that uses the REST API you implemented in Project 3 to help users write, update, and publish blogs on the server.

Development Environment

The development for Project 4 will be done using the same docker container that you created in Project 3, which can be stared with the command:

```
$ docker start -i mean
```

Make sure that MongoDB, NodeJS, and Angular is configured correctly by running the following commands:

```
$ mongo -version
$ node --version
$ ng --version
```

In writing the code for Project 4, you are likely to encounter bugs in your code and need to figure out what went wrong. Chrome Developer Tools is a very popular tool among Web developers, which allows them to investigate the current state of any web application using an interactive UI. We strongly recommend it for Project 4 and make it part of your everyday tool set. There are many excellent online tutorials on Chrome Developer Tools such as this one.

Part A: Learn Angular and Basic Concepts

Angular is a front-end Web-development framework that makes it easy to build applications for the Web. Angular combines *declarative templates*, *dependency injection*, *end-to-end tooling*, and integrates best development practices to solve challenges in Web front-end development. Angular empowers developers to build applications that live on the Web, mobile, or the desktop.

Before starting on any materials related to Angular, first get yourself familiar with JavaScript, and new language constructs from recent JavaScript standards such as classes and modules. The latest Angular version uses TypeScript, an extended version of JavaScript, as its primary language. Fortunately, most Angular code can be written with just the latest JavaScript, with a few additions like types for dependency injection, and decorators for metadata. We go over essential TypeScript for Angular in class, and the class lecture notes is available.

Angular official website provides an excellent introductory tutorial on Angular development: Tour of Heroes tutorial. It introduces the fundamental concepts for Angular development by building a simple demo application.

- Tour of Heroes tutorial

It may take some time to finish this tutorial, but we believe **following this tutorial is still the most effective and time-saving way to get yourself familiar with the Angular development**.

Note that when you follow the tutorial using the Angular CLI preinstalled in our container, you will need to use the following command to "run" your Angular code:

```
$ ng serve --host 0.0.0.0
```

not `ng serve --open` as described in the tutorial.

Note on `--host` option: By default, Angular HTTP server binds to only "localhost". This means that if any request comes from other than localhost, it does not get it. When Angular runs on the same machine as the browser, this is not a problem. Angular binds to localhost and the browser sends a request to localhost. But when Angular runs in a docker container, the localhost of Angular is different from the localhost of the browser. Angular sees localhost of *container* and browser sees the localhost of the *host*. By adding "–host 0.0.0.0", we instruct Angular to bind to *all network interfaces* within the container, not just localhost, so that Angular is able to get and respond to a request forwarded by Docker through network forwarding.

If you have previous Angular or similar Web-framework development experience, you can choose to read the Angular documentation directly instead. However, for most students who have not worked with Angular extensively before, reading the documentation may take more time than following the step-by-step tutorial. Thus, our recommendation is to start with the tutorial and then go over the documentation after you get familiar with the basics.

Some caveats: Do not confuse Angular with AngularJS! When you search for Angular related issues on the Internet, **please use Angular 2 or Angular CLI as search keywords, not AngularJS.** AngularJS is an older version of the Angular framework and is no longer a recommended version. The difference between the "old" AngularJS and the "new" Angular is quite extensive, as you can read from more detailed comparison articles on the Web. For our project, you may ignore previous AngularJS versions and just learn the latest Angular CLI using the links and tutorials provided in this spec.

After you finish the tutorial, go over the following questions and make sure you can answer them by yourself.

- What is a Component in Angular?
- What is a Template? What are Directives in a Template?
- What is a Module in Angular? How is NgModule different from a JavaScript module?
- How does Angular support Data binding?
- What is a Service and how is Dependency injection done in Angular?
- How is Routing done in Angular?
- What are commonly used Angular CLI commands, such as generating a component or service?

Please read corresponding sections in the Angular documentation for review if the answer to any question is not clear.

Now you have equipped with enough Angular knowledge to get started with Project 4. Good Luck!

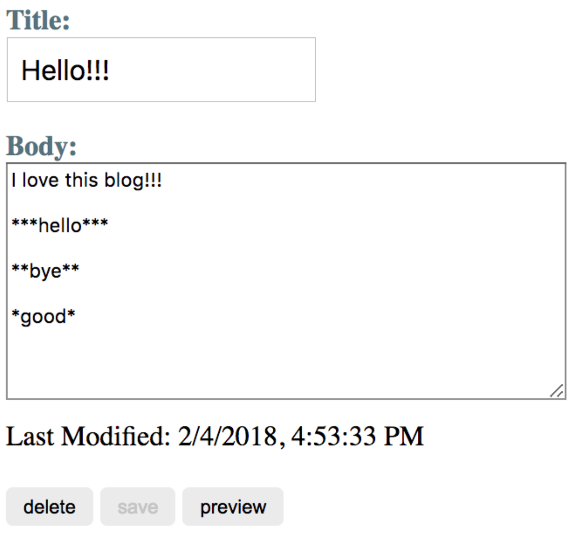
Part B: Project Demo and Requirements

Project 4 is all about a front-end markdown blog editor and previewer. It should be implemented as a single-page application (SPA), which means that your entire application runs on a “single page.” The website interacts with the user by dynamically updating only a part of the page rather than loading an entirely new page from the server. This approach avoids long waits between page navigation and sudden interruptions in the user interaction, making the application behave more like a traditional desktop application. A typical example of a SPA is Gmail.

An important feature of a SPA is that **a specific state of the application is associated with the corresponding url**, so that a user does not accidentally exit from the app by pressing a back button. When a user presses the browser back button, the user should go to the *previous state within the app* (unless the user just opened the app) as opposed to exiting from the app and go to the page visited before the app. As an example, open Gmail, click on a few mail messages and/or folder labels, and then press the browser back button. You will see that you do not exit from the Gmail app, even though all your interaction in the app happened on a *single page*, and, technically, the “previous page” in your visit history should be the page that you visited *before* you opened the Gmail app. In addition, if you cut and paste the Gmail’s drafts folder URL https://gmail.com/#drafts into the browser address bar, you will see that you directly land on the draft folder of Gmail, not its generic start page. You will soon learn how to implement this behavior by using the *routing module* of Angular.

We made a demo website of Project 4 available at http://oak.cs.ucla.edu/classes/cs144/project4/demo/. It is rather simple, does not contain many CSS-styling instructions, does not actually store blog posts on the server, but it still helps you understand the key UI requirements of this project.

In the first image, we show the **edit view** of the application, which allows the user edit a post. In this view, we require you to implement the following functionalities:



Hello!!!

I love this blog!!!

hello

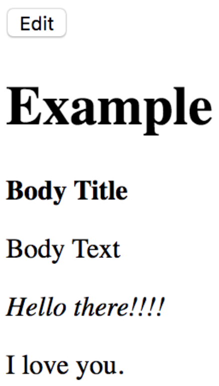
bye

good

Last Modified: 2/4/2018, 4:53:33 PM

- The view shows one `text` input box for the title and one `textarea` for the body.
- The "last modified" date and time is shown below the text boxes.
- The view should contain at least three buttons, "save", "preview", and "delete".
- When the "save" button is pressed, it should permanently save the post and update the last-modified date of the post to the current time.
- When the user "exits" from this view (by pressing the "preview" button, by clicking on another post in the list, by pressing the browser "refresh" or "back" buttons, or by typing a URL into the address bar), any unsaved changes should be saved automatically, so that the *app never loses the user's work*.
- When the "preview" button is clicked, the app should switch to the "preview view" (see below).
- When the "delete" button is clicked, the post should disappear from the "list pane" (described below) and be permanently deleted.

In the second image, we show the **preview view** of the application. In this view, we require you to implement the following functionalities:



Example

Body Title

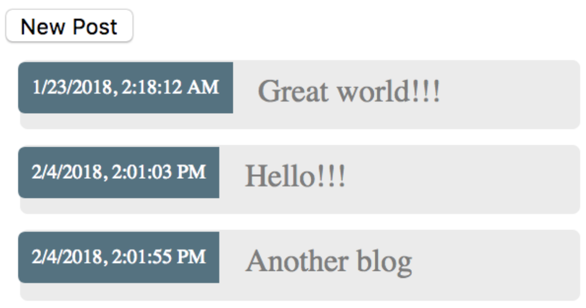
Body Text

Hello there!!!!

I love you.

- The view shows an HTML-rendered preview of the current markdown post, including its title and body.
- There is an "edit" button to switch to the edit view.

In the third image, we show the "list pane", that should meet the following requirements:



1/23/2018, 2:18:12 AM	Great world!!!
2/4/2018, 2:01:03 PM	Hello!!!
2/4/2018, 2:01:55 PM	Another blog

- It shows the list of all blog posts that have been written by the user.
- The posts in the list should be sorted by their "postid" (a unique integer assigned to a post) in the ascending order.
- Each post in the list must show the creation date & time of the post and the title.
- The user can add a new post anytime by clicking the "new post" button, which **opens the edit view of the newly created post on the right side.**

- When a new post is created by the user its postid should be +1 of the largest postid that was created before by the user.
- The user can edit any of existing post by clicking its entry in the list, which **opens the edit view of the post on the right side.**

Note that differently from Project 2, you are required to make the markdown editor as a single-page application and **the "list pane" should be always visible on the left side** of either edit or preview view.

In addition, when the user presses the browser's "back button", the user should go to the "previous state" of the app, not to the page visited prior to your app. For example, If the user opened the app, clicked on the first post in the list pane, and pressed the "preview" button, the user should go to the state before the "preview" button was pressed if the user clicks on the browser back button. In particular, you need to associate the three "states" of our app with the following URL patterns:

URL	state
`/editor/#/`	This default path shows only the list pane, without showing the edit or preview view
`/editor/#/edit/:id`	This path shows the list pane and the "edit view" for the post with `postid=id`
`/editor/#/preview/:id`	This path shows the list pane and the "preview view" of the post with `postid=id`

Note that all user-created/edited blog posts must be stored in MongoDB through the server that you implemented in Project 3. Also, the preview page should be generated by a JavaScript code running inside the browser, using the commonmark.js library. Finally, **if the user tries to access Angular editor at the URL `/editor/` without authenticating herself first, the request must be redirected to `/login?redirect=/editor/`, so that the user should be able to provide her authentication information and automatically come back to the editor.** This will ensure that when the client-side Angular code is loaded in the browser, the browser has already obtained a valid JWT cookie, so that the JWT can be sent to the server when it tries to access the Blog-Management REST API to create, retrieve, and update blog posts by the user. Implementing this redirection-for-authentication mechanism will require minor changes to the server-side code that you implemented in Project 3.

In the rest of the project spec, we describe more detailed guidance on how you can implement the rest of Project 4. However, keep it mind that the rest of our project description is a *suggestion, not a requirement*. As long as your code meets the requirements in Part B, you can implement the rest of your application however you want. We provide further description here in case you need more guidance and help to finish this project.

Part C: Create Project Skeleton using Angular CLI

Now it is time to start working on the project using the Angular Command-Line Interface (CLI). First create a new Angular application using the following command:

```
$ ng new angular-blog
```

This may take a while since a lot of files are fetched and generated. When the skeleton application is created successfully, we will see the following output.

```
Project 'angular-blog' successfully created.
```

You can launch the just-created application using `ng serve --host 0.0.0.0` and access it in your browser at http://localhost:4200/.

```
$ ng serve --host 0.0.0.0
```

Do not forget `--host 0.0.0.0` as we are serving the Angular app from the container. The page you see is the application shell. The shell is controlled by an Angular component named *AppComponent*.

Components are fundamental building blocks of any Angular application. They display data on the screen, listen for user input, and take an action based on that input.

The `angular-blog` directory that contains the initial skeleton code looks like the following:

```
angular-blog
 +- e2e
 +- node_modules
 +- src
    +- app
        +- app.component.css
        +- app.component.html
        +- app.component.spec.ts
        +- app.component.ts
        +- app.module.ts
    +- assets
    +- environments
    +- index.html
    +- styles.css
    +- typings.d.ts
    +- ...
 +- package.json
 +- README.md
 +- ...
```

This may look like a lot of files at the first glance, but don't get overwhelmed. The files you need to touch are *all inside the `src/app` folder*. Other files can be ignored in most cases. In the `src/app` folder, the Angular CLI has created the *root module*, `AppModule`, and the main application component, `AppComponent`.

You'll find the implementation of `AppComponent` distributed over three files:

- app.component.ts — the component class file, written in TypeScript.
- app.component.html — the component template, written in HTML.
- app.component.css — the component's style, written in CSS.

The ".spec.ts" file is used for unit testing, and you can ignore it for now.

Refer to the Application Shell section of the tutorial if you still have confusions about how these files work together to form a component.

Part D: Implement Blog Service

Now you create a "Blog Service" using the following command:

```
$ ng generate service blog --module=app
```

The option `--module=app` inserts the necessary `imports` and `providers` statements to the `AppModule`, so that you don't have to add them yourself. You will see the service files created in the terminal.

```
  create src/app/blog.service.spec.ts (362 bytes)
  create src/app/blog.service.ts (110 bytes)
```

The `BlogService` plays two important roles in our application.

1. It allows other components in the application to store and retrieve blog posts. This service hides the exact storage mechanism of the blog posts, so that other components can be implemented independently of the storage mechanism. In this project, all blog posts are stored in MongoDB through the server implemented in Project 3.
2. `BlogService` works as a "local memory cache" for blog posts, so that (potentially) long delays of the underlying storage mechanism is hidden to other components.

Now, open the `blog.service.ts` file and declare a `Post` class with the following properties:

```
export class Post {
  postid: number;
  created: Date;
  modified: Date;
  title: string;
  body: string;
}
```

Note that we need to `export` this class, so that it can be imported and used by other components of the application. `postid` is the unique id of the blog post, `created` and `modified` are the post's creation and last modification date and time, `title` and `body` are the actual content of the post formatted in markdown.

Add a private `posts` property to the `BlogService` class like the following:

```
private posts: Post[];
```

The property `posts` works as the "memory cache" of all blog posts. When the application starts running, all blog posts by the user will be retrieved from the underlying storage mechanism and be held here. In addition, any changes to a post will be temporarily held here until they are permanently written to the underlying storage.

Now implement the following methods in the `BlogService` class:

1. **fetchPosts(username: string): void** – This method must "populate" the `posts` property by retrieving all blog posts of the current user. Send a GET request to `/api/:username` and populate `posts` property from the response.
2. **getPosts(username: string): Post[]** – This method simply returns `posts`.
3. **getPost(username: string, id: number): Post** – Find the post with `postid=id` from `posts` and return it.
4. **newPost(username: string): Post** – Create a new post with a new postid, an empty title and body, and the current creation and modification times, add it to `posts`, send a `POST` request to `/api/:username/:postid` (after setting up the response event handler), and return the newly created post.

   The response handler should do nothing if the response status code is "201 (Created)". Otherwise, it should delete the newly created post from `posts`, display an alert message saying that there was an error creating a new post at the server, and navigate to `/`, the "list pane" of the editor.
5. **updatePost(username: string, post: Post): void** – From `posts`, find a post whose postid is the same as `post.postid`, update its title and body with the passed-in values, change its modification time to now, and send a `PUT` request to `/api/:username/:postid` (after setting up the response event handler). If no such post exists, do nothing.

   The response event handler should do nothing if the response status code is "200 (OK)". Otherwise, it should display an alert message saying that there was an error updating the post at the server, and navigate to the "edit view" of the post.
6. **deletePost(username: string, postid: number): void** – From `posts`, find a post whose postid is the same as the passed in value, delete it from `posts`, and send a `DELETE` request to `/api/:username/:postid` (after setting up the response event handler). If no such post exists, do nothing.

The response event handler should do nothing if the response status code is "204 (No content)". Otherwise, it should display an alert message saying that there was an error deleting the post at the server, and navigate to `/`, the "list pane" of the editor.

Notes

1. The HTTP request to the server can be sent through various mechanisms, such as XMLHttpRequest object, Fetch or HttpClient object in Angular. If you decide to use `HttpClient`, you may need a basic understanding on the concept of "observable" (or event stream) and "observer" (or subscriber). If you feel that you need to learn more on this, there exist a number of good online introductory materials, such as observer pattern Wikipedia entry or this tutorial on reactive programming. If you decide to use `HttpClient` and `observable`, you are welcome to modify the BlogService API to use and return an observable, but this change is not necessary for this project.
2. When you modify `posts`, an array of `Post`, in the above methods, keep in mind that ***mutators, such as `splice()`, `push()`, and `shift()`, directly modify the input array, while accessor methods, such as `slice()`, `filter()`, and `map()`, don't. Accessor methods create a completely new copy of the output array, leaving the original input array intact.***
3. Note that any communication with a server is a potentially blocking function call and you will have to worry about asynchronous response handling in many methods of `BlogService`. This can be done in many different ways. You may implement the response handling logic inside BlogService itself and set the proper callback function internally. Alternatively, you may change the API of `BlogService`, so that it either takes a callback function as a parameter or it returns a promise/observable, so that the caller can decide what to do once the response is obtained from the server.
4. Since you have not implemented other components yet, it may be too early to implement proper error handling in case the server replies with an error response code. For now, you may want to skip error handing part, and come back to it only after you finish implementing other components.

Part E: Implement List and Edit Components

Roughly, our editor may be split into three different components:

1. *List component*: This component is responsible for displaying the "list pane". This component should be visible on the left side of the app all the time. The user should be able to click on a post in the list to edit it.
2. *Edit component*: This component is responsible for the "edit view" of the app. When the user clicks on a post in the list pane, this component should be displayed on the right side of the list and let the user edit the title and body of the post. It should also contain a buttons for "save", "delete", and "preview".

3. *Preview component*: This component is responsible for the "preview view" of the app. When the user clicks on the "preview" button in the edit component, this component should replace the edit component and show the HTML version of the post.

Add the List Component

We now create the list component.

```
$ ng generate component list
```

Note that all components in our app, including `ListComponent`, needs to use `BlogService` to retrieve and/or update blog posts. Thus, you will have to import `Post` and `BlogService` classes in `list.component.ts` through an import statement. Also, you will need to make `BlogService` available in `ListComponent` through dependency injection by modifying the component's constructor signature. If this sounds confusing, go over the Angular tutorial again, in particular the services section. The Angular documentation on dependency injection can also be helpful.

Note that the currently authenticated username can be obtained from the JWT token and the JWT token is stored as a cookie, which is accessible through `document.cookie`. To extract the username from JWT, you may use a code similar to the following, which will take a JWT a the input and returns a JavaScript object constructed from the payload of the JWT:

```
function parseJWT(token)
{
    let base64Url = token.split('.')[1];
    let base64 = base64Url.replace(/-/g, '+').replace(/_/g, '/');
    return JSON.parse(atob(base64));
}
```

Remember that the list component needs to take the following actions depending on the user interaction:

1. It has to obtain all blog posts through `BlogService` and display them in a list on the left side of the app.
2. When the user clicks on the "new" button, it has to create a new post using `BlogService` and open the "edit view" for the post.
3. When the user clicks on a post in the list, it has to open the "edit view" for the post.

To support the above interactions, remove the auto-generated HTML code in the template, `list.component.html`, and add necessary HTML elements. In modifying the template, you may find the following information useful:

- You may find the *structural directives* helpful in displaying the list of posts obtained from `BlogService` as a list.
- You can use *interpolation* (e.g., `{{post.title}}`) if you want to display a property value in the template.

- You can use *event binding* (e.g., `(click)="delete()"`), to call a method of the component for a triggered event.

Add CSS rules to `list.component.css` to make the component look reasonable.

Note that you can implement the second and third actions correctly only after you implement the Edit component. So start with implementing the first action, displaying all blog posts in a list, in `list.component.ts`.

Once you finish implementing code for the first action, you will need to add `ListComponent` as a child component of `AppComponent`, so that it will be displayed inside the Angular App. For example, you can add `<app-list></app-list>` to `src/app/app.component.html` as follows:

```
<h1>{{title}}</h1>
<app-list></app-list>
```

Testing the List Component

Eventually, your Angular app must be deployed to your node/express server, but during the development of this project, you may want to run your Angular app through the "ng serve" command, so that you can test, revise, and iterate quickly. Unfortunately, running Angular app through `ng serve` has a few unintended consequences:

1. You have to run two servers – the node/express server through `npm start` and the angular app through `ng serve --host 0.0.0.0` – within the same container. Running two servers simultaneously can be done by executing the two commands *in the background* like the following:

   ```
   $ npm start &
   $ ng serve --host 0.0.0.0 &
   ```

   If you are not familiar with the Unix process control and job management, read the Process section of our Unix tutorial.

2. Your Angular app is loaded from http://localhost:4200 but your Express server runs at http://localhost:3000. This means that your Angular app may not be able to use the JWT cookie set by the Express server due to cross-origin restrictions, which won't be the case once your Angular app is deployed to the Express server. To get around this problem, you have to ***allow third-party cookies*** and may use Chrome's `--disable-web-security --user-data-dir` option. To use this option, quit all running instances of your Chrome browser, and start a new instance of Chrome by executing:

   ```
   (On macOS)   $ open -a Google\ Chrome --args --disable-web-security --user-data-dir
   (On Windows) $ start chrome --disable-web-security --user-data-dir
   ```

   from a terminal.

3. Note that when your Angular app is eventually deployed to the Express server, the server will ensure that the user is authenticated before they can access the Angular app. Unfortunately, this is not the case when your app is loaded from its own server; when your Angular app runs, it may not have a proper JWT cookie during development. Therefore, you will have to explicitly visit the Express server's login page `/login` first, authenticate yourself, and obtain a proper JWT cookie from the server.

If you have the `ng serve --host 0.0.0.0` command running, you should now see that the `ListComponent` displays the list of blog posts in your MongoDB. If not, it is very likely that your code has a bug. Fix it before you move on to the next task.

Note: In case the `ng serve` does not auto-rebuild your app after a file is changed, make sure that you forwarded port 49152 when you initially set up the container. Otherwise auto-rebuild does not work. Some students also report that "ng serve" does not auto-rebuild even with proper port forwarding, particularly on a Windows machine. If that is the case, try `ng serve --host 0.0.0.0 --poll=2000`. The development server will look for file changes every two seconds and compile if necessary.

Add the Edit Component

Now let us create the second non-root component, the edit component:

```
$ ng generate component edit
```

Make `Post` and `BlogService` available in `EditComponent` by including an appropriate import statement and modify its constructor signature to inject `BlogService` through dependency injection. Also, add `post: Post` as a property of `EditComponent`, which will hold a copy of the post that is being currently edited.

Remember that `EditComponent` is responsible for the following user interactions:

1. The title and body of the current post should appear in `text` input and `textarea`, respectively, so that the user can edit them.
2. When the user clicks on the "preview" button, any unsaved changes should be saved, and the "preview view" should open.
3. When the user tries to leave the edit view, all unsaved changes should be automatically saved via `BlogService`, so that no work is lost.

To support the above interactions, remove the auto-generated HTML code in the template, `edit.component.html`, and add necessary HTML elements. In modifying the template, you may find the following information useful:

- Data can be dynamically exchanged between a template element and a component property using Angular's *two-way binding* and the `ngModel` directive (e.g., `[(ngModel)]="post.title"`). This mechanism can be used, for example, to support displaying and editing the post's title. Note that `FormsModule` needs to be imported in the *RootModule*, `app.module.ts`, if you want to use the `ngModel` directive.

- You can use the *structural directive* `*ngIf` or the *safe navigation operator* `?.` to guard against `null` or `undefined` values in a property.

Add CSS rules to `edit.component.css` to make the component look reasonable.

Once you finish updating the template and CSS style, implement the functionalities described above to the component class, `edit.component.ts`. In particular, you may want to add one "event handler" method per each button-click event with the names like `save()`, `delete()`, and `preview()`. Note that you are not able to implement `preview()` method yet, since you need to implement the preview component first to switch to the "preview view".

Part F: Add the AppRoutingModule

As we mentioned earlier, an important feature of the single-page application is that **a specific state of the application is associated with the corresponding url**. In particular, you need to associate the three "states" of our app with the following URL patterns:

URL	state
`/`	This default path shows only the list pane, without showing the edit or preview view
`/edit/:id`	This path shows the list pane and the "edit view" for the post with `postid=id`
`/preview/:id`	This path shows the list pane and the "preview view" of the post with `postid=id`

Associating a URL with a particular state of the application and displaying a different component based on the state can be achieved through a *router* in Angular. If you do not remember what a router is or how to use it, go over the Angular tutorial again, in particular the routing section. It may be useful to look at the routing & navigation section of the Angular documentation.

Angular's best practice is to load and configure the router in a separate, top-level module that is dedicated to routing and import it in the root AppModule. By convention, the class dedicated for a routing module is named as `AppRoutingModule` defined in the file `app-routing.module.ts` if you answered Yes to routing creating when you created the initial angular project. Check whether you have the file in `src/app` directory. If not, it can be generated using the following CLI command:

```
$ ng generate module app-routing --flat --module=app
```

The `--flat` option creates `app-routing.module.ts` directly in `src/app` not in its own subdirectory. The `--module=app` option adds necessary import statements to the *RootModule*, `app.module.ts`.

Now open `app-routing.module.ts` and update its content as follows:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { EditComponent } from './edit/edit.component';

const routes: Routes = [
  { path: 'edit/:id', component: EditComponent }
];

@NgModule({
  imports: [ RouterModule.forRoot(routes, {useHash: true}) ],
  exports: [ RouterModule ]
})
export class AppRoutingModule { }
```

In the above code, the `routes` config maps the URL pattern `edit/:id` to the `EditComponent`. (Mapping the the URL pattern `preview/:id` to the `PreviewComponent` need to be added later after we implement the "preview component". The path `/` does not need a mapping since the app does not need to display any component other than the list component for `/`.)

Add a router outlet in `app.component.html`, so an appropriate component will be displayed if the URL pattern matches to a route:

Now that we have the URL-to-component mapping in place through `RouterModule`, we need to modify our code to implement the correct routing behavior. In particular, we have to add the following logic to our code:

1. When the user clicks on a post in the list, our app should "navigate" to the URL `edit/:id`, where `:id` is the `postid` of the clicked post.
2. When the app navigates to `edit/:id` and the `RouterModule` displays `EditComponent` in the router outlet, our `EditComponent` should retrieve the post whose `postid` is `:id` and update its template elements accordingly.
3. When the user clicks the "delete" button in the `EditComponent`, the component should delete the current post through `BlogService` and navigate to the URL `/`, so that `EditComponent` is no longer displayed in the router outlet.
4. When the user "navigates away" from the "edit view", any unsaved work by the user should be automatically saved.
5. When the user clicks on the "preview" button in the `EditComponent`, the component should save any unsaved changes and navigate to the URL `preview/:id`, where `:id` is the `postid` of the current post.

To implement the above functionality, you may find the following information helpful:

1. Within your component method, you can "navigate" to a particular URL by calling the `navigate()` method of the Router object, like `router.navigate(['/'])`. Inside a template, you can use the `routerLink` directive, like `<a routerLink="/">`, to have the same effect when the user clicks on the link.

2. You can obtain the `:id` part of an “activated URL” (or “activated route”) with the ActivatedRoute object, by calling `activatedRoute.snapshot.paramMap.get('id')`.
3. The `EditComponent` can “subscribe” to the “URL activation event” – so that whenever a new URL is activated, it can retrieve the correct post from `BlogService` and update its elements – using the `ActivatedRoute` object. For example, the following code

   ```
   activatedRoute.paramMap.subscribe(() => this.getPost());
   ```

   will ensure `getPost()` method to be called whenever the route is activated. This mechanism can be used to save any unsaved user edits when the URL changes.
4. To save the user’s unsaved edits in the “edit view” when the user presses the browser’s refresh button or close the browser tab, you may want to intercept the “beforeunload” (or “unload”) event of the global `window` object. In Angular, a method can be declared to be called for a general browser event by decorating it with the `@HostListener(event)` decorator. For example, if you decorate a method of `EditComponent` with `@HostListener('window:beforeunload')`, the method will be called whenever “beforeunload” event is triggered to the global `window` object.

To be able to use `Router` and `ActivatedRoute` objects in `EditComponent`, update its constructor signature to make the two classes available through dependency injection.

If you use `@HostListener` decorator in `EditComponent`, you will need to import it in `edit.component.ts` by including the following import statement:

```
import { HostListener } from '@angular/core';
```

Now your web application should have all functionalities implemented except “preview”.

Part G: Add the Preview Component

You should be fairly familiar with the Angular development process if you finished all previous parts and reached here. In this part, most of the tasks are similar to what you have done already, so we provide a minimal guidance.

First you need to add the *preview* component through the Angular Cli. Then add the the route mapping from `preview/:id` to this component in your routing module.

For markdown to HTML rendering, we will use commonmark.js library, so install the `commonmark` module through the following command:

```
$ npm install --save commonmark
```

Open **preview.component.ts**, add the following import statement

```
import { Parser, HtmlRenderer } from 'commonmark';
```

to use commonmark's `Parser` and `HtmlRenderer` objects in your code.

Update the constructor signature of `PreviewComponent` with necessary dependencies and add the appropriate `import` statements. Subscribe to the URL activation event, so that the correct post is retrieved from `BlogService` and rendered as HTML when a "preview URL" is activated. Edit **preview.component.html** and **preview.component.css** to add the HTML elements and css rules needed for the component.

Finally, make sure that you update `EditComponent`, so that it correctly handles the user's click event on the "preview" button.

Congratulations! You now have finished all the requirements and functionalities of the project.

Part H: Deploy, Update, and Test your project

Once you finish implementing Project 4, build the final Angular application by the command `ng build` and copy all produced files inside `dist/angular-blog/` directory to the `editor` subdirectory of your Express server's public folder (`public/editor/`). Before you build your Angular app, make sure to modify the base URL of the app to `/editor/` by changing the base tag in the file `src/index.html`

```
<base href="/editor/">
```

Please make sure that if the user tries to access the Angular editor at the URL `/editor/` without authenticating herself first, the request is redirected to `/login?redirect=/editor/`, so that the user should be able to provide her authentication information and automatically come back to the editor. To implement this behavior, you will need to make a small change to the code in Project 3, where **your node server ensures that any request to `/editor/` or below contains a valid JWT. If not, the server should redirect the request to the login page like `/login?redirect=/editor/`.** Please note that no change is needed for the REST API handling or your Angular editor code. As long as the just described change is made, everything will work.

To simplify the project, you may assume that the first entry point to the blog post editor is always `/editor/`. That is, users never access the editor through the edit page `/editor/#/edit/:postid` or the preview page `/editor/#/preview/:postid` as the first entry point. Therefore, you may assume that the user's all blog posts are ready and available by the time either Edit or Preview Components are rendered.

Please check your web application is shown in the browser correctly and you can complete the post create, update, delete and preview tasks without any issues. Also, the post list should be always be displayed on the left side of any page and be updated whenever a new blog is created or an existing blog is updated or deleted. Make sure that all blog posts are corrected saved, updated, and deleted in MongoDB.

Lastly, please verify your application does not throw any errors in the javascript console. This can be checked in chrome by right clicking the browser window and choose “inspect”. Then choose the “Console” tab to verify if any errors are appearing.

Note: When you run your Angular app in Chrome, Chrome developer console will throw an error for every 4XX or 5XX response from a server. Unfortunately, there doesn’t seem to be any way to “catch” it to make it disappear. As long as you handle these responses according to our spec, you won’t have to worry about this particular Chrome console error.

Submit Your Project

What to Submit

For this project, you will have to submit ***two zip files***:

1. `project4.zip`: You need to create this zip file using the packaging script provided below. This file will contain all source codes that you wrote for Project 4.
2. `project3.zip`: You must resubmit `project3.zip` file again created using the packaging script of Project 3. This new submission must include any changes that you made during Project 4 development, including the code for the authentication protection for `/editor/` and bug fixes.

Creating `project4.zip`

After you have checked there is no issue with your project, you can package your work by running our packaging script. Please first create the *TEAM.txt* file and put your team’s uid(s) in it. This file must include the 9-digit university ID (UID) of every team member, **one UID per line. No spaces or dashes.** Just 9-digit UID per line. If you are working on your own, include just your UID. Please make sure **TEAM.txt and package.sh are placed in the project-root directory**, *angular-blog*, like this:

```
angular-blog
 +- e2e
 +- node_modules
 +- src
      +- app
           +- ...
      +- assets
      +- environments
      +- index.html
      +- styles.css
      +- typings.d.ts
      +- ...
 +- package.sh
 +- TEAM.txt
 +- ...
```

When you execute the packaging script like `./package.sh`, it will build a deployment version of your project and package it together with your source code and the *TEAM.txt* file into a single zip file named **project4.zip**. You will see something like this if the script succeeds:

```
[SUCCESS] Created '/home/cs144/shared/angular-blog/project4.zip', please submit it to CCLE.
```

Please only submit this script-created project3.zip and project4.zip to CCLE. Do not use any other ways to package or submit your work!

IMPORTANT: In case you use Docker Toolbox and have been using `192.168.X.X` as the hostname of the Project 3 server in your Angular code, it is critical **to change it to `localhost` in your submitted version. Otherwise, your submission is unlikely to work on the grader's machine!**

To ensure this, if the packaging script finds that your source code contains the string `192.168.X.X`, it displays the following warning message:

```
[WARNING] The script detected that your code has string 192.168.X.X
[WARNING] Please ensure that your submitted Angular code uses localhost as
[WARNING] the hostname of the server API, not 192.168.X.X
```

Please double check your code to ensure that your submitted code uses `localhost` not `192.168.X.X`. **Otherwise, your submission will not work on the grader's machine!**

Grading Criteria

We will grade your Project 4 mainly based on the functionalities. Specifically, we will test following things:

- **Post List**: Post list is always shown on the left side of any page, and is updated whenever a blog is created, updated or deleted.

- **Post Operations**: Create, Update and Delete blogs without any issues.
- **Post Preview**: Render the markdown blog correctly and be able to return to the edit view from the preview view.
- **User Interface**: The User Interface should at least adopt the similar CSS decorations like what is done in demo. **Better decorations may gain extra credit, but no more than 10%**.
- **No Errors**: JavaScript Errors are very severe issues, we will **deduct 20% of total points for each type of the error** that appears on the browser console, except the 4XX and 5XX errors that we mentioned in Part H.
- **Other Requirements in Description:** For example, auto-save behavior, etc.