# COMP 2406: Fundamental of Web Applications

Carleton University
Fall 2024 Final Exam Solutions

December 12, 2024

## Be sure to *carefully* read all of the instructions below.

There are 18 questions (26 including sub-parts) on 3 pages worth a total of 50 points. Answer in a copy of the supplied text file template with the filename `comp2406-final-username.txt`, replacing username with your MyCarletonOne username. Please do not corrupt the template as we will use scripts to divide up questions amongst graders. **If your submission has the wrong Student ID, is not formatted so questions are separated properly, or does not otherwise pass the supplied validator, 10% will be deducted from your final exam grade.**

This test is open book. You may use your notes, course materials, and other online resources. If you use any outside sources during the exam, you **must cite** the sources. Your citation may be informal but should be unambiguous and specific (i.e., if you referred to the textbook, indicate what chapter and page you looked at rather than just citing the textbook).

All explanations should be concise and to the point (generally no more than a few sentences, sometimes much less). Citations are only required if you consulted outside materials. If you find a question is ambiguous, explain your interpretation and answer the question accordingly. You won't be graded for spelling or grammar, just do your best to make your answers understandable.

You **may not** collaborate with any others on this exam, and *you may not use any LLM-based AI services.* You may add a question to the exam as question nineteen where you discuss what you liked about this class and what would you suggest could be changed; this question will be worth up to five extra points, depending on the quality of the answer. This exam should represent your own work. **Randomized and selected interviews will be conducted after the exam. If an interview reveals that your exam is not your own work, your work will be forwarded to the Dean for disciplinary action.**

**Do not share this exam or discuss it with others who have not taken it.** Some students will be taking it at other times due to accommodations. Solutions will be released once everyone has finished the exam (including deferred exams).

If you have questions during the exam, please go on the **lecture zoom** or **message Prof. Somayaji via Teams**. (Zoom will only work during the normal exam time.)

You have 150 minutes (2.5 hours). Good luck!

1. (5) Answer the following questions about the function `validateSubmission()` in validator.js from authdemo2 (from Assignment 4):

   (a) (2) At a high level, what does the function argument `fn` contain? What about `f`?

   **A: fn contains the filename and f the contents of the submission to be uploaded.**

   (b) (1) At a high level, what is the for loop in the function doing?

**A: The loop adds the text of the question answers to the page.**

(c) (2) What does the variable `q` contain, at the level of JavaScript (the type of data) and in terms of the program's semantics?

**A: q is an object. It contains the parsed submission.**

2. (2) How could you restructure routeGet() to eliminate the series of if statements without using a switch statement? Explain your approach at a high level.

**A: We could instead create an object where the properties are the routes to be tested for and the values are functions that should be executed for that route.**

3. (2) At the beginning of authdemo2.js, there are the following lines:

```
15 import * as db from "./authdb.js";
16 import * as template from "./templates.js";
17 import { expectedQuestionList,
18          checkSubmission } from "./static/validator-core.js";
```

These lines contain two different syntaxes for import. What is the functional difference between the `* as` and {}, and why would you use one versus the other?

**A: The first syntax for import makes all exported symbols into properties of the "as" object in the current scope, while the second includes specific exported symbols into the current scope. The first is useful if you don't want a module's exported symbols to conflict with those in the current code, as they will all be assigned to an object for which you can specify the name. However, if you want a module's exported symbols to exist in the current scope, use the second syntax—so long as you know exactly what those exported symbols are and don't want to change any of their names.**

4. (2) How could you restructure addSubmission() to be a synchronous function using `then` rather than `await`? Explain how the code would have to be restructured.

**A: addSubmission() only makes one await call, on line 164 for req.json(). So instead, to make it synchronous, we can replace this line with** `req.json().then((submission) => { rest of function })`**, where we put addSubmission()'s code code after line 164 into the body of the anonymous function, replacing "rest of function".**

**However, its actually much trickier than it seems as handler is an async function that must return a response object, and with then() the addSubmission() function will have nothing to return and so handler won't have anything to return either. So everything would have to be changed to use then() rather than async/await including Deno.serve().**

5. (5) Consider the following code:

```
import { DB } from "https://deno.land/x/sqlite/mod.ts";
var db = new DB("test.db");
```

   (a) (1) What Deno permissions does this code require to run?

   **A: Needs `--allow-read` and `--allow-write`. (Note that downloading the module doesn't require network permission because it is done by Deno not the app.) (half a point for each permission)**

   (b) (1) Will this code ever cause Deno to access the network? Why?

   **A: Yes, when first run, to download the sqlite module. (0.5 for yes, 0.5 for the explanation)**

   (c) (1) Will this code ever create a file? Explain briefly.

   **A: Yes, if the sqlite database doesn't exist it will be created. (0.5 for yes, 0.5 for the explanation)**

   (d) (2) After this code is executed, can we be certain that `db` is ready for regular data queries (that insert, update, or select records)? Why or why not?

   **A: We cannot be certain the database is ready for regular data queries. If the database did not already exist, it will be created with no data tables, and thus regular data queries referencing specific tables will fail. (1 for not ready, 1 for the explanation)**

6. (2) Briefly describe the process for adding a dynamic HTML document to dbdemo2. What must be changed or added, and why? Be sure to refer to specific functions where appropriate.

   **A: To add dynamic HTML content, first you have to add an if test to routeGet() or routePost(), depending on whether it is for GET or POST client requests, that when successful runs a new function. This new function should return an object that has a content, status, and contentType fields (at minimum) filled in appropriately for the new dynamic page. (1 for modifying routeGet/routePost, 1 for defining a new function that returns the right kind of object)**

7. (2) Where in authdemo2 are student IDs extracted from submissions? What constraints are placed on student IDs in this part of the code?

**A: On line 81 of validator-core.js, in checkSubmission(). It places basically no constraints except that it be at least one character in length—a period matches any character in a regular expression.**

8. (2) Does authdemo2 verify that the student ID of the uploaded submission is the same as that of the currently logged in student? How can you tell by looking at the code?

   **A: No, because addSubmission() doesn't compare student ID's anywhere in its code (unlike the version of this function in authdemo).**

9. (3) Consider the following code from authdb.js (part of authdemo2):

```
24  const expectedQarray = expectedQuestionList.split(",");
25  const createQFields = expectedQarray.map((q) => "q" + q + " TEXT").join
        (", ");
```

   (a) (1) What is the purpose of the `createQFields` constant in authdb.js?

   **A: This constant specifies the question fields when the submissions table is created.**

   (b) (2) Briefly explain what the `map` method does in the definition of `createQFields`. Be specific.

   **A: At a high level, this map transforms every element of expectedQarray from a number to a field declaration. It works by running the function passed into it on every element of the given array, returning a new array where the elements are the values returned by these function executions.**

10. (2) Why can't web applications depend on client-side code to do input validation? In answering, give one example of this problem from class. Be specific.

    **A: Web applications can't depend on client-side validation because it is very easy for such validation to be bypassed. We saw this in class when we used wget or curl to upload invalid submissions.**

11. (4) What are four HTTP response codes we've used in class? What was each used to indicate in the class code? Be specific.

    **A: All the codes we used in class are included at the beginning of authdemo2.js. 200 is for success (whenever we returned what was requested), 303 is to redirect to a new page (used after POSTs to redirect to a new page), 400 is for invalid data (a messed up submission), 401 was used when a wrong username or password was entered (unauthorized), 403 was for when the current user isn't allowed to access the current resource**

**(student users trying to access the admin pages), 404 was for invalid URLs (page not found), 500 is for errors that shouldn't happen (internal server errors, things like a session with an invalid access string), 501 was for HTTP methods that weren't implemented by the server (anything other than GET or POST).**

12. (4) Consider the following two lines from getCookies() in authdemo2.js:

```
134     if (!cookieStr) return null;
135     const cookies = cookieStr.split(';');
```

Please explain what each line is doing 1) at the level of JavaScript code execution and 2) its purpose in the context of the function.

**A: The first line checks to see whether we have any cookies at all, and if not makes the function return with null. This line works by checking the truthiness value of cookieStr, and if it is false (e.g., empty object, null, undefined), then it returns null.**

**The second line transforms a string that contains all of a request's cookies into an array of cookies (with each element of the array being a key value pair string, with them separated by =). It works by splitting the cookie string into an array with the separator being a semicolon.**

13. (2) The string "COMP 2406 Authorization Demo" is repeated in multiple places in authdemo2. How could you change authdemo2 so it was only specified in one place in the application? Explain the necessary changes at a high level.

**A: We'd have to transform all of the static HTML pages (in the static subdirectory) into templates that had a variable in place for the page titles. We could do this by embedding them in JavaScript files (like template.js), or we could use a templating library such as handlebars.**

14. (2) Where in authdemo2 is submission data uploaded to the server? And where is it extracted from the incoming request for further processing? Be sure to specify where the transmission is initiated and where the final data is received for processing. Please specify precise filenames, functions, and line numbers for each.

**A: The upload happens on line 92 of validator.js, in doUploadSubmission() (the call to fetch()). The data is received for further processing on line 164 of authdemo2.js, in addSubmission() (the await req.json()).**

15. (2) Consider the following code from authdb.js:

```
32 db.execute(`
33   CREATE TABLE IF NOT EXISTS ${submissionTable} (
34     id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
35      studentID TEXT UNIQUE,
36      ${createQFields}
37    )
38  `);
```

If the word "UNIQUE" was deleted from the above, how would the behavior of authdemo2 change? Be specific.

**A: If UNIQUE was deleted, it would be possible to upload multiple submissions for the same user/student ID, rather than the current situation where newer submissions overwrite older ones.**

16. (5) Consider the function addAccount() from authdb.js:

```
132     export function addAccount(username, password, access, studentID,
            name) {
133       return db.query(`INSERT INTO ${authTable} ` +
134                       "(username, password, access, studentID, name) " +
135                       "VALUES (?, ?, ?, ?, ?)",
136                       [username, password, access, studentID, name]);
137     }
```

(a) (1) Why does this function have "export" as part of its declaration?

**A: so it can be used by code in authdemo2.js, specifically on line 263 in createAcct()**

(b) (2) Will the INSERT query here ever overwrite data in the database? Explain briefly.

**A: It will never overwrite data because simple INSERTs can only add data to a table. To overwrite data on an INSERT, we have to add OR REPLACE.**

(c) (2) Why do we not use string interpolation/construction to put the new values directly into the SQL string, rather than passing them in a separate argument and using question marks for VALUES?

**A: We don't use string interpolation here because the inserted data is supplied by untrusted parties (users of the application). String interpolation of untrusted data into an SQL statement can result in SQL injection vulnerabilities (i.e., an attacker can run arbitrary SQL queries on a back end database).**

17. (2) If an attacker got a copy of the accounts table from submissions.db, could they use that information to login to a system running authdemo2 with the same accounts table? Would the attacker have to do any additional work to successfully login? Explain briefly.

**A: If they could extract the passwords from the password hashes, an attacker could**

**use this information to login into accounts. However, for this to happen, the attacker would have to run a dictionary attack on the password hashes, hashing and comparing all possible passwords to see which one matched. Because passwords are salted, they would have to do this separately for every account. Unless the passwords were pretty bad, this would not be easy to do because argon2 is a strong password hashing algorithm (it is very inefficient to compute in bulk).**

18. (2) What code is responsible for session expiration in authdemo2? Be sure to specify the file, function, and lines.

    **A: Line 189 in authdb.js, in getSession(), enforces session expiration. It tests whether expiration time has passed by comparing the expiration field with the current time, returned by Date.now().**

19. (5) See the exam directions, this question was for extra credit, up to 5 points.