

# COMP 2406 B - Fall 2022

## Tutorial #4

**Due Sunday, October 30, 23:59**

---

### Objectives

- Practice using JavaScript for client-side programming
- Practice performing asynchronous requests using XMLHttpRequest
- Implement a client that interacts with a server to retrieve new information to include in an HTML page
- Implement an HTTP server that serves HTML, JavaScript and to-do list data
- 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 need to submit solutions to all the problems in this tutorial. Our TAs will mark your submission. Remember to use the available resources (w3schools, documentation, Eloquent JavaScript book, lecture materials, etc.) for more information if you are struggling to complete the problems.

**Marking scheme.** For each tutorial, you will receive:

- 2/2 for submitting high-quality solutions to all problems. “High-quality” means that your code works (solves the problem) and is also neat and concise (no overengineering, please).
  - 1/2 for submitting solutions for all problems but some need improvement, or you are missing a problem.
  - 0/2 if you are missing several problems or your solutions are poorly done; or you do not make a submission, or your submission cannot be executed.
- 

### Part 1 (client-side interaction with an open web API)

In the first part of the tutorial, you will write client-side JavaScript that will be capable of interacting with an open web API called OpenTriviaDB. Your JavaScript will load one trivia question data from the API, populate the web page with question data, and allow the user to select an answer. Your code will then check the user's answer, update their score, and repeat the process. It is important to note that data for

## Tutorial 4

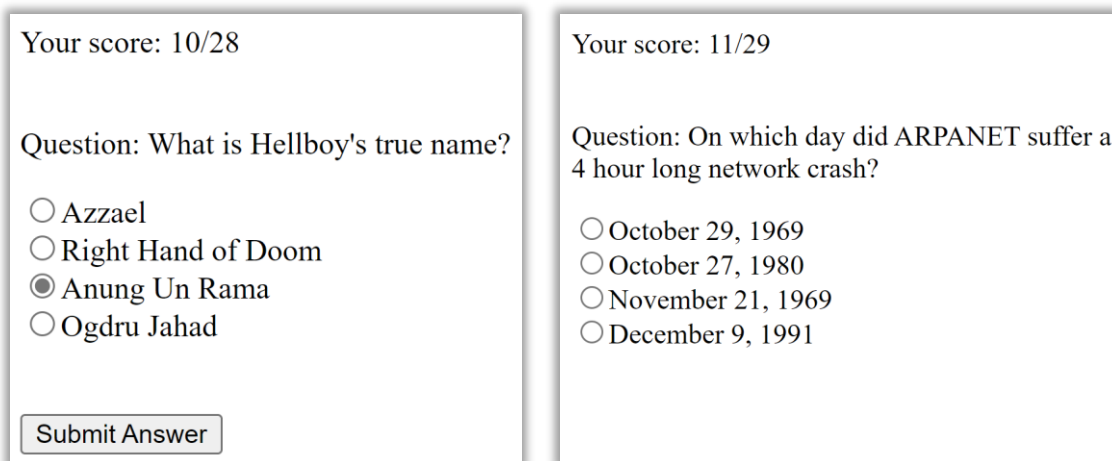
the questions is not contained in your JavaScript file - your JavaScript code will request question data from a server on the web.

Start by downloading the **triviaTest.js** and **triviaTest.html** files from Brightspace. You will need to complete **triviaTest.js**. Do not change the provided HTML file (**triviaTest.html**) – you do not need to submit it.

The provided **triviaTest.html** file creates a simple HTML page that contains the following:

- a `<div>` section for the user's score,
- a `<div>` section to hold question data, and
- a button for the user to submit the answer.

Here are two consecutive samples (the questions were loaded from OpenTriviaDB):



Notice the button appears when the user makes a selection. To make the button visible, change its style to `"display:block"`. This should be done in the JavaScript file as a reaction to the user's click.

The provided JavaScript file retrieves a question from the OpenTriviaDB API using AJAX. It creates an XMLHttpRequest object and submits a "GET" request to the URL

<https://opentdb.com/api.php?amount=1&type=multiple>. If this request is successful, the response text available in your callback function will be a JSON string with the following format:

```
{
  "response_code": 0, //you can ignore this for now
  "results": [ //an array of objects, which in this case only has one item
    { //a question object
      "category": "Science & Nature",
      "type": "multiple",
      "difficulty": "hard",
      "question": "What is considered the rarest form of color blindness?",
      //question text
      "correct_answer": "Blue", //some answer that is correct
    }
  ]
}
```

---

Tutorial 4

```
    "incorrect_answers": [    //incorrect answers
        "Red",
        "Green",
        "Purple"
    ]
}
]
```

The data in this JSON string represents the question data you will need to add to the page. The file **triviaTest.js** already has the code that converts JSON into a JavaScript object using the `JSON.parse(string)` method. You need to complete `render()` function that displays the question (and all four of the answers) on the page. It would make sense to store the answers as radio buttons or in a drop-down list. If you are using radio buttons, ensure you set the 'name' attribute of each radio button you create to be identical. This will ensure only one can be selected at a time. The placement of the correct answer should differ from question to question. It should not always be the first or last option.

Now that you can retrieve the first question and put it on the page, you will need to be able to accept an answer from the user. Add a handling code that allows the user to submit their answer by clicking the submit button. Once the user submits an answer, you will need to check if the answer is correct. You will have to get the value of the answer associated with the selected radio button and compare it to the correct answer of the current question. How will you do this? One of the most straightforward approaches is to set the 'value' attribute of the radio button to the answer it represents when you create the button. You can then find the radio button that is selected and access its value. Once you can retrieve the selected answer and compare it, determine if the user was correct and update their score on the page.

Now that the user can answer a single question and have their score updated, you can request another question by making another AJAX request to the same URL. You can repeat this process indefinitely or add a limit, so the game ends after X turns.

If you are having trouble with this part, you can consult the lecture notes or look at the w3schools tutorials.

## Part 2 (ToDo Server)

### Problem 1 (Creating a Simple Server)

In this part of the 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 a solution that will allow many clients to interact with the same to-do list.

---

## Tutorial 4

To start this tutorial, download the **todo-server.zip** file from Brightspace. This zip contains three files: **todo-server.js**, **todo.html**, and **todo.js**. If you completed the second tutorial, you may 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 is assumed that these two files are located in the same directory as the **server.js** file. Review the basic server code to ensure you understand what it is doing. 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 the previous tutorial, the to-do list was stored in the client-side JavaScript and was initially empty any time the page opened and all the data was lost when the page was refreshed. We want to modify our application such that the to-do list is stored on a server. This way, multiple clients will be able to interact with the server to retrieve, view, and manipulate the same to-do list information. When a user refreshes their page or opens the page for the first time, the client software will request up-to-date to-do list data from the server.

When the server starts, it should create a default array with several items. All you need to store is the names of the items on the to-do list.

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 URL requested is `/list`).

Once the information is received from the server, the client should update the page 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. At this point, if you choose to divide the data and display it in your original tutorial #3 solution, the changes required should be minimal. The source of your data will change (i.e., you now request it from a server instead of having it stored in an array within the client JavaScript), but the display portion of your code will likely remain unchanged.

Once you have this working, view the requests being made in the Chrome developer tools console (mouse right button click, then "Inspect", to bring up the developer tools). This will be a useful tool in debugging your implementations as the problems become more complex.

If you are unsure how to send an asynchronous request from client-side JavaScript, consult the AJAX lecture materials or the W3Schools page on the XMLHttpRequest object:

[https://www.w3schools.com/xml/xml\\_http.asp](https://www.w3schools.com/xml/xml_http.asp)

---

**Tutorial 4****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. To do this, your client JavaScript will have to make an asynchronous request to the server and include the item data inside the request body. As you are creating new data on the server (i.e., a new to-do list item), the request made by the client should be a POST request. There are multiple ways in which you can approach solving this problem. One option is to make the POST request to the same URL as you used in the previous part (/list). This strategy of using a GET /list request to retrieve the list data and a POST /list request to create a new item within that list data is a principle of RESTful design, which we will discuss later in the course. As your server code can check the HTTP method of an incoming request (e.g., GET vs. POST), the server can distinguish between a request to retrieve the list data (GET) and a request to add to the list data (POST).

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. Review the Node.js readings or the lecture materials 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 #3 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 PUT request. The exact HTTP verb that should be used in this case is debatable, but using a PUT request will allow our server to distinguish between retrieve/add/remove requests. As an example, the request body could have the following format (where "task#1", etc. are the names of the items to 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.

---

**Tutorial 4**

### 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 updated list data. 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 execute 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 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. If your server was accessible to the outside world, anybody could connect to your server and share the same to-do list data (we'll get to this later). It is important to remember that the server plays the primary role in storing the data. Each client is requesting its list data from the server and updating the server any time a change is made. The clients do not communicate directly with each other at all.

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.

This issue can be resolved to some degree using web sockets. We will not study Socket.io in this course. However, if you are interested, you can learn this topic on your own.

## Part 3 (Save Your Work & Submit)

Once you have completed part 1 and part 2 problems, zip all your files for this tutorial and submit them to Brightspace. Do not forget to add comments to your code to clearly specify what each part of the code is doing and which problem it is solving.

To submit your tutorial, you will need to **zip** (compress) all your files for the required tutorial into a single .zip file and submit it to the Tutorial submission on Brightspace. Name your file **T4-YourName.zip**. Make sure you download your .zip file and check its contents after submitting it. If your .zip file is missing files or corrupt, you will lose marks.