Overview

This consists of two problems, one corresponding to an Angular app and the other corresponding to an Express app.

Problem 1 (Angular)

Your task here is to implement an Angular application, `FinalAngular`, that obtains a list of strings from a server and displays it. Other than the root `AppComponent`, the app consists of one component, `DisplayComponent`, and one service, `FetchService`, that do the following:

- `DisplayComponent`: This component is responsible for displaying a list of strings. It supports one property binding `[list]="expr"` and one event binding `(refresh)="callback();"` and can be used like the following:

  ```
  <app-display [list]="list" (refresh)="refresh();"></app-display>
  ```

  - Given the above directive, the component displays the input `list` as an unordered list.
  - The component also includes a button, which throws a `refresh` event when clicked
- `FetchService`: This service is responsible for sending a request to a server and obtaining a list of strings to display. In particular, it provides the following method.
  - `get(url: string): Promise<string[]>`: This method sends a GET request to the given `url` and returns a promise, which is resolved to the array of strings obtained from the server.
- `AppComponent`: When the application starts, the main application shell, `AppComponent`, uses `FetchService` to send a request to a server and passes the obtained list to the `DisplayComponent` for display. It should also monitor the `refresh` event from the `DisplayComponent`, so that whenever it is pressed, it obtains a new list from the server and passes it to `DisplayComponent`.

In final-angular.zip file, we created the skeleton code using Angular CLI. Note that the zip file includes `package.json`, but not the content in the `node_modules` directory, so you will have to install the packages in `node_modules` first before you start working on this problem. Your job now is to modify a subset of existing files in the zip file to implement the above functionality in the following sequence. For Problem 1, please *do not create or add any new files or install new modules that do not exist in **`package.json`***. Implement everything simply by modifying existing files in the zip file.

1. Go over the files related to `DisplayComponent` and understand what is in there. Your job for this part is to *modify only the files related to **`DisplayComponent`*** to implement `DisplayComponent` functionality. In particular, it should
   1. Support property binding like `[list]="expr"`

    2. Support event binding like `(refresh)="callback();"`
    3. Display the list of strings obtained from the property binding as list items `<li>` of an unordered list `<ul id="display-list">`.
    4. Include a button with `id="refresh-button"` which triggers `refresh` event when pressed.
2. Go over the files related to `FetchService` and understand what is in there. Your job for this part is to *modify only the files related to **`FetchService`*** to add `get(url: string): Promise<string[]>` method. When called, this method sends a GET request to the given `url` and returns a promise. If request is successful, the promise must be resolved to the array of strings obtained from the server. Assume that a successful response from the server includes the list of strings in the resoponse body as a JSON string array. If there is any error during request or if the server returns non `2xx` status code, the promise must be resolved to an empty array. That is, the promise is always resolved and never rejected.
3. Go over the files related to `AppComponent` and understand what is in there. Your job for this part is to *modify only the files related to **`AppComponent`*** to add the following functionality.
    1. When the app starts, it uses the `FetchService` to obtain a list of strings from http://oak.cs.ucla.edu/classes/cs144/examples/exam/angular/load/ and sends the obtained list to `DisplayComponent` for display.
    2. It monitors `refresh` event from `DisplayComponent`. When the event is triggered, it sends a `GET` request to http://oak.cs.ucla.edu/classes/cs144/examples/exam/angular/refresh/ and passes the obtained list to `DisplayComponent`.

Please make sure to follow the API exactly as described. Even if your implementation is reasonable, if you don't follow our API, you may get as low as zero point. Once you make sure that everything works fine, package your files using the `package.sh` file in the zip file and submit it to CCLE.

Problem 2 (Express)

Your task here is to implement an Express server that provides the following services:

- At `/proxy/`, it provides a "proxy" service to the oak server. That is, a request received at this end point will be forwarded to the oak server using the **`fetch()`** method of the **`node-fetch`** module to obtain response from it.
- At `/api/`, it allows a client to retrieve a student information from the MongoDB server.
- At `/student/`, it allows a Web browser to retrieve an HTML page rendered from a student data in the MongoDB server.

In final-express.zip file, we created the skeleton code using the Express application generator. Note that the zip file includes `package.json`, but not the content in the `node_modules` directory, so you will have to install the packages in `node_modules` first before you start working on this problem. Your job now is to implement the above functionality in the following sequence. Please *do not install any node modules other than the ones already included in **`package.json`***. For Problem 2, it is *OK to create and add new files** that do not exist in the zip file.

1. Implement the `/proxy/` end point that provides a proxy service to the oak server. More precisely, this end point must provide the following service.

    1. Any `POST` request received at this end point should be forwarded to http://oak.cs.ucla.edu/classes/cs144/examples/exam/proxy/
    2. The request forwarded to the oak server must use the `POST` method, include the `Host: oak.cs.ucla.edu` header, add the body received from the client with the appropriate `Content-Type` and `Content-Length` headers.
    3. In sending the request to the oak server, use the `fetch()` method of npm `node-fetch` module (which has already been included in the provided `package.json`). This module allows using the standard `fetch()` method in node.JS.
    4. Once the response is obtained from the oak server, respond to the client with the same status code obtained from the oak server and the `Host: localhost:3000` header. If the oak server's response includes a body, the response to the client should include the same body with the appropriate `Content-Type` and `Content-Length` headers.

2. Update the included `load.sh` script, so that we can load the following example student data into the `Students` collection of the `Final` database in the MongoDB server simply by running `mongo < load.sh`.

```
[{ "sid": 123456789, "name": "John Cho", "dept": "Computer Science", "title": "Designer" },
 { "sid": 234567890, "name": "Megan Fox", "dept": "Residential Life", "title": "Coder" }]
```

3. Implement the `/api/` end point that allows retrieving a student's information from the MongoDB database. More precisely, this end point must provide the following service.

    1. Given a `GET` request with the `sid=:sid` ***query string***, it must retrieve a JSON object whose `sid` is `:sid` from the `Students` collection of the `Final` database in the MongoDB server
    2. If successful, it returns the retrieved JSON object in the response body with `200` status code with appropriate `Content-Type` and `Content-Length` headers
    3. Before returning the retrieved JSON object, it must remove the `_id` attribute from the JSON object to ensure that this information is never exposed to any client
    4. If there is no matching student exists or the request does not include `sid=:sid` query string, return `404` status code
    5. If there is any error while retrieving data from MongoDB, return `500` status code

4. Implement the `/student/` end point that allows a browser to display any student's information in HTML. More precisely this end point must implement the following:

    1. Given a `GET` request with the `sid=:sid` ***query string***, it must retrieve a JSON object whose `sid` is `:sid` from the `Students` collection of the `Final` database in the MongoDB server
    2. If such student exists, it must render the student information in HTML using `views/student.ejs` template included in the zip file. Please do not modify the `views/student.ejs` file. Use the existing template as it is to display the student information in HTML.
    3. If there is no matching student or the request does not include the `sid=:sid` query string, return

`404` status code with the body rendered with `views/error.ejs`

    4. If there is any error while retrieving data from MongoDB, return `500` status code with the body rendered with `views/error.ejs`

5. Implement “redirection” mechanism, so that if any request is received at an end point other than the above three, the browser is redirected to `/student/?sid=123456789`, which will display the student John Cho’s information.

Please make sure that your Express server can be run simply by the command `npm start`. When `npm start` is executed, you can assume that the student data has already been loaded into the MongoDB server. Please make sure to follow the server API exactly as described. Even if your implementation is reasonable, if you don’t follow our API, you may get as low as zero point. Once you make sure that everything works fine, package your files using the `package.sh` file in the zip file and submit it to CCLE.