SER421 Online Lab 3 Fall 2017 due Tuesday 11/07 11:59:00PM

Learning Outcomes:

1. Understand and apply the programming model using basic HTTP functionality
2. Understand and apply server-side MVC in a modern web development frameowrk.

Requirements:

NOTE: I decided to give you an extra day for this lab, and you may work in pairs for Activity 2. Hopefully this alleviates some of the stress I saw on Lab 2. However, your next lab (Lab 4) will go on as scheduled and be due Monday 11/13.

Activity 1: HTTP Programming (50%)

The example code *http_server_external.js* retrieves JSON data from openWeatherMap.org's API based on a city the user enters. Starting with this program, modify it in the following ways:

1. Make it so that if any HTTP method other than GET or POST is submitted, a proper "Method not Allowed" message is displayed with the proper error response code. On the separate error page, provide a link to go back to the landing page.
2. Have the program "remember" what the last city entered was *on that browser, even if the browser was closed*. Indicate this in the app by pre-populating the web form with that city (if it is available).
3. Add a feature (via a second submit button whose value is *Forecast*), and when clicked retrieves the forecast JSON from the API. You should use the 5 day / 3 hour forecast (it is available on the free plan).
4. Right now this code will simply dump JSON to the screen for weather and forecast. Beautify the response in HTML.
   a. For weather, change this to output, in HTML, to include the city (name), current temperature (main.temp), the main/short weather description (weather.main), and the expanded weather description (weather.description).
   b. For forecast, change this to output, in HTML, to include the city (city.name), the min daily temperature (list.temp.min) and the max daily temperature (list.temp.max), the main/short weather description (list.weather.main) and the expanded weather description (list.weather.description).

Note that to complete these modifications to the existing sample code, you will need to get a free API key from openweatherdata.org/api. You do not need a paid plan; you will get a short-term free API key.

Constraints

- This Activity must be completed individually
- This Activity should use no other libraries than basic HTTP in Node. The sample code provide already has the require statements for http, url, and querystring; you should not need anything else.
- Absolutely no Javascript/CSS on the client-side.

Activity 2: MVC/Express Programming (50%) – *You may have a partner for this Activity!*

The web application to construct is functionally the same as Activity 2 from Lab 2. The requirements are repeated here with *changes underlined italicized red front*. You are to build a small fictional new site for *NEW News Inc.* using the *NodeJS/Express*. New News provides news stories for casual and subscribed readers alike. It also provides a community reporter interface, where any registered reporter in the field can provide news stories. The main functionality centers on a content management system, with flavors of CRUD (Create, Read, Update, Delete) functionality for various roles.

Requirements:

Roles:

1. 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), and delete stories. *You do not have to implement edit.*

Login/Logout (*use HTTP POST on any forms. Note one requirement has been removed from Lab 2*):

2. For this lab there is no real authentication. On any Guest page, provide a “Login” link that goes to a page that presents a username field in an HTML form. If the user is in the persistence store, then log that user in (track the login) under the Role the user was previously assigned. Obviously this is not how real authentication should work!
3. 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 and returns the user to the landing page as a Guest. If the user is a Guest then the screen should indicate a Guest at the top, and a link that returns the user to the landing page.

CRUD operations (*use HTTP POST for all HTML form submissions. Note there is no edit feature anymore*):

4. A View News (the landing) page will display the titles of all news stories for all users. 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 that takes the user to a page asking for confirmation.
5. Reporters should have a hyperlink in the page navigation 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.

An example of this application is available http://lead2.poly.asu.edu:8080/NewNews/news. *Your program must replicate the URLs of this application as closely as it possibly can (you are allowed some minor deviations to account for your design ideas). That is, you really need to think about your Express* ***routes*** *when doing this lab. These routes will replace the long if-else of your Lab 2*.

Constraints

The intent of this activity is for you develop *using NodeJS/Express and session middleware*, and to manage login state. With this in mind, here are your nonfunctional requirements:

1. The news file must be named news.xml and be in the same directory as your *Express* program. Similarly, users and roles should be stored in a JSON file in the same directory and named newsusers.json. *Use the same examples as Lab 2*.
2. *You must use Express session middleware to manage login state and conversational state in the application. Do not use cookies or hidden form fields or URL rewriting as in Lab 2*.
3. You are not allowed to use ANY client-side (in the browser) Javascript or CSS ANYWHERE in this activity. You do not have to make the application aesthetically pleasing.
4. Your server should return proper client error messages. Specifically, the HTTP response codes 400, 403, 404, and 405 should be returned in those situations where a web browser or other client is trying to improperly access the server (wrong HTTP verb, improperly specified request, unauthorized access to a page requiring proper authentication, and so on.
5. General programming best practices apply. Code should be readable, well-documented, and efficient. You need to create a portable solution (no absolute file paths and no hardcoded values). The solution should exhibit proper behavior even if hit by multiple users from multiple machines at the same time.
6. *You must render all of your webpages using either the Pug or EJS templating engine*.

Hints

- *You are no longer writing a CGI, but* your program will have to *route* incoming requests to different parts of your program. Think about the steps this will require, and what your main decision logic will be, and how to make modular these "parts".
- 3rd-party libraries are *NOT* allowed. *Note that you do not need 3rd party libraries for XML and JSON as they are built-in*.
- *Your program has to be correct in the face of many concurrent HTTP requests at once. You need to consider what can happen in your workflows from page-to-page and ensure a concurrent action does not violate the integrity of the data. This is not the same problem as the CGI as the architecture of your application is different*.
- *Note that requirement #4 takes some thought and study of HTTP response codes for request errors. You must not let a client directly access a URL is s/he is not allowed in the current role. Further, we will negative test your application by trying to hit URLs with incorrect HTTP verbs and incorrectly constructed requests.*

Extra Credit:

The original extra credit of Lab 2 is back, since now it makes sense as you are doing it in Node!

There is some optional content on File I/O in the Intro to NodeJS module. This content is not going to be on an exam, and is not needed for your labs. But, if you would like some extra credit, you can add the following to your Lab 2 solution:
Add the functionality to save new users when they are created to newsusers.json. Add the functionality to save new stories, and update/delete news stories, to news.xml. You can do this one of 2 ways:

1. Use synchronous file I/O. This type of I/O resembles the way you are used to doing file I/O from other coursework
2. Use asynchronous file I/O. You will need to fully understand the ramifications to your design.

If you do #1 I will award 10 extra credit points. If you do #2 instead, I will award 20 points. Note it makes no sense to do both, all I/O has to be either synchronous or asynchronous.

The catch: Keep in mind that your web application has to work in the face of many concurrent requests. Imagine a scenario where 2 reporters edit 2 different stories at the same time. The potential exists where reporter #1 completes edits first and saves news.xml. Concurrently, reporter #2 completes her/his edits and saves news.xml a fraction of a second later. The edits of reporter #1 could be lost (the "lost update" problem as it is known). Your EC solution must prevent this.

Submission:

- You may include a file named README.txt that explains anything you think we should know before grading.
- Everyone submits a zipfile (.zip only, not .rar, not .7z, not .tar.gz, not .tgz; note Java's "jar" tool actually creates zipfiles) for Activity 1 named <asurite1>_act1_lab3.zip. Note that your ASURITE is the one for ASU's computer systems, not your 10-digit id. I will deduct 5 points for not following the naming convention as it creates more work for us when grading.
- Only one student in each pair needs to submit Activity 2, and the submitters should be listed in the README. Name this submission <asurite1>_<asurite2>_act2_lab3.zip. You do NOT have to pair either, in which case <asurite>_act2_lab3.zip is fine.
- Note the "work in pairs" in Activity 2 exists so you may rely on each other (pair program), but it is not meant to "divide and conquer". Pairings that are determined to be divide-and-conquer will receive at most 50% of the credit for Activity 2. I reserve the right to interview students we suspect are not working collaboratively.
- If you want to do the extra credit but your partner doesn't, then complete it and note the person who completed it in the README.
- 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! Code that does not compile or throws runtime errors will result in severe point deductions and only limited partial credit!
- 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.
- Keep in mind you can submit as many times as you want but we always grade the last submission. So don't get shut out at the submission deadline!