SER421 MVC/Express Lab

Learning Outcomes:

1. Understand and apply server-side MVC in a modern web development framework.
2. Use middleware (cookies, sessions) to provide forms of stateful behavior in a server-side app.
3. Use templating to render responses

Background:

The web application to construct is based on Lab 2. You are to build a small fictional new site for *NEW News Inc.* using *NodeJS/Express*. We will do this in increments, starting with creating your functional interface, then building a generic app, and then adding some additional features based on user roles.

Activity 1: Create a News API (40 points)

For the first activity you will construct an Application Programming Interface (API) for news based on Activity 1 of Lab 2, specifically R1-R5 that are repeated here:

R1. The ability to write a new news story to the persistent store (one or more files)
R2. The ability to update the headline of an existing news story
R3. The ability to change the content of an existing news story
R4. The ability to delete an existing news story
R5. The ability to return a collection of NewsStory objects based on a filter, where the filter checks for one or more of:
    a. Substring of the title
    b. Date Range
    c. Author

You can (and should) use your Lab 2 code for the business logic part of this. But now you will create an HTTP interface to these functions, and return an appropriate response. Specifically, design criteria for each of R1-R5:

D1. Your Activity 1 Express server should identify the proper HTTP verb for an incoming request
D2. You should produce a response with the proper response code and response headers
D3. Any request payloads required may be in application/x-www-url-encoded form.
D4. Response payloads should be in JSON.
D5. Error situations should be handled. What if the client sends the API server a request it does not understand? The wrong verb to a URL? Missing headers? A URL to nowhere? Respond with the proper 4xx-level error.

What about R6? Yes, you should submit test cases again, this time by creating those test cases in Postman in a collection and exporting that collection to JSON (just like you did in Task 2 of Lab 3)

Constraints:

C1. Lab 2 was a single user CLI application. You should design your API to work *at scale*. Assume that a large number of clients may be hitting your API at once. What does this mean for the design of your internal (in-memory) data structures? What does it mean for your management of the persistence store? Lab 2 constraints C1 and C2 apply, but C3 does not (how you manage the one or more files and even what is in the news files is entirely up to you). Constraint C7 from Lab 1 Activity 1 you have to think about – should you use synchronous or asynchronous file I/O? What are the ramifications for your persistent design? Finally, as part of this design you need to solve the "identification problem" – how do you uniquely identify a news story?
C2. Put your API server in a subdirectory named activity1 and file named NewsAPI.js.
C3. You may use Express and any Express-helper libraries, as well as any other libraries you see us use in the sample code.
C4. You should have a README.txt that enumerates the API specification for R1-R5 above. Indicate the expected request, the expected response, and error cases handled for each. If you want to get fancy you may even document your API as an HTML page or use a tool to generate the documentation (for example, apidocjs.com), though this is not required
C5. Please make your server listen on port 3001.

Pro tip 1: Work incrementally, meaning start with R1-R5 (I would suggest R5, R1, R4, R2, R3). Get test cases to work (R6). Then worry about the persistent store problem on the filesystem. Point breakdown: 5 points per R1-R6 (30), 10 points for refactoring the persistent store to work correctly at scale.

Activity 2: Create an MVC app with View Templates (25 points)

For activity 2 we will create an MVC app using Express with a View templating engine. The nice thing is, if you thought ahead in Activity 1, a lot of this is already done! Requirements for your server-driven app:

R1. Your app should have a landing page (/) that presents a proper web interface for the user to create stories, delete stories and search for articles. In short, perform R1, R4 and R5 for Activity 1. You do not have to implement R2 and R3 from Activity 1 in your UI. I am not going to give you the HTML as I did in Lab 3, you design a user-friendly page (or pages if you prefer).
R2. Your application should be personalized. If a user in this browser instance has not visited the app before, then you should include a web form asking the user her/his name. For all subsequent pages in the app the first line should say "Hello <username>". If a user has visited the app in this browser instance in the past 10 minutes, then the "Hello <username>" should appear on the landing page without asking for a new username.
R3. Each of the R1-R5 functionalities must return a web page (HTML) now, not JSON as in D4 of Activity 1. So now you need to support BOTH JSON and HTML. Further, if return HTML you must use Pug or EJS view templates.

In essence, Activity 2 is putting an HTML-based UI on our Activity 1, with the wrinkle of personalizing the app via the username.

Constraints:

C1. You are not allowed to use ANY client-side (in the browser) Javascript in this activity. If you use CSS you may not use it to implement a functional aspect of the app, but you can make it look nice if you want (however, aesthetically pleasing or responsiveness of the UI is not part of your requirements, we just need a functional understandable UI).
C2. Your HTML responses must all be rendered using Pug or EJS templates (you choose). This also includes error messages.
C3. Put your application in a subdirectory activity2 and name it NewsApp2.js
C4. Have your server listen on port 3002.

Activity 3: Make your app stateful (45 points)

Enhance your app to provide news stories for casual and subscribed readers alike. Also provide a community reporter interface, where any registered reporter in the field can provide news stories. To do this you will need to add *roles*.

Requirements**:**

R1. There are 3 roles in the system: "Guest", "Subscriber", and "Reporter". Here is what each can do:
   a. Guests can read any public news story
   b. Subscribers can read any news story
   c. Reporters can read public news stories and news stories they have posted, but not news stories posted by other reporters.
   d. Reporters can create public and private (for Subscribers only) stories, and delete stories. *When a reporter creates a news story, her/his username must automatically be assigned as the value of the Author attribute on the story.*

R2. On any Guest page, provide a "Login" link that goes to a page that presents an HTML asking for username, password, and role choice (Subscriber or Reporter). If the user responds with a username or password that match, then log that user in (track the login) under the Role selected. Obviously this is not how real authentication should work!

R3. All screens for a logged in user (by definition, a Subscriber or Reporter) should indicate the user's name, the user's role, and provide a "Logout" link that logs the user out. A confirmation page should be rendered indicating the user has successfully logged out with the set of actions that user did in this session (see R4) and a hyperlink back to the landing page as a Guest.

R4. Track the operations (R1) done for any logged in user (Subscriber or Reporter). At the bottom of *every page* for a logged in (non-Guest) user under a heading "Activity for <rolename> <username>", list the operations that logged in user has done so far (R1). When the user is logged out the final list is shown on the confirmation screen, and the list construction ends on logout. For example, suppose Jane is a Subscriber that has read 3 stories, 1 public and 2 private, the section would be:

```
Activity for Subscriber Jane:
-   Read the private article "Amphibious pitcher makes debut"
-   Read the public article "State population to double by 2040, babies to blame"
-   Read the private article "Breathing oxygen linked to staying alive"
```

(I got these headlines from here)

Note: R2 to R4 are not the same as, or even extensions of, Activity 2 R2. They *replace* Activity 2 R2. You should remove that code and put in this feature, and use a different underlying implementation mechanism (think of how R4 needs to be done).

R5. The landing page will display the titles of all news stories for all users regardless of role. This page should do the following:
   a. If the user in her/his present role can View the story, then the title should be a *hyperlink to that story*.
   b. If the user can Delete the story, then there should be a Delete button *next to* the story on the right. If the button is clicked the server should respond by deleting the story and re-rending the landing page.

R6. *Reporters* should have a hyperlink in the landing page to Create a new story. The link should take them to an Add story page, where the Reporter should be able to add a title, add a story, and indicate whether it is public or for subscribers only.

Constraints:

C1. There is no UI required to update headlines or content within the UI of the application (no Activity 1 R2 and R3).
C2. You must use Express session middleware to manage login/conversational state in the application.
C3. You are not allowed to use ANY client-side (in the browser) Javascript. You can use CSS to make the app look nice if you want but not to implement any functionality (CSS is not part of your requirements, we just need a functional understandable UI).
C4. Your HTML responses must all be rendered using Pug or EJS templates (you choose). This also includes error messages.
C5. Note: even with this contrived login and roles, it brings in new errors (failed authentication/authorization) for you to handle.
C6. Put your solution in a subdirectory activity3 and name the app NewsApp3.js
C7. Have your server listen on port 3003.

Overall Lab Submission:

- You may always include a README in any directory that explains anything you think we should know before grading.
- Submit a zipfile named <asurite1>_lab3.zip. When expanded, it should have the 3 required subdirectories specified above.
- We expect to run this code by using "node <filename>" at our command-line. However if you prefer we use .load, or if you understand NodeJS packaging you may use it (meaning we are fine with "npm install" and "npm run" and things like that if you know how to build a package.json). I put out some optional notes on packaging but it is not required in this course!
- Please submit stable solutions! Resist the temptation to change code last minute, and put the proper files in your submission. Absolutely no late or resubmissions. If your solution does not work we will not try to fix it and you will lose max points!
- I strongly suggest taking your own zipfile submission and installing, compiling, configuring and running your solution in a clean directory on your system. "It works on my laptop" is not an accepted reason for a grade appeal.
- Pay attention to the specified port numbers. We want to be able to run all 3 of your solutions at once.
- You can submit as many times as you want but we always grade the last submission. Don't get shut out at the deadline!