HW4 Common Code Quality Mistakes

Below is a list of common code quality mistakes that are often made when first writing server side code using Node.js. While this is by no means exhaustive, nor is following this guide a guarantee of a grade, it is in your best interest to make sure your code meets these standards as well as the expectations we have outlined in section and in lecture.

Read the spec! Read the spec! Read the spec!

The specs for the assignments in this class are long but very detailed. Make sure you read the entire file to ensure you are not missing any of the requirements. A good strategy is printing out the spec and crossing things out as you finish them.

Make sure *all* your code lints!!

This one is pretty self explanatory. 100% of the time get a better grade if you make sure that the code you submits passes the linter. Remember that both your server and client side JS must lint. The linter runs every time you commit and push to GitLab so do so often and early!

DO NOT include a BASE_URL of localhost:PORT in your client side JS file

This is not a modular implementation as well as unnecessary. You are restricting a client to only being able to access the responses from that hard coded port in the base url. You can simply `fetch` for the route. For example, if you have an endpoint in you `app.js` file with path `/really/cool/route` you can simply `fetch("/really/cool/route")` and **NOT** `fetch("localhost:8000/really/cool/route")`.

Good project directory structure

All forward facing static files (such as the client side JS, HTML and CSS) should be in the provided `public/` directory. Your app.js file should be located at the root. See section and lecture for examples of good directory structure.

Overwriting Content Types

You should never be overwriting content types in your endpoints. The best way to avoid overwriting types is to only set the type before sending the response. If all responses sent by your endpoint (including error handling) have the same content type, it is fine to set the type once at the beginning of the endpoint.

Not setting the content types on error handling

Just like with a normal response, it is important to be setting the content type in any of your error handling cases otherwise it will be the default `HTML` type which is not what we want. You can always check the content type of your response in the network tab of the chrome dev tools.

Await not in a try/catch

You should always be using a `try/catch` block to handle any possible errors that arise with `async/await` and pending promises.

Package.json not updated with the correct dependencies and "main" file

There is no need to ever submit the `node_modules` directory that is generated when you `npm install`. However, it is very important that you submit an updated `package.json` with the proper dependencies and main file. When you run `npm install`, your `package.json`, provided that it was initialized, is automatically updated with the dependencies. You also need to ensure that your `package.json`'s "main" key is set to your main node file, so that node knows what file to run when called.

Passing the response and request object as params to helper functions
The `res` and `req` objects should not be passed around as parameters as they belong to the endpoint. Passing them around can often lead to weird bugs that are hard to trace. It is great and required to factor out code into helper functions but you should prefer pulling out other functionality such as file processing or data manipulation.

Long callback chains instead of promisifying
Nested callback chains often make your code very complex, cluttered and harder to understand. Prefer promisifying as demonstrated in both lecture and section paired with `async/await.`

Incorrect const naming conventions and line length.
Magic values (both strings and numbers) can be factored out as constants using the `UPPERCASE_NAMING` convention. We also declare modules such as express as `const`s in our code at the top of our file but there should be done with the `camelCase` naming convention. Additionally, make sure your lines do not exceed 100 characters.

Good documentation
On assignments like HW4 you are not required to submit an `APIDOC.md` containing the documentation for your web service. This means that you will need to include the documentation details as part of the header comment for your `app.js` file. This includes the request format, request type, description of the request, example request and example response (just like in your APIDOC for this assignment). Additionally, make sure to include a brief comment above each endpoint describing its purpose. If you are unsure of how to format this, please refer to section examples from week 8 such as the tricky typing test or the hybrids exercise.

Functions defined in the middle of file instead of the end
In order to make your code easier and cleaner to look at it is best to define all your endpoints first and define any helper functions below.

Modules not defined at the top of the file
All modules you define should be defined at the top of your file. You should only define modules you will use so if you are not supporting any `POST` requests, there is no need to include `multer.`

app.use(express.static("public")) should be at the bottom of the JS file
This line of code should be at the bottom of your file right above the `PORT` your server is listening for.

Program does not terminate after res.send()
`res.send()` does not terminate your program meaning that if you have code after a `res.send()` it will execute. This is not what we want as this can often lead to other unexpected errors. Instead you should be properly wrapping your code in `if/else` blocks or using a boolean flag to ensure that no code executes after any of the `res.send()`s. This includes both regular responses and responses sent as part of your error handling. A particularly common error is executing code after a catch block sends an error message.

Check if query and body parameters are set, but not path parameters
Path parameters, which are a part of a path, will always need to be included for the path to be reached. However, query parameters (for example, “`/words?limit=5`”) and parameters in `POST` bodies can be

omitted by users, so you should check if they are undefined to avoid computing with nonexistent values, and perhaps send an error if they are invalid.