Creative Project 4 Specification

Due: Saturday, June 3rd 11:30PM PST

Important note for seniors/grad students: The final day of classes for seniors is June 2nd. Seniors will have a few modifications to support the earlier end of the term, including CP4 being an optional assignment (if submitted, they can get practice with API development and receive feedback that may be helpful for their final project).

Overview

For your fourth (and final!) Creative Project, you will create your own Node.js/Express web service available for use with AJAX and `fetch`. Once again, as a Creative Project, you have freedom to have more ownership in your work, as long as you meet the requirements listed below.

Ideas for CP4

Being that this is our last Creative Project, it is also your chance to wrap up your work in CS 132 with a final portfolio you can publish, showcasing your exploration of web programming technologies. As always, we encourage you to explore the new material covered in class, as well as related (but optional) content we may link to along the way, as long as you follow the course Code Quality Guidelines and adhere to Academic Integrity policies. You may choose to do a new website for each CP, or build on the existing project from previous CP's. **Your CP4 must be distinct from your Final Project API (if you are unsure about possible overlap or want to request exceptions, you can reach out to EI).**

As long as you meet the requirements outlined below, you have freedom in what kind of website you create. Here are some ideas for Spring 2023, some of which were APIs built by previous students:

- A service to retrieve data about Caltech houses and memberships
- A service to get/post notes
- A service to retrieve data for personal projects/artwork
- A service to get/post movie or game reviews
- A service to get information about Caltech courses
- A service to manage song playlists
- A service to manage recommendations for course-scheduling
- A service to retrieve information about dog breeds/other animals
- ...

Recall that one of the benefits of writing functions in Node.js instead of code you can write in JS on the client-side is that you can quickly process data with file I/O. In this CP, it can help create a pretty useful API. As an idea, consider writing your own folders and files to process and return data in JSON format in your API (e.g. refer to the Cafe API Case Study which implemented an API version that processed directories and files to return text and JSON responses).

We've primarily practiced processing txt/json files and directory contents, but depending on your own interests and project focus, you can process other files like `.csv`, `.gcode` (3D coordinate files for 3D Printing models), `.pdb` (protein code encodings), etc.

This CP is designed to give you an opportunity to practice writing both client (JS) and server-side (Node.js) code on your website. This is a great chance to think about how your project could showcase what you've learned so far in web programming for your own code portfolio after the term ends, so we encourage you to explore implementing different features of your web service!

We encourage you to explore the new material covered in class, as well as related (but optional) content we may link to along the way, as long as you follow the CS 132 Code Quality guidelines. This is your chance to:

1. Continue to build a website that you can link to on your resume or code portfolio (CPs can be public, most HWs cannot be).
2. Ask EI and/or TAs about features you want to learn how to implement (we can provide more support for CPs than HWs) and ask for feedback/ideas on Discord.
3. Apply what you're learning in CS 132 to projects you are personally interested in and may use outside of just a CS 132 assignment.
4. Get feedback on code quality when learning new technologies in CS 132 to implement for the Final Project which will be worth more points.

External Requirements

Your project must include the following six files at a minimum (all client-facing files should be in a `public` directory served with `express.static("public")`):

- index.html - main page of your website
- styles.css - file to style your .html file(s)
- *.js - containing your client-side JavaScript code (including `helpers.js`)
- **new:** an `app.js` web service your `.js` file fetches from with at least two different GET requests.
- **new:** a APIDOC.md file to document your `app.js` web service. You can find a starter template here, or you may add a link to published Postman API Documentation (recommended).

- **new:** your project's `package.json` file generated using `npm init` and including any dependencies you use (at minimum `express`).

Important Note: All non-static (public) files (e.g. your Node.js/Express web service) should be at the root. **Do not submit your `node_modules` directory.** We will run `npm install` using your submitted `package.json` to install the modules necessary for your project.

Similar to the Final Project, you will be writing both client-side JS and Node.js to incorporate in your website, where your client-side JS makes AJAX requests to **your** Node.js web service which responds with information.

Client-Side JavaScript:

Your website must somehow dynamically load information from the web API you've implemented and present information from that response on the page. This requires that you must:

- Respond to some event (whether it's the window load event, any UI event, or, a timer event) to determine when to fetch the data (using fetch), and
- Dynamically integrate the data returned from the API into the page by manipulating the DOM elements in some non-trivial way using `element.appendChild`, `element.removeChild`, or `element.replaceChild`
- Use the `checkStatus` function from lecture to throw an Error if the `fetch` response status is not ok before processing the data (refer to this example). This is a helper function we are allowing (encouraging) you to use in your AJAX programs (included in the provided `helpers.js` program), but the rest of your functions must be your own.
- Handle any errors caused in the `fetch` request/response process by displaying a helpful message to the user on the page (i.e. **without** using `alert`, `console.log`, or `console.error`). To do so, you should define a function (e.g. `handleError`) to implement the error-message-displaying and pass that function in the `fetch` call chain's `catch` statement. **Do not output any raw API error message text to the page.** You may (and are strongly encouraged to) use an equivalent AJAX solution in your client-side JS with `async`/`await` and `try`/`catch`.

Node.js/Express Web Service

Your Node.js web service should handle at least two **different** endpoints, one of which outputs JSON and one which outputs plain text (`res.type("text")`).

- Ideas: You may choose to implement two GET endpoints and have an optional query parameter of type or format to let a client choose their returned format as text or JSON, or you may choose to implement one endpoint that always returns text and one endpoint that always returns JSON.

- At least one endpoint must support req.params and/or req.query parameters. Remember that path parameters are best used for required endpoint parameters, and query parameters are usually preferred for optional parameters (refer to Week 8/9 lecture slides and videos for examples and discussions on this)
- Your Node.js must handle at least one invalid request with a 400-level error header (with content type set as plain text or JSON) and a descriptive message as demonstrated in lecture examples (although you are encouraged to handle more than one type of invalid request). Possible errors include missing required GET query parameters or passing invalid values for parameters (whether they are path or query parameters). Remember that a missing path parameter will not be appropriate to handle, since the Express route will always require the parameter is passed to match the path string. You may include other error codes as well (e.g. 500-level errors for server-side errors relating to file-processing), but must support at least one 400-level (invalid client request) error for full credit. Part of your grade in CP4 and the Final Project will come from identifying necessary error-handling and supporting 400- vs. 500-level errors appropriately.
- You may implement other types of Node.js functionality for more practice before the Final Project. In particular, we **strongly** encourage you to practice with file/directory processing with the `fs/promises` and/or `globby` module and `async/await` which will be required in the Final Project (or the SQL equivalent if you took CS 121).
- Document your API in an APIDOC.md file or Postman. You can find a good example with best practices discussed here (while written in 2012, it's still one of the best references for getting started with API documentation). It is not uncommon to use an .md file for API documentation (the link has such a template you can build off of) A .md file is written in Markdown, documentation on which is here. You can also use other documentation tools such as Postman, which you can alternatively link to in your submitted APIDOC.md file (make sure it is publicly-accessible). We recommend Postman, as it is a common API Documentation generator that previous students have had strong documentation scores without missing endpoints, error-handling, or example responses. A video is posted on Canvas on how to use Postman.

Internal Requirements

For full credit, your page must not only match the External Requirements listed above, you must also demonstrate that you understand what it means to write code following a set of programming standards. Your code should maintain good code quality by following the Code Quality Guide. Make sure to review the section specific to JavaScript! We also expect you to implement relevant feedback from previous assignments. Some guidelines that are particularly important to remember for this assignment are included below.

Overall Requirements

- Your HTML, CSS, JavaScript, and Node.js should demonstrate consistent and readable source code aesthetics as demonstrated in class and detailed in the Code Quality Guidelines. Part of your grade will come from using consistent indentation, proper naming conventions, curly brace locations, etc. Don't forget about the Prettifier VSCode extension to help with formatting!
- Links to your .html, .css and .js files should be **relative links**, not absolute. For your static (front-end) files, refer to Week 9 lecture slides/videos on what we expect for serving a public directory with the express.static middleware.
- Similar to other assignments, **all file names and links in your project must be lowercase without spaces** (e.g. img/puppy.jpg but not img/puppy.JPG or img/Puppy.jpg). This is enforced to avoid broken links commonly occurring in CP/HW submissions due to case-insensitivity of file names on Windows machines.
- If you want to explore other HTML/CSS/JS features beyond what we've taught in class, you must cite what resources you used to learn them in order to be eligible for credit. We strongly encourage students to ask the staff for resources instead of finding online tutorials on their own (some are better than others). **Citation is also expected for any code that is inspired by lecture examples** (a brief source code comment or page footer citation is sufficient).
- Your page should include appropriate content and copyrights and citations. If we find plagiarism in CPs or inappropriate content, **you will be ineligible for any points on the CP**. Ask if you're unsure if your work is appropriate.

HTML/CSS

- HTML and CSS files must be well-formed and pass W3C validation. This also includes any generated HTML as a result of DOM manipulation in JS (e.g. remember to include descriptive `alt` attributes when creating `img` elements dynamically).
- Previous assignment expectations apply here as well.

JS

Continue to follow the standards in the JS Code Quality Guidelines and CP2/HW2/CP3/HW3 specs. This includes good use of function decomposition, separation of JS from HTML/CSS, minimizing module-global variables, etc. A few reminders of Code Quality requirements for JS so far:

- When adding interactivity to your website, you should handle any events (like a mouse event, keyboard event, timer, etc.) by responding to them using a JavaScript function(s) in your `.js` file. You should not have any JavaScript code in your HTML and you should not

have any HTML tags as strings in your JavaScript code (e.g. `el.innerHTML = "<p>Foo</p>";`).

- Minimize styling in JS (e.g. changing the `.style` property of elements) - prefer adding/removing classes to DOM elements instead, and style the classes in your CSS. Remember that there is an exception when dynamically generating values for styles or positions that are not reasonably factored out in CSS.
- Your *.js file should be linked to your `index.html` or other `.html` files using `<script defer src="...">` **in the HTML `<head>`**.
- Any .js code you write must declare `"use strict";` at the top
- All client-side JS must use the module-global pattern (remember that you should not use this module pattern in `app.js` though)
- Localize your variables as much as possible. You should not use any global variables (outside the module pattern in client-side JS) - see Code Quality Guide for reference. Only use module-global variables whenever absolutely necessary. Do not store DOM element objects, such as those returned by the `document.querySelector/qs` function, as module-global variables
- Define program constants with UPPER_CASED naming conventions (using `const` instead of `let` in JS). Examples of common program constants include a file path to your images if you are working with many images in your JS, an API base url (`BASE_URL`), `CLIENT_ERR_CODE`, and `SERVER_ERR_CODE` as demonstrated in class).
- Decompose your JS by writing smaller, more generic functions that complete one task rather than a few larger "do-everything" functions. Limit your use of anonymous functions - meaningful behavior should be factored out with a named function.
- Any AJAX requests in your JS code must use the Fetch API (e.g. do not use jQuery's ajax function)
- Do not make unnecessary requests to your API. That is, there should be no code in your JS that requests from an API and **never** does anything with the response. Also be mindful of any duplicate requests (e.g. if you have the information from the first request when clicking a button to fetch data, think about how to avoid another repeated request later if the data will always be the same).
- Your JS code must pass JSLint with no errors (note: we have modified some of the configurations to help you catch common style issues in this link).

New CP4-Specific Requirements (in addition to following the Node.js section of the Code Quality Guide):

- Your Node.js web service should specify the correct content type by setting the header using a consistent convention before outputting and response (this includes 400 errors). These headers should only be set when necessary and should not be overridden.
    - It is easy to override headers, such as `resp.type`, at first - if you are uncertain about how to check this, feel free to ask on Discord/OH! More details are discussed in Week 8/9 lecture materials.

- Your Node.js code should not generate any HTML (though you may check with EI for exceptions to this rule depending on the context of your project)
- Similar to your client-side JS, decompose your Node.js/Express API by writing smaller, more generic functions that complete one task rather than a few larger "do-everything" functions - no function should be more than 30 lines of code, and your Node.js should have *at least* one helper function defined and used. Consider factoring out important behavior for your different GET requests into functions. If you need to send the `req`/`res` objects in a helper function, use middleware appropriately.

Documentation

Place a comment header in each file with your name, section, a brief description of the assignment, and the file's contents. Examples:

HTML File:

```
<!--
    Author: Lorem Hovik
    CS 132 Spring 2023
    Date: April 1st, 2023

    This is the index.html page for my portfolio of web development work.
    It includes links to side projects I have done during CS 132,
    including an AboutMe page, a blog template, and a cryptogram generator.
-->
```

CSS File:

```
/*
   Author: Lorem Hovik
   CS 132 Spring 2023
   Date: April 1st, 2023

   This is the styles.css page for my portfolio of web development work.
   It is used by all pages in my portfolio to give the site a consistent
   look and feel.
*/
```

Use JSDoc to document *all* of your JS functions with `@param`, `@returns` as discussed in the Code Quality Guide. If a function does not have parameters and/or a return statement, these annotations should be omitted in the respective function.

```
/**
 * Brief description of the function (including requirements, important
 * exceptional cases, and avoiding implementation details).
 * @param {datatype} parameterName1 - parameter description
 * @param {datatype} parameterName2 - parameter description
```

```
 * @returns {datatype} - Description of the return value (if applicable)
 */
function functionName(parameterName1, parameterName2) {
  ...
}
```

CP4-Specific Documentation Requirements (in addition to following the Node.js section of the Code Quality Guide):

- Document your Node.js functions using the same JSDoc requirements as your client-side JS file (e.g. `@param` and `@returns`).
- Briefly document any request-handling functions (e.g. `app.get`, `app.post`, etc.) with comments about the endpoint (you can use lecture code for examples). If you use named functions for your callback, you can rely more on JSDoc conventions.
- In your `app.js` file comment, a brief description of your Node.js web service and the parameters/responses that would be important for you/other developers to understand the program. See the Code Quality Guide for an example. Use your APIDOC for a more descriptive public documentation of your API (used by clients).

Grading

Our goal is to give you feedback, particularly on the internal requirements and style and documentation, so you can incorporate this feedback in your Final Project which will be worth more towards your final grade.

This CP will be out of 9 points and will be distributed as:

- External Correctness (4 pts)
- Internal Correctness (3 pts)
- Documentation (1 pt)
- APIDOC (1pt)

You can earn one extra point (up to 9 points total) for successfully adding a form (in a `<form>` tag) to your site that includes a way to "submit" the information to the server. Upon submit, the form uses the FormData object and AJAX Fetch with a POST request to retrieve, then modify the site in a "non-trivial" way (review Lecture 14 for forms and POST requests if needed). Our definition of "non-trivial" is that it must dynamically integrate the data returned from the API into the page by manipulating the DOM elements (similar to the base assignment using `element.appendChild`, `element.removeChild`, or `element.replaceChild`), and you must appropriately handle errors.