Code for Grading

Samuel Edson, Sabrina Drammis, Grant Gunnison, Danny Sanchez

Basic Coding

This project is getting large, so legible code that is quick to understand is essential for this point in development to go smoothly. We try our very best to keep all code under 80 characters wide. I have even taken out the handy \ operator a few times to split up strings and such, though I still don't like doing that because it looks messy no matter what.

We made a good effort to line things up and make things easy on the eyes. I don't remember seeing any require statements that didn't have their equal signs lined up. Our comments, that aren't specifications, are concise and just prevalent enough that they don't blast you with the comment syntax color every other line. I had a comment for every line once and found it was actually physically hurting my eyes, and my grader's.

Another important aspect of a large project: directory structure that makes sense. Our app is structured in a really definitive manner. Sectioned off js for node in *app*, statically served files in *public*, views in *views*, and other folders so obvious they aren't worth mentioning. I'll preemptively give some commentary on Modularity right now, but a directory structure that everyone understands makes that first step for adding a file so much easier, and people know that the first step is often the hardest.

Modularity

Our util.js is larger that it has been in previous projects. The content of that file is actually an interesting reflection on our app. We talk about this in Verification,

but we have functions to determine whether the user is logged in, and two others to determine whether they are logged in as the right type of User (Employer or Student). This makes our app more modular because it prevents us from writing the same checking code at the top of every route. Also, a reader of the code can quickly spot *isLoggedInEmployer*, and see that the route requires that the user be logged in as an employer before the request will work.

Other functions in our util.js include verification code for keeping our database invariants intact, and code to send responses and error responses. These are great for the same reasons mentioned for the login checkers, but have the added bonus of being easily *changeable*. For example, if we had an issue escaping quotes down the road for whatever reason, we could just fix the problem in utils.js, and we know it would be fixed everywhere else. This reduces a lot of struggling now, and later.

Given that our project is very large at this point, there are even more things we can do that may be small, but have a big impact on our ability to put the polishing features to our app for the final. I mentioned it in the last code writeup, but detailed comments above routes are essential for us to build our front end. We did a good job with this early on and it has seriously decreased the time spent writing incorrect ajax requests on the front end, even after we went through a couple iterations of a few views.

Verification

We use many techniques to verify our code is running correctly. For starters, we have unit tests for all Employer and Student routes. We use QUnit with Ajax requests, and check that the response is what we want. This is extremely important in any API Web App because of the separation of routes and pages that request those routes. This is a sensitive area because they are functionality separated across multiple files, but must “line up” correctly. They are also fundamental to the working

of the app, so writing tests makes you think like the client, and actually helps you write a better API.

We also used a method for validating our database schema in Mongoose. The library provides a convenient validator function that prevents the database from saving entries that do not return *true* from a validator function. This is useful for preventing undefined or empty entries from saving, or making sure an email is valid before it is ever saved. We use the Validator Node Package which provides convenient functions for this type of schema invariant, like validating emails.

We also prevent Students searching for Students and Employers searching for Employers. This isn't necessarily a problem depending on how we define our privacy policy, but it is clearly an *incorrect* functionality. It is something that would feel wrong given the context of the App. We did this using using three utility methods, *isLoggedIn*, *isLoggedInEmployer*, and *isLoggedInStudent*, which do what they say they do. Passing this as a parameter as a route ensures that the route isn't entered unless the function says that it is a valid login.

Security

We take a number of steps to address security issues. For starters, we equip our database with an escaping mechanism to make our text in HTML format. Luckily, Mongoose has a *pre* middleware method that functions on the input of a database action. We also use the Node package called Validator that has an escape() function that replaces special characters with their percent escaped equivalents. Together, we use them, for example, in a Mongoose *pre* save on an Employer's name and company name. This would mess up most Javascript code that could be injected, and also makes our data HTML friendly.

We use the Node Module, Helmet, to enforce our Content Security Policy. We make sure no inline Javascript is allowed. Also, Helmet has an xssFilter that our app can use to prevent cross-site scripting by setting the *X-XSS-Protection* HTTP header. This isn't a complete CSP, so we define our own cross site scripting rules, which only let code run from jquery, ajax, and self. We do the same for style, allowing jquery, bootstrapcdn, and self. This does a lot to prevents code from other web sites being run, preventing many cross-site scripting attacks. Lastly through Helmet, we tell our app to set *frameguard* to *deny*. This ensures that our site does not become part of an iframe, restricting our content to our own site.

We originally wanted to use the *csurf* Node Module to prevent cross site request forgery, but we are leaving it out because we haven't figured out how to incorporate it with tests. We may want to use it for the final product, but for now we are leaving it out because much of what it does is prevented by Helmet.

We also put some protection on our cookies. We make the session cookie httpOnly because it ensures that clients cannot access the cookie from their own code.

Lastly, we disable x-powered-by so that hackers don't know we are using Express with Node.