COMP 2406A
Winter 2020 – Tutorial #4

Objectives

- Implement an HTTP server that serves HTML, Javascript and to-do list data
- Practice performing asynchronous requests using XMLHttpRequest
- Implement a client that interacts with the server to manipulate a to-do list
- Allow multiple clients to work with the same to-do list using polling

Expectations

- You should complete problems 1-5 from this tutorial document. The additional problems are there only to provide additional challenges for anybody interested. Problem 6 deals with an alternative implementation combining HTTP and web sockets. Problem 7 deals will adding extensions to the to-do list software.

Problem 1 (Creating a Simple Server)

In this tutorial, you will be expanding on the client-side to-do list application from a previous tutorial. The storage of the to-do list data will be moved to a server and you will build up a solution that will ultimately allow many clients to interact with the same to-do list.

To start this tutorial, download the todo-server.zip file from cuLearn. This zip contains three files: todo-server.js, todo.html, and todo.js. If you completed the second tutorial, you can use your own HTML and client-side Javascript for this tutorial. If you did not complete the previous tutorial, you can use the todo.html and todo.js files included in the zip file.

The server.js file contains code to respond to requests for the todo.html and todo.js resources. It assumed that these two files are located in the same directory as the server.js file. Run the server and request http://localhost:3000 or http://localhost:3000/todo.html to verify that the server is set up correctly before proceeding.

Problem 2 (Moving the To-Do List to the Server)

In tutorial 2, the to-do list was stored in the client-side Javascript and was initially empty any time the page opened. We want to modify it such that the list is stored on the server. This way, multiple clients will be able to interact with the server to view and manipulate the same to-do list information.

When the server starts, it should create a default list with several items. For now, all you need to store is the names of the items on the to-do list. However, if you do the challenge questions in Problem 7, the server will be expected to store data on which list items have been highlighted. You may want to consider using an easily extensible design if you intend to complete the challenge questions. One possible design would be to represent the to-do list data as an array of objects, where each object represents a to-do list entry. These objects can then have additional properties added to them to store additional data (highlight status, etc.).

Modify the client-side Javascript so that when the page loads, it makes an asynchronous GET request to the server to retrieve the to-do list data from the server. To handle this request on the server-side, you will have to create another route within your server's handling function (e.g., for when the method is GET and the resource is /list).

Once the information is received from the server, the client should update the webpage to show the to-do list to the user. This way, any time a user requests the to-do list page from the server, they will receive whatever list data the server currently has. Once you have this working, view the requests being made in the Chrome developer tools console. This will be a useful tool in debugging your implementations as the problems become more complex.

If you feel unsure how to send an asynchronous request from the client-side Javascript, consult the AJAX lecture slides/code or the W3Schools page on the XMLHttpRequest object:

https://www.w3schools.com/xml/xml_http.asp

Problem 3 (Modifying the Add Handler)

Modify the Add Item button click handler so that the new item name is added to the server's to-do list data. You will have to make an asynchronous request to the server and include the item data. As you are modifying data on the server, the request made by the client should be a POST request. There are multiple ways in which you can approach solving this problem. You could define a new route, such as /additem, and

send only the item information from the client as JSON text. An alternative approach is to make the POST request to the same URL as you use in the previous part (/list). The server can distinguish between a request for the list data (GET) and a request to modify the list data (POST). If you follow this approach, you may need to add some additional information to the request body to specify the type of operation (add or remove items). For example, your request could look something like:

{ "operation" : "add", "item": "task #1" }

As with the previous problem, you will need to add new code within your server to process this type of request. Remember that there are some extra steps required to process the body of a POST request. See the Intro to Node lecture example code if you do not remember how to do this. Note that since you want the client and server data to be synchronized (i.e., the same list at both locations), after the Add button is pressed, you should not add the new item to the client's page until you confirm the request was successful. You can use the status code in the response from the server to decide if you should add the new item to the list or not. If there is an error that prevents the item from being added, you should display an alert to the user so that they know to retry their request.

To test this functionality, load the to-do list page in your browser, add a new item, and then refresh the page. Since the client initializes the page by requesting the list data from the server, any new items should persist after the refresh, unlike tutorial #2, where the list would be reset to its original state every time the page loaded.

Problem 4 (Modifying the Remove Handler)

Modify the Remove Items button click handler so that the list of items to remove (i.e., the checked items) is sent to the server using an asynchronous POST request. As with above, you will need to add some information into this request to ensure the server knows what to do. As an example, the request body could have the following format:

{ "operation" : "remove", "items" : ["task#1", "task#2", "task#7"] }

The server should extract the items from this request and update the to-do list data by removing the associated items. As with the problem above, the client should not update its list until it receives a response from the server indicating that the request was successful.

Test this functionality by removing items from the list, refreshing the page, and ensuring the changes are reflected in the new list data received from the server.

Problem 5 (Polling the Server)

The final step of this tutorial will allow multiple clients to work with the same list data simultaneously. To do this, you will implement a 'polling' mechanism where the clients regularly ask the server to provide an updated list. To start, modify the initialization of your client-side Javascript to schedule an interval timer using the setInterval(function, milliseconds) function. For now, set the time interval to be five seconds. The function that this interval timer will call should make an asynchronous GET request for the to-do list data (i.e., the same thing that you do when the page loads) and update the user's page to contain the correct information. Thus, the client's page will update every five seconds to reflect any changes made to the server's data.

If you have done this correctly, you should be able to open two browser tabs and load the to-do list page in each of them. Any items you add/remove in one of the tabs will ultimately be reflected in the other when the five-second interval timer triggers. You can now allow any number of users to work with this same list data and have each of them see the changes made by the other users.

Think about how this approach is working. There is an important trade-off to consider. What happens if you have MANY users using this system? There will be MANY requests for new data, even if no changes have been made. If you increase the interval time, there are fewer requests, but the client may not see updates for a prolonged period. This is a common problem with a polling approach. You must decide if you want to increase the polling rate, which consumes more network and server resources, or decrease the polling rate and have the client's view seem less reactive.

Later in the course, we may cover web sockets, which are a way in which we can address this issue to some degree. If you are interested in learning it on your own, lecture recordings from a previous term covering Socket.io are posted along with this tutorial's resources.

Problem 6 (Switching to Socket.io)

Copy your completed polling code before starting this problem, as you will have to demonstrate both the polling and Socket.io solutions to get full credit for tutorial #4. Once you have a copy of your previous solution, start by using NPM to add the Socket.io module and its dependencies to your project:

```
npm install --save socket.io
```

Using the demonstration code from the Socket.io lecture and the documentation from the Socket.io website as references, modify your server code to create an instance of

socket.io and modify your to-do list HTML page to include the required /socket.io/socket.io.js resource.

Now, update your server and client code so that they use Socket.io to handle the communication of to-do list operations, instead of using polling and HTTP requests. Note that you will still need to handle HTTP requests for the HTML and Javascript resources, but will be able to clean up your server code by removing a number of the routes you no longer require. To accomplish this, you will have to define the types of events that your sockets will need to handle. It is likely you will need events for the addition of an item, the removal of items, and the initial load of the list data on a newly connected client.

There are several ways in which this can be implemented. While maximizing efficiency will not be necessary to get full marks, try to think of an approach that minimizes the overall amount of data that will be sent. For example, it would be more efficient to avoid sending the entire list whenever a single change is made. Instead, you may be able to send only the items that each client needs to either add or remove from its local list.

Problem 7 (Additional Challenges)

If you are looking to practice more or challenge yourself, consider the extensions below:

1. Allow the server to remember the list data, even when the server is restarted. To achieve this, you can save the list data to a file on the server. Should you write the updated data to a file on each request? Or would it be better to schedule the server to write out the data at regular intervals? What are the pros and cons of each approach?
2. Allow highlight data to be stored on the server and provided to the clients. This way, each user can choose items to highlight/un-highlight and changes are reflected to other users. You can also add color-coding, as was specified in last week's tutorial. If you want to go further, add more complex data to each item and provide a way for users to see this data on the to-do list page. For example, you might store the name of the person who added the item in the first place, or allow clients to add additional notes to each item. Provide a second area of the page where this information can be viewed after selecting an item. If the page gets too cluttered, you can add additional pages to do specific things (e.g., a 'view list' page and a 'view item' page).
3. Allow the user to sort the list by name or highlight status. Ideally, each user will be able to specify their own sorting preference. So, the server stores the same data, but each user can sort/display that data as they prefer, without interfering with the display of other user's data.

4. Add multiple to-do lists on the server. Allow the user to select which list they want to work with using a drop-down list. The data used to populate the drop-down list should be requested from the server as well.