Project 4: Markdown Editor Using Angular

Change History

1. 05/13/2020 1AM: Removed deprecated option `--module=app` from `ng generate` commands
2. 05/13/2020 2:30PM: Part H has been updated to include more detailed instruction on how authetication integration can be done
3. 05/14/2020 6:00AM: Provided a link to a tutorial to Angular testing tools, Jasmine and Karma.
4. 05/23/2020 6:15PM: Added `--deploy-url` option to the `ng build` command to make it work on Safari

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.

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! 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?
- 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!

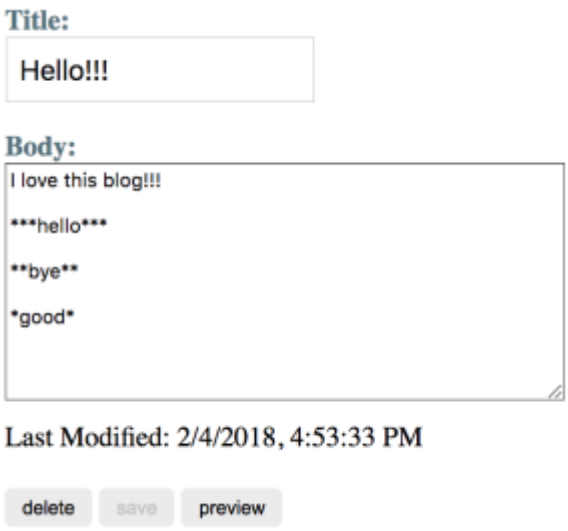
Part B: Project Demo and Requirements

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.

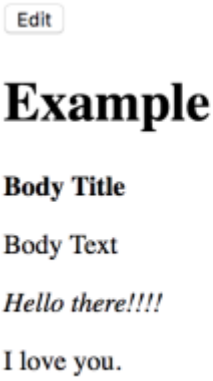
We made a demo website of Project 4 available at http://oak.cs.ucla.edu/classes/cs144/project4/demo/. It is rather simple, does not contain many CSS-styling instructions, does not actually store blog posts on the server, but it still helps you understand the key UI requirements of this project.

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:



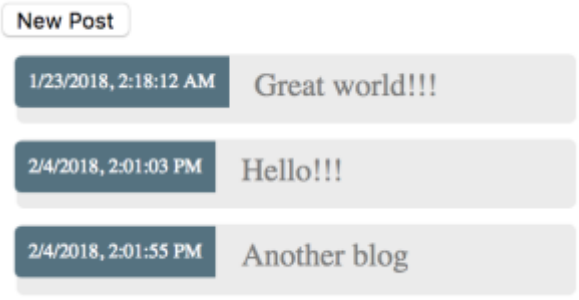
- The view shows one `text` input box for the title and one `textarea` for the body.
- The "last modified" date and time is shown below the text boxes.
- The view should contain at least three buttons, "save", "preview", and "delete".
- When the "save" button is pressed, it should permanently save the post in the server and update the last-modified date of the post to the current time.
- 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 from the server.

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



- 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:



- 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 title and the creation date of the post.
- The user can add a new post anytime by clicking the "new post" button, which **opens the edit view of a new empty post on the right side.**
- When a new post is created by the user its postid should be +1 of the largest existing postid 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 when the user pressses the "save" button, the corrsponding post must be stored/updated 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.

Finally, if you code includes any URL to access various functionalities of your site, make sure that the ***URLs do not include the hostname portion***, so that your code can be hosted on any domain without any change. For example, if you need send a request to `/api/cs144/1`, use the URL `/api/cs144/1` not `http://localhost:4200/api/cs144/1`.

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
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? (Use arrow keys)
› CSS
  SCSS   [ https://sass-lang.com/documentation/syntax#scss                ]
  Sass   [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
  Less   [ http://lesscss.org                                             ]
  Stylus [ http://stylus-lang.com                                         ]
```

Answer with `Y` for the question "Would you like to add Angular routing? (y/N)" and choose `CSS` for style sheet format. This may take a while since a lot of files are fetched and generated (≈ 500MB). When the skeleton application is created successfully, we will see the following output.

```
Packages installed successfully.
```

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

You will see the service files created in the terminal. The primary role of `BlogService` is to allow other components in the application to retrieve, update, and save blog posts via the REST API implmented in Project 3. It will also work as a "communication hub" between multiple components of your app, by temporaily storing and returning the current "draft" that is being edited by the user before it is permenently saved to the server.

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 you 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.

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

1. `fetchPosts(username: string): Promise<Post[]>` – This method sends an HTTP GET request to `/api/:username` and retrieves all blog posts by the user. If successful, the returned promise resolves to a `Post` array (of `Post[]` type) that contains the user's posts. In case of error, the promise is rejected to `Error(response_status_code)`.
2. `getPost(username: string, postid: number): Promise<Post>` – This method sends an HTTP GET request to `/api/:username/:postid` and retrieves the particular post. If successful, the returned promise resolves to a `Post` that corresponds to the retrieved post. In case of error, the promise is rejected to `Error(status_code)`.
3. `newPost(username: string, post: Post): Promise<void>` – This method sends an HTTP POST request to `/api/:username/:postid` to save the new post in the server. In case of error, the promise is rejected to `Error(status_code)`.
4. `updatePost(username: string, post: Post): Promise<void>` – This method sends an HTTP PUT request to `/api/:username/:postid` to update the corresponding post in the server. In case of error, the promise is rejected to `Error(status_code)`.
5. `deletePost(username: string, postid: number): Promise<void>` – This method sends an HTTP DELETE request to `/api/:username/:postid` to delete the corresponding post from the server. In case of error, the promise is rejected to `Error(status_code)`.

6. `setCurrentDraft(post: Post): void` - This method "saves" the post as the current "draft", so that it can be returned later when `getCurrentDraft()` is called.
7. `getCurrentDraft(): Post` - This method returns the draft saved in the earlier `setCurrentDraft()` call. Return `null` if `setCurrentDraft()` has never been called before.

Notes

1. The HTTP request to the server can be sent through various mechanisms, such as Fetch or HttpClient object in Angular. We recommend Fetch, but the decision is really up to you.
2. `setCurrentDraft()` and `getCurrentDraft()` should be performed as a local operation without any interaction with the Express server. You can add `draft: Post` property to `BlogService` and use it to save the passed post, for example.
3. Make sure that the ***URLs in your API requests do not include hostname***, so that your code can be hosted on any domain without any change. For example, you need to send a request to `/api/cs144/1`, not to `http://localhost:4200/api/cs144/1`.

Notes on Angular development server

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` in your Project 3 directory and the angular app through `ng serve --host 0.0.0.0` in your Project 4 direcotry – both within the same container. Running two servers simultaneously can be done by executing the two commands *in the background* like the following:

   ```
   // change to your Project 3 directory
   $ npm start &
   // change to your Project 4 directory
   $ 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. Because your browser will consider `localhost:4200` and `localhost:3000` two completely different web sites, if your Angular app sends an HTTP request to `localhost:3000` it will be considered as a cross-origin request, creating many unexpected problems. This will clearly not be an issue once you finish developing your Angular app and deploy to the Express server, but it is a nasty issue to deal with during development. To

get around this problem, you need to set up Angular CLI proxy, so that your app can send requests to `localhost:4200` not to `localhost:3000`. To learn how, read our Angular CLI proxy setup tutorial.

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 development server; during development, your app will not be "protected" behind the `/login` page and may not obtain a proper JWT cookie before it is served. To ensure that your browser obtains a proper JWT cookie from the server, manualy visit the login page http://localhost:4200/login and authenticate yourself as one of the two existing users before your load your app in your browser.
4. Due to an unknown reason, Angular development server sometimes behaves unexpectedly and throws errors when it clear that there should be no error. When this happens, stopping and restarting `ng serve` seems to fix the problem. Unexplainable errors may magically disappear when your restart `ng serve`.

Notes on Testing Code on Angular

Once you finish implementing `BlogService`, you may want to test its functionality. Unfortunately, this can be a bit hard because your `BlogService` is created and managed by the Angular Framework. To help developers to test their code, Angular has integrated two excellent testing tools, *Jasmine* and *Karma*. While learning them is not strictly necessary for this project, we strongly recommend going over this excellent tutorial on unit testing on Angular. It will take a few hours to go through, but you will be well rewarded especially if you decide to be a serious Angular developer. In fact, once you learn how to use these tools, you may wonder why you haven't picked up tools similar to these for other programming projects since they are extremely helpful for writing, managing, and running testing code. If you decide to use Jasmine and Karma for testing, don't forget to enable proxy for Karma as described in the Angular CLI proxy tutorial

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 as 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. When the app loads, 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 a post in the list, it sets the clicked post as the "current draft" by calling `setCurrentDraft(post)` of `BlogService`, and open the "edit view" on the right side.
3. When the user clicks on the "new" button, it has to create a new empty post whose `postid` is `+1` of the maximum `postids` of the user, sets it as the current draft by calling `setCurrentDraft()`, and opens the "edit view".

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.
- 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.

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>
```

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: Some students report that `ng serve` does not auto-rebuild, 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, and the “preview view” should open.
3. When the user clicks on the “save” button, and post should be updated/saved at the server.

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.
- In most cases, users will get to `EditComponent` by clicking on a post or the new button in the `ListComponent`, so the post is likely to be available through `getCurrentDraft()` of `BlogService`. If `getCurrentDraft()` returns `null` or if the returned post’s `postid` is different from the `id` in the URL (for various reasons), you need to obtain the post from the Express server by calling `getPost()` of `BlogService`. Exactly how and where to call `getCurrentDraft()` to obtain the current draft will be clearer when you implement the routing module in Part F. For now, you can just use a post with hard-coded initial values (like `{ postid: 1, title: "MyTitle", body: "MyBody" }`) as the current draft and test various functionalities of `EditComponent`.
- 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”.
- Pressing the “delete” button (and potentially “save” button as well) introduces changes to `ListComponent` component. This interaction crosses inter-component boundary and requires extra attention. Think carefully about how you can implement this interaction. Possibilities include expanding the `BlogService` API to facilitate this interaction or making `EditComponent` throw a custom event when the “delete” button is pressed using `EventEmitter`.

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`.

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.

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

1. Replace `RouterModule.forRoot(routes)` with `RouterModule.forRoot(routes, {useHash: true})`. The second parameter `{useHash: true}` ensures that the app routing path is encoded as *URL fragment identifier* behind a hash symbol.
2. Create a route that maps the URL pattern `edit/:id` to `EditComponent` by importing `./edit/edit.component`

   ```
   import { EditComponent } from './edit/edit.component';
   ```

   and adding the mapping to `routes`

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

   Note: 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 `/`.

After the above updates, your `app-routing.module.ts` will look similar to the following:

```
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 { }
```

Once these changes have been made, 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`, you need to modify your code to implement the correct routing behavior. In particular, you have to add the following logic to your code:

1. When the user clicks on a post in the list, your app should set the clicked post as the current draft by calling `setCurrentDraft(post)` of `BlogService` and "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, `EditComponent` should try to obtain the post to display through `getCurrentDraft()` of `BlogService`. If `getCurrentDraft()` returns `null` or if the returned post's `postid` is different from the `id` in the URL for various reasons, you need to obtain the post from the Express server by calling `getPost()` of `BlogService`.
3. When the user clicks the "delete" button in the `EditComponent`, the component should delete the current post at the server through `BlogService` and navigate to the URL `/`, so that `EditComponent` is no longer displayed in the router outlet.
4. When the user clicks on the "preview" button in the `EditComponent`, you should (locally) save the current (edited) draft by calling `setCurrentDraft(post)` of `BlogService` (so that `PreviewComponent`) can obtain the (edited) draft) and "navigate" to the URL `preview/:id`.

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 obtain the post to display 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.

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

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 commands:

```
$ npm install commonmark
$ npm install @types/commonmark
```

The second command installs the type definition file for the `commonmark` module to help TypeScript.

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 post to display can be obtained with `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. Make sure that you update `EditComponent`, so that it correctly handles the user's click event on the "preview" button.

Now you have finished all major requirements of Project 3 except the integration of user authentication. Before implementing this part, you need to deploy your Angular App to your Express server.

Part H: Deploy Your Angular App to Express Server

To deploy your Angular app to your Express Web server, take the following steps:

1. Build the production code of your Angular app by the command:

   ```
   $ ng build --base-href /editor/ --deploy-url /editor/ --prod=true
   ```

   Note the `--base-href` and `--deploy-url` options, which is required because our app will be deployed at the path `/editor/` not at the root `/`.

2. Once your production Angular code is built in the `dist/angular-blog/` directory, copy all files in the directory to the `editor` subdirectory of your Express server's public folder (i.e., `public/editor/`).

3. If it is not running, start your Express server by the command:

   ```
   $ npm start
   ```

4. Visit http://localhost:3000/login page of your Express server and authenticate yourself as `cs144`.

   Note: Notice that you are visiting `localhost:3000` not `localhost:4200`. You are accessing the Express server directly from your browser now, not through the Angular CLI proxy as you have done so far.

5. Load your Angular app by visiting http://localhost:3000/editor/ with your browser. If everything has been done correctly, your Angular app will be loaded from your Express server and start working. Again, note that the App is not coming from your `ng serve` development server at `localhost:4200`. In fact, the development server is no longer needed because everything is served by the Express server at this point.

6. Make sure that your Angular app functions correctly, ensuring that the post list is always displayed on the left side and "new", "save", "delete" and "preview" buttons work as intended.

Congratulations! You have successfully "deployed" your Angular app to the Express server.

Part I: Integrate User Authentication

Now that your Angular app is successfully deployed, your final task is to integrate user authentication with your Angular app. This can be done by adding the following logic to your Express server code:

In short, you have to add the code to your Express server that ensures that any request to `/editor/` contains a valid JWT. If not, responds with a redirect to `/login?redirect=/editor/`. **Remember that this change is all you need and no change is needed within your Angular App**. As long as this change is made to your Express server, everything will work because of the following reason:

When the Express server gets a request at the path `/editor/`, it checks whether the request contains a valid JWT cookie. If yes, everything is fine because the user has already authenticated themselves. If not, it "forces" the user to authenticate themselves by responding with a redirect to `/login?redirect=/editor/`. When this response is received, the browser will redirect to the `/login?redirect=/editor/` page, letting the user login. If the user provides correct authentication information to `/login?redirect=/editor/` via the HTML form, your Express server responds with another redirect to `/editor/` due to the optional parameter `redirect=/editor/`. This time, the request to `/editor/` sent by the browser will be completed successfully because a valid JWT cookie is included.

Now you are all done! Please verify that all functionalities of your application work by accessing your Angular app deployed at the Express server, http://localhost:3000/editor/. Make sure that it does not throw any errors in the javascript console, maybe except due to 4XX or 5XX responses from a server. 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.

Submit Your Project

What to Submit

For this project, you will have to submit ***two zip files and one demo video***:

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 your Angular production code placed in `public/editor/`, the new authentication-integration code, and bug fixes.
3. Demo video: To demonstrate that you have a web site that works according to our spec, you need to record and submit a "demo" video. The length of your video should be maximum three minutes, demonstrating the basic functionality and interaction of your site. More details on the video will be described shortly.

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!

Recording Demo Video

The length of your demo video should be maximum three minutes. In your video, demonstrate the following functionalities exactly in the given sequence:

- Redirect (Total 15 points)
  - If you try to access `/editor/#/` without authentication, you'll be redirected to login page (5)
  - Once you login as user `cs144` (same as project3), you'll be redirected back to `/editor/#/` (5)

    - Show that `/editor/#/` displays all existing posts of user `cs144` (5)
- Post operations (Total 50 points)
    - Able to create/update/delete posts and verify the posts have been created/updated/deleted in database (30)
    - Show that Post List always shown on left side of page and is updated whenever post is created/updated/deleted (10)
- Post Preview (Total 20 points)
    - Able to render markdown blog correctly (10)
    - Able to return to edit view from preview view (10)
- Back Button (10 points)
    - Show that the browser back button changes to previous state of the same app
- Deep Link (10 points)
    - Show that an appropriate page is displayed if the user directly types deep links like `/editor/#/edit/1` and `/editor/#/preview/1` in the address bar.
- User Interface (5 points)
    - Have at least basic CSS like demo

There exist a number of excellent free/paid software for recording your computer screen while you demonstrate the functionality of your Web site. For example, OBS Studio is a free open-source software that is widely using for screen capturing and live video streaming. There exist many online OBS tutorials that teach you how to use OBS to record your computer screen. Windows and Mac also have built-in utilities that can be used for this purpose, such as Xbox Game Bar (for Windows) and QuickTime Player (for Mac). Whatever software you use, make sure that your video is playable with the latest version of VLC Media Player without installing any proprietary codec to avoid any codec/format incompatibility issue. Recording your video using the H.264/WebM codec in the MP4/MOV/MKV container format will be a safe choice.

Grading Criteria

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

- Redirect (Total 15 points)
    - If you try to access `/editor/#/` without authentication, you'll be redirected to login page (5)
    - Once you login as user `cs144` (same as project3), you'll be redirected back to `/editor/#/` (5)
    - Show that `/editor/#/` displays all existing posts of user `cs144` (5)
- Post operations (Total 50 points)

    - Able to create/update/delete posts and verify the posts have been created/updated/deleted in database (30)
    - Show that Post List always shown on left side of page and is updated whenever post is created/updated/deleted (10)
- Post Preview (Total 20 points)
    - Able to render markdown blog correctly (10)
    - Able to return to edit view from preview view (10)
- Back Button (10 points)
    - Show that the browser back button changes to previous state of the same app
- Deep Link (10 points)
    - Show that an appropriate page is displayed if the user directly types deep links like `/editor/#/edit/1` and `/editor/#/preview/1` in the address bar.
- User Interface (5 points)
    - Have at least basic CSS like demo