Project 4: Markdown Editor Using Angular

Change History

05/17/2021 1:00PM

We now requires specific `id` attributes values for various HTML elements in your Angular App. In particular, we are requiring the following:

1. The "New Post" button in the List component should have the `id` attribute `list-new-button`.
2. Each blog post entry appearing in the List component should be in an HTML element with `id="list-post-:postid"` where `:postid` is the `postid` of the post.
3. The "Save", "Delete", and "Preview" buttons in the Edit component must have `id` attributes, `edit-save-button`, `edit-delete-button` and `edit-preview-button`, respectively.
4. The title and body input boxes/text areas in the Edit component must have `id` attributes, `edit-title` and `edit-body`, respectively.
5. The "Edit" button in the Preview component must have `id="preview-edit-button"`.
6. The rendered title and body in the Preview component must appear in HTML elements with the `ids`, `preview-title` and `preview-body`, respectively.

Because your code will be autograded now, you are no longer required to submit a video that demonstrates the functionality of your code.

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 the 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 started 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
```

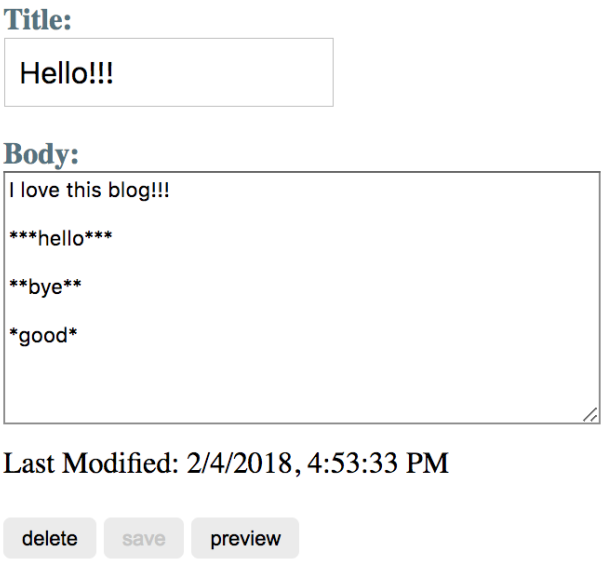
Part A: Project Demo and Requirements

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 an 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 the browser back button. When a user presses the 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.

Before we describe the detailed requirements for Project 4, we encourage you to visit our Project 4 demo website at http://oak.cs.ucla.edu/classes/cs144/demos/project4/. It is rather simple, does not contain many CSS-styling instructions, does not actually store blog posts on the server, but it will help you understand the basic UI interactions to be implemented for Project 4.

Now here is more detailed spec for Project 4. In the first image on the right, we show the **edit view** of the application, which allows the user to edit a post. In this view, we require you to implement the following functionalities:

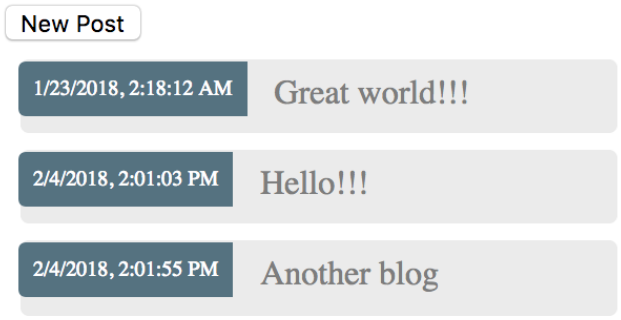


- The view shows two separate input box/textarea for the title and the body of a blog post with `id` attribute `edit-title` and `edit-body`, respectively.
- The "last modified" date and time of the post is shown below the input areas.
- The view should contain at least three buttons, "save", "preview", and "delete" with the `id` attribute `edit-save-button`, `edit-preview-button`, and `edit-delete-button`, respectively.
- When the "save" button is pressed, the post should be permanently saved/updated in the server and be added to the "list pane" (described below) if it is a new post. The last-modified date of the post must be updated 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 be permanently deleted from the server and disappear from the "list pane" (described below).

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 being edited by the user. In particular, the title and the body of the post should appear in an HTML elements whose `id` attributes are `preview-title` and `preview-body`, respectively.
- There is an "edit" button with `id` attribute `preview-edit-button` that allows switching back 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. Each blog post entry should appear in an HTML element whose `id` attribute is `list-post-:postid`, where `:postid` is the `postid` of the corresponding post.
- The posts in the list should be sorted by their `postid` in the ascending order.
- Each post in the list must show the title and the creation date of the post.
- The user can start writing a new post anytime by clicking the "new post" button whose `id` attribute is `list-new-button`. Clicking the button should **open the edit view of a new empty post on the right side.**
- 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 an SPA and **the list pane should be always visible on the left side**.

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, pressed the "preview" button, and clicks on the browser back button, the user should go to the "edit view" of the clicked post. 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" of the post with `postid=:id`. `:id=0` means the edit view for a "new post."
`/editor/#/preview/:id`	This path shows the list pane and the "preview view" of the post with `postid=:id`

Note that blog posts stored in MongoDB of your Project 3 are updated/saved/deleted only when the user presses the "save" or "delete" button in the edit view. Also, the HTML rendering of a blog post displayed in the preview page should be generated by a JavaScript code running inside the browser, using the commonmark.js library. In addition, **if the user tries to access the**

Angular editor at the URL `/editor/` without authenticating themselves first, the request must be redirected to `/login?redirect=/editor/` with the status code 302, so that the user should authenticate themselves first and automatically come back to the editor. This will ensure that when the client-side Angular app 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 implemented in Project 3. Implementing this redirection-for-authentication mechanism will require minor changes to the server-side code of Project 3.

Here are a few additional technical requirements for your implementation.

1. Your application **must not to use an alert or dialog box** to prompt a message to the user (e.g., when there is an error). If you want to deliver a message to the user, display it inside the main page, not in an alert box.
2. If your 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/posts`, use the URL `/api/posts` not `http://localhost:3000/api/posts`.
3. The changes from any interactions with your App, including the initial loading, should be **completed in less than 1 second**. Since your Express server and your browser runs on the same physical hardware, there is close-to-zero network delay. Therefore, meeting this 1-second requirement is trivial and is unlikely to require an explicit optimization from your side. But be aware of this requirement and make sure that your App is reasonably responsive.
4. Your submission will be tested using a Firefox browser. When your code has a bug, your App may exhibit an erroneous behavior only under a particular browser and/or interaction timing. If you are surprised with the output from our autograder, you may want to test your code using the Firefox browser and see anything surprising pops up there.

In the rest of the project spec, we describe more detailed guidance on how you can implement 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 all requirements in Part A, you can implement your application however you want. We provide further description here in case you need more guidance and help to finish this project.

Notes on Chrome Developer Console

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 B: 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 are 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. 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**. Please go over the tutorial at least from *Introduction* through the *Add Services* sections.

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.

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.

Notes on auto-rebuild: Some students report that `ng serve` does not auto-rebuild your app, particularly when your files are located somewhere below `~/shared/`. If that is the case, either move your files out of `~/shared/` to some other location or 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.

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, ignore previous AngularJS versions and just learn the new 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 can Dependency Injection be used in Angular?
- What are commonly used Angular CLI commands, such as generating a component or a service?

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

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

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 skeleton Angular application using the following command:

```
$ ng new angular-blog
? Do you want to enforce stricter type checking and stricter bundle budgets in the
  This setting helps improve maintainability and catch bugs ahead of time.
  For more information, see https://angular.io/strict No
? Would you like to add Angular routing? No
? 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 `No` for the first two question 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 code is created successfully, you will see the following output.

```
Packages installed successfully.
```

In case you encounter an error during the code generation, please see Notes on Project Directory.

The `angular-blog` directory contains the initial skeleton code and 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.

Notes on Project Directory: For Project 4, we strongly recommend developing your code outside of the shared directory, say at `~/project4/angular-blog/`, not within `~/shared/`. Students experienced a whole set of issues in the past when the project code was placed within the shared directory, starting from an error during the generation of the skeleton code (!). This setup, unfortunately, makes it a bit hard to use your favorite editor in your host machine to edit your code. A possible workaround could be to copy all your files in `~/project4/angular-blog/src/app/` into a shared folder, say `~/shared/project4`, by a command like:

```
$ cp -r ~/project4/angular-blog/src/app/* ~/shared/project4/
```

and use your host editor to edit the code in `~/shared/project4`. Once you are done with a batch of edits, you can then "synchronize" your main code in the container with your edits in the shared folder by copying back the edited files like:

```
$ cp -r ~/shared/project4/* ~/project4/angular-blog/src/app/
```

Since you will have to execute the above command many times during development, it may be a good idea to create an alias, like:

```
$ alias ccc="cp -r ~/shared/project4/* ~/project4/angular-blog/src/app/"
```

so that you can simply type `ccc` to copy your edited files back to the main project directory.

Serving and Accessing Generated App

Once the skeleton code is generated, you can compile and serve the generated app using "ng serve"

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

and access it with your browser at http://localhost:4200/. **Do not forget `--host 0.0.0.0` as we are serving the Angular app from the container.**

When you launch and access the app, 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 triggers an event based on that input. In the `src/app` folder, the Angular CLI has created the four files that are responsible for the `AppComponent`:

- `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.
- `app.component.spec.ts`: the ".spec.ts" file is used for unit testing, and you can ignore it for now.

Refer to the Angular components section of the tutorial if you are not sure how these files work together to form a component.

Part D: Implement UI Components

Roughly, our editor will be split into three components and one service:

1. *List component*: This component is responsible for the UI interaction of 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. It should also contain a "new post" button to start editing a new post.

2. *Edit component*: This component is responsible for the UI interaction of 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 buttons for "save," "delete," and "preview."
3. *Preview component*: This component is responsible for the UI interaction of the "preview view." 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. This component also contains an "edit" button to let the user go back to the edit view.
4. *Blog service*: This service provides the abstraction to the REST API to allow our app to access the Express MongoDB server implemented in Project 3.

We will first start with implementing the three components responsible for the UI interaction, but before we do that, we need to define the `Post` class, which is the key data structure to model the user's blog posts.

Add `Post` class

The `Post` class is the key data model representing a user's blog post:

```
export class Post {
    postid: number = 0;
    created: number = 0;
    modified: number = 0;
    title: string = "";
    body: string = "";
};
```

Note the `export` keyword since it will be imported and used by all components of our app. `postid` is the unique id of the blog post, `created` and `modified` are the post's creation and last modification time represented as milliseconds since the UNIX epoch, and `title` and `body` are the actual content of the post in markdown.

Create the file `post.ts` in the `angular-blog/src/app` folder with the above content, so that it can be imported and used by the components that we will implement soon.

Implement List Component

As the first basic UI element, we create the list component. First, generate its skeleton code using Angular CLI:

```
$ ng generate component list
CREATE src/app/list/list.component.css (0 bytes)
CREATE src/app/list/list.component.html (19 bytes)
CREATE src/app/list/list.component.spec.ts (614 bytes)
CREATE src/app/list/list.component.ts (267 bytes)
UPDATE src/app/app.module.ts (540 bytes)
```

Remember that the list component is responsible for three types of user interactions:

1. It displays all blog posts by the user.
2. It allows the user to "click" on a displayed post, so that they can start editing it.
3. It displays a "New Post" button, so that the user can start writing a new post.

Note that components are "dumb" UI elements that do not know or care the high-level semantics of the app. They simply display what they are asked to display and notify others of any events triggered by the user actions on them.

Given the three user interactions described above, we see that iteraction 1 requires the list component to be able to "take an input" from "outside" to get "posts to display." Interactions 2 and 3 require the list component to be able to "trigger events" and notify others that a user action has been taken. To support these three interactions, `ListComponent` will have one "input property", named `posts`, and two "output events", named `openPost` and `newPost`. That is, anyone should be able to create and use `ListComponent` by adding the following directive in the template:

```
<app-list [posts]="postsToDisplay"
          (openPost)="openPostHandler($event);"
          (newPost)="newPostHandler();"></app-list>
```

Note that there are three data bindings inside the `<app-list>` directive, one input property binding and two output event bindings. In the above directive, `ListComponent` takes `postsToDisplay` (whose type should be `Post[]`) as an "input", which is bound to its `posts` property. In addition, the directive will call `openPostHandler($event)` and `newPostHandler()` functions when `openPost` and `newPost` events are triggered by `ListComponent`, respectively. The `$event` object passed from the `openPost` event should be a `Post` object representing the post

clicked by the user. The `newPost` event does not pass any event object. If our description here sounds cryptic, please go over class notes on Angular and Angular documentation on *property binding* and *event binding*.

Now, add the three properties — `posts`, `openPost` and `newPost` — to the `ListComponent` class to support the above property and event bindings. The following information will be helpful in adding the properties:

1. To use the `Post` class, you will have to import it in `list.component.ts` like the following:

```
import { Post } from '../post';
```

2. To allow property binding to a component class property, you need decorate the property with @Input(). Import the decorator to use it in your code:

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

3. To trigger a custom event from a property, you need to decorate it with @Output() and create and assign a EventEmitter. Once created, you can simply call emit(obj) on such a property to trigger a custom event with the property's name and pass `obj` as the `$event` object. Import `@Output` decorator and `EventEmitter` class to use them in your code:

```
import { Output, EventEmitter } from '@angular/core';
```

If you are not clear about how to use @Input(), @Output() and EventEmitter, please go over the linked Angular documentation.

Once you finish adding the three properties to the component class, update your component template file `list.component.html`. Remove the auto-generated HTML code and add appropriate HTML elements and event bindings to display `posts` as a list and to trigger `openPost` and `newPost` events given corresponding user actions. 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 data in a component class property in the template.
- You can use *event binding* (e.g., `(click)="delete();"`), to call a component class method in your template when an event triggered on an HTML element.

- You can create a JavaScript `Date` object from `post.created` like `new Date(post.created)`. This may be helpful in converting the time to a string representation.

Once you finish modifying the template and the component class, thoroughly test its functionality by adding it as a child component of `AppComponent`. Pass a list of "fake" posts to `ListComponent` through property binding and take "fake" actions (like printing a message in Chrome Developer Console) when the custom events are triggered to ensure that `ListComponent` behaves as expected.

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

Notes on Testing on Angular Development Server:

1. Again, you can compile and serve the Angular app through "`ng serve --host 0.0.0.0`" command and access it from your browser at http://localhost:4200/. If your `ng serve` does not auto-rebuild, try "`ng serve --host 0.0.0.0 --poll=2000`".
2. Due to an unknown reason, `ng serve` sometimes behaves unexpectedly and throws errors when it shouldn't. When this happens, stopping and restarting `ng serve` seems to fix the problem.
3. Note that under `ng serve` your "Angular editor app" is accessible at http://localhost:4200/ not at http://localhost:4200/editor/ as required in Part A. You don't need to worry about this for now. When you build your final app in Part H, you will add the option `--deploy-url=/editor/ --base-href=/editor/` to make your app available at `/editor/`.

Notes on Unit Testing:

Whenever you update your code, you may want to test the functionality of individual part to ensure that your change does not break anything. To help developers "unit test" their code, Angular has integrated two excellent tools, *Jasmine* and *Karma*. While learning them is not necessary for this project, this excellent tutorial on unit testing on Angular explains how to use these tools. 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. Unfortunately, to reduce its footprint, our container does not have the web browser necessary for unit testing Angular components. If you want to test out Jasmine and Karma as you follow along the tutorial, you will need to install Angular on your host machine yourself and use that version, not the one in our container.

Implement Edit Component

Now let us create the second component, the edit component:

```
$ ng generate component edit
```

Recall that `EditComponent` is used to support the following user interactions:

1. The user should be able to edit the title and body of the post.
2. When the user clicks on the "save" button, the post should be updated/saved at the server.
3. When the user clicks on the "delete" button, the post should be deleted from the server and the displayed list.
4. When the user clicks on the "preview" button, the "preview view" should open.

Item 1 requires "taking an input" from outside, since the component has to obtain the current content of the post to edit. Items 2 through 4 require "notifying user actions" to outside of the component. Thus, `EditComponent` will have one input property `post` and three output events, `savePost`, `deletePost` and `previewPost`. That is, the component may be used with the following directive:

```
<app-edit [post]="postToEdit"
          (savePost)="savePostHandler($event);"
          (deletePost)="deletePostHandler($event);"
          (previewPost)="previewPostHandler($event);"></app-edit>
```

The `post` input property binds to a `Post` value/variable. The three output events pass the `Post` object (the post that has been edited by the user) as the `$event` object.

Add the four properties — `post`, `savePost`, `deletePost` and `previewPost` — to the `EditComponent` class to support the above property and event bindings. Once you finish adding the four properties, update your component template file to support the UI interaction.

Once you finish modifying the template and component class, thoroughly test its functionality by adding it as a child component of `AppComponent`.

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

Implement Preview Component

The `PreviewComponent` renders a post in HTML. It also has an "edit" button to let the user go back to the "edit view." Thus, the `PreviewComponent` will have one input property and one output event and can be used with a directive like the following:

```
<app-preview [post]="postToDisplay"
             (editPost)="editPostHandler($event);"></app-preview>
```

Here, `postToDisplay` is a `Post` object to be displayed by the component and the `editPost` event passes the displayed `Post` object as the `$event` object. Create the preview component through the Angular CLI, modify its component class, template, and CSS files to support the above usage.

For markdown to HTML rendering, we will use commonmark.js library. Install the `commonmark` module through the following commands:

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

The second command installs the type definition file used by the TypeScript compiler. Once installed, you can use its `Parser` and `HtmlRenderer` by including the following import statement:

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

After you finish developing `PreviewComponent`, test it thoroughly by including it in the `AppComponent` template and binding to fake posts and event handlers.

Congratulations! You have successfully finished implementing all basic UI elements of our app.

Part E: Connect the Dots

Now that we have basic UI elements ready, let us "connect" them to support the user interaction described in Part A. At a high level, this step requires two things:

1. **Maintaining application states**: Any running app needs to maintain "local states" to represent the user data and keep track of the exact state of the app after many user interactions. Roughly our app needs to maintain three key states:
    1. *User's post list* (`posts: Post[]`): The app has to keep track of the user's blog posts, so that the list component can display them and let users click on them.
    2. *The current post being edited/previewed* (`currentPost: Post`): The app also has to keep track of the current post being edited/previewed and hold any unsaved edits made by the user before they are permanently saved in the server.
    3. *The current state of the app* (`appState: enum { List, Edit, Preview }`): At any point of the time, the app is in one of three states: (1) `List`: In this state, the app only

displays the list component on the left (2) `Edit`: In this state, the app displays the list component on the left and the edit component on the right (3) `Preview`: In this state, the app displays the list component on the left and the preview component on the right.

2. **Connecting events to actions**: As the app goes through its lifecycle via user-driven "events," it has to populate and update the above states to reflect what has happened so far. Roughly, the following events are critical milestones in the app's lifecycle that lead to significant changes in its states.
    1. *Initialization*: When the app is loaded, the app has to obtain the user's blog posts from the server to populate `posts` and pass them to the list component. The app always starts in the `List` state (i.e., `appState = List`).
    2. *Clicking on a post*: When the user clicks one of the posts in the list, the app should set `currentPost` to the clicked post. The `appState` changes to `Edit`.
    3. *Clicking on the new button*: When the user clicks on the "new post" button, the app should set `currentPost` to a new empty post. The `appState` changes to `Edit`.
    4. *Editing the current post*: As the user edits the title and the body of the current post, the content of `currentPost` should be updated.
    5. *Clicking the save button*: When the user clicks on the save button, changes in `currentPost` should be saved in the server and updated in local `posts` list.
    6. *Clicking the delete button*: When the user clicks on the delete button, the current post should be deleted both from the MongoDB server and the local `posts` list. The `appState` changes to `List`.
    7. *Clicking the preview and edit buttons*: When the user clicks on the preview button in the edit component or the edit button in the preview component, `appState` changes between `Edit` and `Preview`.

In our implementation, we will make the `AppComponent`, the root component of our app, be responsible for maintaining the key states of our app and taking the appropriate actions when key events are triggered. Before we explain how, we need to briefly talk about `BlogService`.

Fake `BlogService`

The primary role of `BlogService` is to allow our Angular app to retrieve, update, and save blog posts at a remote server via the REST API implemented in Project 3. Now create the `BlogService` skeleton code 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)
```

Later in Part F, you will implement the real `BlogService` to integrate our app with the back-end server. In the meantime, to help you continue developing the app without worrying about all complex issues arising from the back-end server integration, we provide a blog.service.ts file that implements a "fake" `BlogService` that behaves almost like a real `BlogService`, but in reality does everything locally within the browser (using `localStorage`). You don't have to worry about understanding the provided code. It is just a temporary patch to help you continue development. Simply download the file and replace the generated `src/app/blog.service.ts` with it.

The key functions that `BlogService` provides are the following:

1. `fetchPosts(username: string): Promise<Post[]>` – This method returns a promise that is resolved to 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 an error, the promise is rejected to `Error(response.status)`.
2. `getPost(username: string, postid: number): Promise<Post>` – This method returns a promise that is resolved to the retrieved post. In case of an error, the promise is rejected to `Error(response.status)`.
3. `setPost(username: string, post: Post): Promise<Post>` – This method either inserts a new post (if `post.postid=0`) or updates an existing one (if `post.postid>0`). If successful, the returned promise is resolved to a `Post` whose `modified` field has the value returned from the server (and `postid` and `created` fields as well in case of insertion). In case of an error, the promise is rejected to `Error(response.status)`.
4. `deletePost(username: string, postid: number): Promise<void>` – This method deletes the corresponding blog post. In case of an error, the promise is rejected to `Error(response.status)`.

Implement `AppComponent`

We now add the local states and the appropriate event handlers to `app.component.ts` and connect them to the three components that we implemented in Part D and the fake `BlogService` that we just downloaded.

As an initial set up, do the following to `app.component.ts`:

1. Add the necessary import statements to use `Post` class and `BlogService` classes

```
import { Post } from './post';
import { BlogService } from './blog.service';
```

2. Create an enumeration type `AppState`, so that we can use it as the type for the `appState` property that we will add soon:

```
enum AppState { List, Edit, Preview };
```

3. Make `BlogService` available in `AppComponent` through dependency injection by modifying its constructor signature:

```
class AppComponent {
    ...
    constructor(private blogService: BlogService) {}
    ...
}
```

   If this sounds confusing, go over the the services section of Angular tutorial again. The Angular documentation on dependency injection can also be helpful.

Now that the basic preparation is done, add the following three properties and six event handlers to `AppComponent`:

```
class AppComponent {
    ...
    posts: Post[];
    currentPost: Post;
    appState: AppState;
    ...
    // event handlers for list component events
    openPost(post: Post) {}
    newPost() {}
    // event handlers for edit component events
    previewPost(post: Post) {}
    savePost(post: Post) {}
    deletePost(post: Post) {}
    // event handlers for preview component events
    editPost(post: Post) {}
    ...
}
```

Hopefully, the meaning of the above properties and methods are clear from our earlier discussion. Bind the above states and event handlers to the appropriate properties and events of the three child components of `AppComponent` by replacing the content of `app.component.html` with the following:

```
<app-list [posts]="posts"
          (openPost)="openPost($event);"
          (newPost)="newPost();"></app-list>
<app-edit [post]="currentPost"
          (previewPost)="previewPost($event);"
          (savePost)="savePost($event);"
          (deletePost)="deletePost($event);"></app-edit>
<app-preview [post]="currentPost"
             (editPost)="editPost($event);"></app-preview>
```

Add the structural directive to each child component as needed, so that only relevant components are displayed in a particular app state. For example, the edit component should appear only if `appState == AppState.Edit`. Keep in mind that the `enum AppState` type is not directly useable in the template, so you may want to add simple helper functions in the component class that compares the current `appState` against a particular value and call this helper function from your template.

Now write the code for the (currently empty) event handlers to implement the actions and state transitions that we discussed earlier. For this implementation, you may find the following information helpful:

1. When the application starts, it has to obtain the list of the user's list of blog posts from the server. This can be done by calling `fetchPosts()` of `BlogService` either in the `AppComponent` constructor or in its `ngOnInit()` lifecycle hook. We recommend using the constructor, just because you have to add `import { OnInit } from '@angular/core';` statement and add `implements OnInit` to the `AppComponent` definition in order to add `ngOnInit()`. (Differently from other components, this is not automatically done to `AppComponent` by Angular CLI.)
2. Our fake `BlogService` completely ignores passed `username` and returns the same set of posts for any `username`. You will need to fix this behavior later in Part F, but it is good enough for now. When you call any method of `BlogService`, you can use any username, like `cs144`, for now.
3. The `openPost()` and `newPost()` event handlers should set `currentPost` to an appropriate post. In case of `newPost()`, you will have to create a new empty `Post` with `postid=0`. Recall that our REST API (and the `setPost()` method of `BlogService` in turn) saves a post as a *new* entry when `postid=0`.
4. The `savePost()` and `deletePost()` event handlers must ensure that the changes are reflected in the local state `posts` array as well. Remember that the new `postid`, `modified` and `created` values of the saved/updated post can be obtained from the resolved value of the promise returned from the `BlogService` function call.
5. If you are not sure how to use the `Promise` returned by `BlogService` methods, go over class notes on SPA, in particular, the slides on Fetch API.

Implement all event handlers and make sure that they all work correctly together and support the user interactions described in Part A.

Once you thoroughly test your implementation for this part, add CSS rules to `app.component.css` and extra HTML elements to `app.component.html`, so that your app looks reasonable in every state.

Part F: Integrate with the Back-End Server

Now that you have an app that works well locally, it is time to integrate it with the back-end server from Project 3.

Running Two Servers

To connect your angular app with the back-end server, you now need to run your Express server from Project 3. Eventually, your Angular app will be deployed to your Express server as well, so both your Express back-end and your Angular front-end will be served by a single server. But during development, you may want to serve your Angular app separately through the "ng serve" command, so that you can test, revise, and iterate quickly. Unfortunately, serving Angular app through a separate `ng serve` server has a few unintended consequences:

1. You have to run two servers – the Express server from Project 3 and the Angular app server through `ng serve --host 0.0.0.0` – in the same container. Running two servers simultaneously can be done by executing them *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 &
   ```

   Note the ampersands at the end, which execute the commands in the background. 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` as 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. This creates nasty CORS problems, which will not be an issue later after you deploy your Angular app to the Express server, but is an issue during development. To get around this problem, you need to set up an Angular CLI proxy, so that your app can send requests not to `localhost:3000` but to `localhost:4200` from which the app is loaded. To learn how, read our Angular CLI proxy setup tutorial.

Obtain Username from JWT Cookie

Eventually when we deploy our Angular app to the Express server in Parts H and I, we will implement the code to ensure that the user is authenticated before they can access the Angular app. Unfortunately, this is not the case for now. You need to ***manually*** visit the page http://localhost:4200/login with your browser and authenticate yourself to obtain a valid JWT cookie from the server. Note that JWT cookie is necessary for your app to communicate with the Express server through the REST API successfully. Otherwise, your Express server will reject any API request. So please visit http://localhost:4200/login and authenticate yourself as one of the two existing users now before your forget. And remember to do it again whenever you restart your browser.

The `BlogService` API requires the authenticated user's name as an input parameter, which can be obtained from the payload of the JWT cookie. To parse cookie values, first install the cookie module:

```
$ npm install cookie
$ npm install @types/cookie
```

and add the following import statement to `app.component.ts`

```
import * as cookie from 'cookie';
```

The browser's cookie values are accessible at document.cookie. You can use the function `cookie.parse()` to parse `document.cookie` into cookie name value pairs:

```
let cookies = cookie.parse(document.cookie);
```

Go over the cookie module documentation to learn the detail.

Once you obtain the JWT token from the cookie, you may use a code similar to the following to convert its payload to a JSON object:

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

Recall that the authenticated username appears in the `usr` field of the JWT payload. Modify the code in `app.component.ts` so that (1) the proper username from JWT is obtained and (2) the obtained username is used whenever a `BlogService` method is called.

Implement New `BlogService`

Now remove the existing code in the `BlogService` class and reimplement it so that it actually communicates with the real Express server. The class should provide the following four key methods:

1. `fetchPosts(username: string): Promise<Post[]>` – This method sends an HTTP GET request to `/api/posts?username=: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 an error, the promise is rejected to an `Error` object `new Error(String(response.status))`.
2. `getPost(username: string, postid: number): Promise<Post>` – This method sends an HTTP GET request to `/api/posts?username=:username&postid=:postid` and retrieves the post. If successful, the returned promise resolves to a `Post` that corresponds to the retrieved post. In case of an error, the promise is rejected to `Error(String(response.status))`.
3. `setPost(username: string, post: Post): Promise<Post>` – This method sends an HTTP POST request to `/api/posts` with the corresponding JSON body, so that the server either inserts a new post (when `post.postid=0`) or updates an existing one (when `post.postid>0`). If successful, the returned promise resolves to a `Post` whose `modified` field (and `postid` and `created` fields in case of insertion) has the value returned by the server. In case of an error, the promise is rejected to `Error(String(response.status))`.
4. `deletePost(username: string, postid: number): Promise<void>` – This method sends an HTTP DELETE request to `/api/posts?username=:username&postid=:postid` to delete the post from the server. In case of an error, the promise is rejected to `Error(String(response.status))`.

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 up to you.

2. If you are not sure how to return a promise that will resolve to `Post[]` or `Post`, go over the class notes on Asynchronous Programming, in particular the slides on promise.
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`.
4. Since the JWT token is set as a cookie, the browser will include it automatically in every REST API request to the server. Just make sure that you manually authenticate yourself with the browser first to obtain a valid JWT cookie from the server.

Before you move on, make sure that your Angular app and Express server work well together without any problem.

Part G: Associate States with URL

As we mentioned earlier, an important feature of a single-page application is that a specific state of the application is associated with the corresponding URL. Remember that a change in the *fragment identifier* (or fragment id) of the URL is handled entirely by the browser. That is, when the current URL's fragment id changes, the browser does ***not*** reload the page from the server. Instead, the change is handled locally by the browser itself. This makes the fragment id the best place to encode the state of an SPA and support deep links and a good back-button behavior. In our project, we associate the three "states" of our app with the following fragment-id patterns:

URL	state
`#/`	This default path corresponds to the state `AppSate.List` and shows only the list pane, without showing the edit or preview view
`#/edit/:id`	This path corresponds to the state `AppState.Edit` and shows the list pane and the "edit view" for the post with `postid=id`
`#/preview/:id`	This path corresponds to the state `AppState.Preview` and shows the list pane and the "preview view" of the post with `postid=id`

To implement this association, we use the `window.location.hash` property and the `hashchange` event:

1. The `window.location.hash` property contains the fragment id portion of the current page's

URL. Whenever `window.location.hash` is set to a new value programmatically, a "new" URL with the given fragment id is appended to the browsing history and the browser "moves to" the new page (even though nothing really happens because it is just a change in the fragment id). Now, if the user now presses the "back" button now, they will "come back" to the "current page" from the "new page," changing back from the new fragment id to the current fragment id.
2. Whenever the `window.location.hash` value changes (because (1) the `window.location.hash` value is set programmatically (2) the user presses the back/forward button or (3) the user pastes a "deep link" in the address bar), the `hashchange` event is triggered to the `window` object.

Roughly, we will have to add the following logic to associate the three `appStates` — `AppState.List`, `AppState.Edit` and `AppState.Preview` — with the above three fragment id patterns:

1. Whenever `appState` changes, we set the appropriate fragment id to `window.location.hash`.
2. We create our custom handler for the `haschange` event, so that whenever the fragment id changes, our event handler takes appropriate actions.

More specifically, make the following changes to `AppComponent`:

1. Add `onHashChange()` method to `AppComponent`. This method is our custom `hashchange` event handler. Whenever this method is called, it should ensure that `appState` and `currentPost` are set to the appropriate values for the new fragment id. Remember that this method may be called when (1) the `window.location.hash` value is programmatically updated (2) the user presses the back/forward button and (3) the user pastes a "deep link" in the address bar.
2. Change the `AppComponent` constructor (or its `ngOnInit()` lifecycle hook) so that after all blog posts have been fetched from the server, `appState` and `currentPost` are set to the appropriate values for the fragment id used to load the app. For example, if the app is accessed through a deep link with the fragment id `#/preview/2`, `appState` should be set to `AppState.Preview` and `currentPost` should be set to the post with `postid=2`. Also, add the following line to the end of the constructor (or `ngOnInit()`) to add `onHashChange()` as a `hashchange` event handler:

```
window.addEventListener("hashchange", () => this.onHashChange());
```

3. Whenever you update `appState` or `currentPost` in your code, add the statement `window.location.hash = "corresponding_fragment_id";`, so that the URL fragment id reflects the app state. Even better, after you add `window.location.hash = "corresponding_fragment_id";`, you may want to move most (if not all) `appState` change statements to `onHashChange()`, so that `appState` changes are managed centrally by a single function. You may want to do the same for `currentPost` change statements as well.

Note: A more "Angular" way to add our `onHashChange()` method to the `hashchange` event handler will be to decorate it with the `@HostListener` decorator like the following:

```
@HostListener('window:hashchange')
onHashChange() {
    ...
}
```

But what we did here will work just fine.

Now you have finished implementing all major requirements of Project 4 except the integration of user authentication. Before implementing the authentication 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 server, take the following steps:

1. Build the production code of your Angular app by running the following command in the `angular-blog` directory:

   ```
   $ 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/` sub 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 already, 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, even though it is now served from the Express server.

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/` with status code 302. **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.

Note that this part of the project may requires the least amount of coding, but it is extremely important to get it right. **Since all server interaction depends on this part, if your authentication does not work properly, you are likely to get very low score for this project.**

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 Chrome Developer Tool, 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.

Part J: Style Your Application Using CSS (Optional)

Once you finish developing the functional part of the Angular editor, the required part of Project 4 is over. But we encourage you to add CSS styling rules to your app, so that the interface is aesthetically more pleasing. This part, however, is completely optional. As long as your code functionally satisfies our requirements, you will get the full score for this project.

There are a number of ways that you can implement styling your site. You can specify the detailed CSS rules all by yourself, without relying on a third-party "CSS library." This will help you learn the intricate detail of the CSS standard. Another way is to use a popular library for web page design and styling. For example, Bootstrap is one of the most popular web front-end libraries developed by Twitter. Bulma is another very popular CSS framework. Learning a third-party library will take time and effort, but once learned, they make it simple to add beautiful and interactive interface to your web site.

You are welcome to use a third-party CSS library as long as you make sure that your submission runs on the grader's machine. If you want to use a third party library, you will have to tell the Angular CLI that you want the third-party CSS (and JavaScript) files to be packaged together when your app is built and/or served. To do that, open the `angular.json` file and add the third-party CSS and JavaScript files to `styles` and `scripts` arrays like the following:

```
...
"styles": [
    "path/to/my/styles.css",
    "src/styles.css"
],
"scripts": [
    "path/to/my/javascript.js"
],
...
```

Once the files are added to the arrays, you can use their styling rules in any of your components just like when you include the files via the `<style>` and `<script>` tags in a standard web page.

As an example setup instruction, if you decide to use Bulma, you just need to install the Bulman module

```
$ npm install bulma
```

include the location of Bulma CSS file to the `styles` array:

```
"styles": [
    "node_modules/bulma/css/bulma.css",
    "src/styles.css"
],
```

Similarly, if you want to use Bootstrap, first install its module and its dependency

```
npm install bootstrap
```

and then add the relevant CSS and JS files to `styles` and `scripts` arrays:

```
"styles": [
    "node_modules/bootstrap/dist/css/bootstrap.css",
    "src/styles.css"
],
"scripts": [
    "node_modules/bootstrap/dist/js/bootstrap.bundle.js"
],
```

Submit Your Project

What to Submit

For this project, you will have to submit one zip file, named `project4.zip`, created using the packaging script provided below. This file will contain all source codes that you wrote for **Projects 3 and 4**. Before you package your code for submission, please do not forget adding the following `id` attribute values to various HTML elements in your app:

1. The "New Post" button in the List component should have the `id` attribute `list-new-button`.
2. Each blog post entry appearing in the List component should be in an HTML element with `id="list-post-:postid"` where `:postid` is the `postid` of the post.
3. The "Save", "Delete", and "Preview" buttons in the Edit component must have `id` attributes, `edit-save-button`, `edit-delete-button` and `edit-preview-button`, respectively.
4. The title and body input boxes/text areas in the Edit component must have `id` attributes, `edit-title` and `edit-body`, respectively.
5. The "Edit" button in the Preview component must have `id="preview-edit-button"`.
6. The rendered title and body in the Preview component must appear in HTML elements with the `id`s, `preview-title` and `preview-body`, respectively.

Since our grading critically dependent on the correct `id` values that are associated with required elements within your app, **if your `id` values are wrong, you may get as low as zero score.**

Creating `project4.zip`

After you have verified that there is no issue with your project, you can package your work by running our packaging script. Testing Project 4 requires running the blog-server of Project 3, so you will need to include your Projects 3 and 4 code in this submission. Roughly, the submitted zip file should be structured as follows:

```
project4
  +- blog-server
  |    +- bin
  |    +- public
  |    +- routes
  |    +- views
  |    +- ...
  |    +- package.json
  |    +- app.js
  |    +- db.sh
  +- angular-blog
       +- e2e
       +- node_modules
       +- src
       |    +- app
       |    |   +- ...
       |    +- assets
       |    +- environments
       |    +- index.html
       |    +- styles.css
       |    +- typings.d.ts
       |    +- ...
       +- ...
```

Create a temporary directory, called `project4`, and place your Project 3 code in the `blog-server` subdirectory and your Project 4 code in the `angular-blog` subdirectory. Then download the packaging script at the directory and execute it like `./p4_package`. The script will take a while to finish because it first builds a deployment version of your Angular code and then package it with other files. Once everything goes well, you will see a message like the following:

```
...
Creating project4.zip file ...
[SUCCESS] Created '/home/cs144/project4/project4.zip'.
```

Please only submit this script-created project4.zip to GradeScope. Do not use any other ways to package or submit your work! To ensure that your code works well on the grading machine, we provide the deployment script deploy.sh that will be used to run your code. Please download our script and make sure that your code can be successfully deployed just with the following command:

```
$ ./deploy.sh project4.zip

Deleting all documents and collections in the BlogServer database...
Loading initial documents to MongoDB using your db.sh...
Deploying your Angular code to the node server...
Installing dependent node modules...
Test deployment finished! Now your server is running at http://localhost:3000/
Visit http://localhost:3000/editor/ and make sure everything works as expected.
To stop the server, press Ctrl+C
```

If everything goes well, you will see messages like above. Once your see the above message, please visit http://localhost:3000/editor/ from your host web browser and make sure that your site works as expected. Roughly, the following functionalities are tested during our grading of your code:

1. Your submitted code successfully runs with `./deploy.sh project4.zip`
2. Authentication redirect
    - If you try to access `/editor/` without authentication, you are redirected to login page
    - Once you login as a valid user, you are redirected back to the angular editor app
3. List component
    - Your list component shows all blog posts from the user and contains a “new post” button
    - Pressing a post in the list:
        - Opens the edit component with the clicked post on the right
        - URL fragment ID changes to `#/edit/:postid`
    - Pressing the new button
        - Opens a new empty blog on the right
        - URL fragment ID changes to `#/edit/0`
4. Edit component
    - Your edit view contains title box, body box, and “save”, “delete”, and “preview” buttons
    - When “preview” button is pressed
        - URL fragment ID changes to `#/preview/:postid`
    - When “save” button is pressed

        - If it is an existing post:
            - The post is updated in MongoDB
            - Its modified time is changed to the current time
        - If it is a new post:
            - The post is added to the list pane (It is OK this happens at the time when "new post" button is pressed)
            - The post is saved in MongoDB
            - The post's postid and created time reflects its true value
    - When "delete" button is pressed
        - If it is a new post:
            - (If the post was added to list pane when the new button was pressed, it is removed from the list pane)
            - Post is deleted from MongoDB
            - URL fragment ID changes to `#/`
        - If it is an existing post:
            - The post disappears from the list pane
            - The post is deleted from MongoDB
            - URL fragment ID changes to `#/`

5. Preview component
    - Your preview component displays the post rendered in HTML and contains the "edit" button
    - When "edit" button is pressed:
        - URL fragment ID changes to `#/edit/:postid`

6. Deep link and back button
    - When the app is accessed through http://localhost:3000/editor/, it starts with the list component only
    - Pasting deep links like http://localhost:3000/editor/#/edit/1 and http://localhost:3000/editor/#/preview/1 displays correct post
    - When the page is "refeshed" by pressing the browser refresh button at a deep link, the browser comes back to the same state as before without any error
    - Accessing invalid deep links like http://localhost:3000/editor/#/lovestory,

http://localhost:3000/editor/#/preview/-21 does not create a problem. Accessing a deep link with nonexisting post like http://localhost:3000/editor/#/edit/2036 does not create a problem.

- Pressing the browser back button moves the app to the prior state within the app, not exits the app

Further Reading and Study

Before you go, we recommend a few articles and tutorials for further study.

First of all, either in our class or in the project, we did not cover the "modern tool chains" that are used by web developers in detail, including package mangers, module bundlers, transpilers, and task runners. Fortunately, Angular integrates these tools as part of its CLI and hides their complexities behind a few `ng` commands. But at some point you will have to know exactly what these tools are and when and why they are needed. To learn more on this topic, read the following article:

Modern JavaScript Explained For Dinosaurs

Second, the following article explains how complex web apps can be developed by (1) splitting them into simple components, (2) hierarchically composing the identified components into more complex forms, and (3) associating each component with minimal local states:

Thinking in React

Even though the article explains this development approach in the context of the React library, we took essentially the same approach in our Angular app development as well. In fact, if you are familiar with React, you may have noticed that our `AppComponent` works as a central place where all application states are stored and managed, a role typically played by the Redux library in React.

The following article generalizes this approach even more and explains modular and composable front-end development in a more general terms:

On composable, modular frontends

As you can see from the articles, the design principle we used in this project is not limited to the Angular framework, but is applicable to any modern app development with UI elements, including everyday mobile apps.

Finally, note that while this project mainly focused on building a “desktop-like web application” using Angular, Angular is a great tool to build a more “traditional” web site as well. We strongly encourage you to watch this two-hour tutorial to learn how to do it.

These are a lot of articles and tutorials to read and study and we understand that you may not have the time to read them all now. But if you are serious about web development, you will find that the information in the linked articles are invaluable.