Sprint Two - Project One

Build a secure search engine using Node.js with data that is stored in PostgreSQL and MongoDB.

INTRODUCTION AND SUMMARY

Hello all, and welcome to the final sprint for this semester. There will be two projects in this sprint, the first one you may complete in a group of up to three people, the second one you must complete on your own (these two projects will be assigned separately). The first project, you will make a search engine website that a user can send search terms and get back results. For the second project, you will create a web service that accepts an array of numbers and gives back a JSON tree representation of those numbers. The steps for each project will first be summarized into easily digestible chunks in the sections of their respective project documents, each document will then go into details on each step to elaborate on the project requirements. Please read the entire summary for each project carefully to get an idea of the scope of the project and the core requirements before continuing into the implementation details.

PROJECT ONE

This project can be completed in a group of up to three. Again, the basic idea of this project is that we're going to create a search engine using mock data that you generate by any means. You'll need to use both postgresql and mongodb to store your data (you can either split it between them, or have redundant data in each, it doesn't matter). You should generate a website that a user can easily use to access the search service that you've built, this does not have to be a complex effort, but do note that the completeness of this website, as well as the usability, will be factored into the competition element discussed at the end of this document. To begin, we briefly outline the project in summarized steps:

Step 1. Choose a subject area for your search engine.

Step 2. Find a mock data generator and generate > 1000 rows of data for your chosen subject; https://www.softwaretestinghelp.com/test-data-generation-tools/ Or find similar subject sample databases for both MongoDB and PostgreSQL.

Step 3. Build both PostgreSQL and MongoDB data storage for the mock data generated.

Reminder, mock data generators can export to either SQL or JSON (or both).

Step 4. Build Node.js objects to query and display the search results. The user should be able to choose their data source(s); postgres, mongo, or both. Before they execute their search.

Step 5. Implement security features to signup and login before any search can be executed.

(Use passport.js or similar node package if you like, don't underestimate the learning curve for these packages. May be simpler to build your own basic registration and login)

Step 6. Results should be displayed to page, similar to google search results.

Step 7. Each requested query keywords should be saved to a log file on disk OR in either postgres, mongo, or all three (you choose). It is important that these keywords be saved with datetime stamp and user_id.

COMPLETE with DISTINCTION

If you are wanting to achieve a COMPLETE with DISTICTION for this sprint you need to complete the following two Bonus items as well as achieving the with DISTINCTION described in the assignment rubric.

Bonus. Use automated unit testing to test your code where possible.

Bonus. Ensure your code is structured to deny any kind of query injection hack.

https://severalnines.com/database-blog/securing-mongodb-external-injection-attacks

https://blog.crunchydata.com/blog/preventing-sql-injection-attacks-in-postgresql

LEARNING OUTCOMES

1. Proven ability to build a simple web site using Node, Express, Postgresql, and MongoDB.
2. Proven ability to organize node.js code into files, folders, and modules to deploy a small full stack project.
3. Proven ability to create a PostgreSQL database with a simple database table structure and insert sample data into this database
4. Proven ability to create a MongoDB database and add documents to the database.
5. Proven ability to query both the PostgreSQL and MongoDB databases for a search term.
6. Proven ability to display the search query results to a web page.
7. Proven ability to create a simple login to authentic search users.
8. Proven ability to log all the search actions to an events log file saved to disk.
9. Proven ability to extend a projects scope to include additional features. And to work independently as an individual, or team, to learn the additional technologies to implement added features.

PROJECT APPROACH

The first step in the process is to mock some data, in order to do that, it's a good idea to have some idea of the type of search engine you'd like to create, but in reality, when I was implementing the canonical solution for this project during the design phase, I actually looked at what types of mock data was available, and then chose a subject based on that. In terms of mocking data, there are many services out there that are suitable, one service that will work well for this project is https://www.mockaroo.com/ , I encourage you to explore alternatives to your heart's content. When I was looking at mockaroo, I noticed that one of the things they could generate was short and long descriptions of medical procedures, as well as fictional procedure numbers, and so I decided I'd make a medical procedure search engine based on this mock data. However, you feel free to use whatever mock data you'd like, on any topic that you'd like, again, for the competition element, aesthetics and creativity will be factored in, so feel free to get as a creative as you'd like. For example, on mockaroo alone, other possibilities included a database of animal information, or car information, or numerous other things, so there are plenty of options out there. In fact, you can even use real data for added utility if you'd like, as long, of course, as it is legal for you to obtain it and use it in an educational context.

In any case, once you've decided on your search engine topic, and you've generated > 1000 rows worth of entries for your databases from your desired mocking service or services of choice, we now need to get into the core implementation. So, to accomplish the next step, which would be step 3, you're going to need to add your data to your databases. Services like mockaroo make this relatively easy to do - they can generate data as SQL insert statements, which obviously make it quite easy to insert data into Postgres, for example. If your mocking service does not allow this, you may have to write a script that takes the data in the format that you get it in, and translates it into an appropriate insert query, perhaps using the pg node module, or just string manipulation written out to a file for you to bulk-insert using pgAdmin. For mongodb, the process is similar. Again, mockaroo can generate JSON files, so when I implemented my solution for this, that's what I did (mockaroo helpfully has an option to generate the file as an array of objects, which is perfect for mongodb). Once you have this data, you have a few options for inserting it into the database, you can of course use MongoDB's Compass utility (roughly the equivalent of pgAdmin), which actually has an "add data" button in it, that accepts a JSON file as input, and will automatically add it to the collection of your choice. Alternatively, you can do it from the mongodb shell by using the insertMany method and pasting the entire file contents as the parameter to that query, for example. Again, if your mocking service doesn't offer a mongodb friendly format, you'd want to create a script that either translated your data into an acceptable mongodb query that you could run from the shell, or simply one that used the mongodb module in node to insert the data in an appropriate format programmatically.

Okay, so now we have our data in our databases, it's time to start implementing some code to complete step 4. So, we actually need to implement a few webpages to accomplish this. Firstly, we know, from the summary, that we're going to need to create a login system and account system somehow, so we may as well plan for that. From a user experience perspective, it makes sense to have a home page to help them navigate around, a place to help them find the login page, the sign up page, and critically, the search page. Don't forget that they need to be logged in

before they can search, and you need their user_id to record their searches to a log file or database.

To actually implement the search, we're going to need two pages. The first one is the "query page" – this is like the google homepage for us, with the search bar on it. On this page, the user can type a string that they'd like to search one or both of our databases for, and it should have a submit button so that they can complete their search. Once the submit button is pressed, we should have a second page for the search results, again, this would be like the google results page, which, in the case of google, would have useful links to websites, but for us instead will just have rows containing the relevant entries from our database. I've included two screenshots of a simple implementation of this idea below, with the query page as figure 1 and the results page as figure 2.

Figure 1 - Search Page

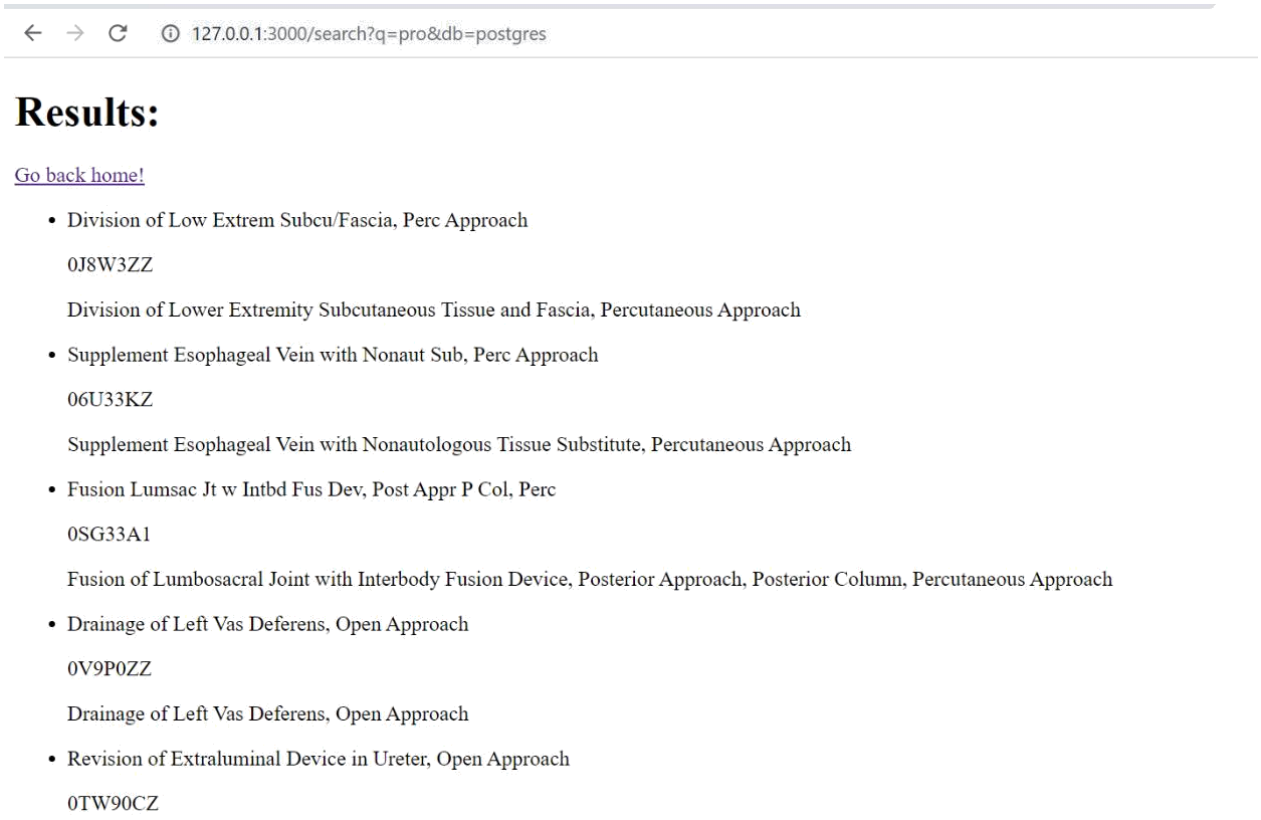

Figure 2 - Results Page

Once you have a client-side completed, you'll be able to test your backend implementation. To implement the backend, you'll want to use express as a base since we're going to be adding a login system among other things. We'll have to implement several routes; you'll want routes for each of your client pages, and you'll also want routes to service the requests from those pages (for example, the login service will probably have a 'get' route that gives the user access to the html page with the login forum, and a 'post' route which services a post request from the html page to log the user in on the back-end). Note that for your login system, you can choose one database or the other to store your user accounts in, you don't need to use both. For your search routes, my recommendation is to use ejs to put the results from your database(s) onto the page, refer to figure 2 to see what the output of that might look like. In terms of actually generating data to inject into those templates, you'll need to query your databases for this. In my implementation of this system, my search route took two parameters, a query, and a database, both of these come straight from my forum as you can see in figure 1 above. The database parameter of this request has a value of either "postgres" to search only postgres, "mongodb" to search only mongodb, or "both" to search both. The query parameter of the request is simply a string that the user provides, and we want to find out if any fields of any entries in our database(s) have any part of a string that matches the provided query. To do this in SQL, I'll give you a hint that you can use the "like" operator (https://www.w3schools.com/sql/sql_like.asp).

For mongodb, you can use a regular expression with the find command – thankfully not a complicated regular expression, if you use a regular expression that has just the query string inside of it, and nothing else, that is actually the same as using the "like" operator in SQL, which is

convenient for us, since we haven't studied regular expressions in depth. In node, you can turn any string into a regular expression like this:

```
const some_string = "pretend_i_am_a_query";

const regular_expression_version = new RegExp(some_string);
```

If we want it to search in a case-insensitive way, we can do:

```
const regular_expression_version = new RegExp(some_string, 'i');
```

With mongodb you could also look at creating an index that is applied across many fields within the JSON structure. When a search term is looking to be found the index will search for the term across all the included fields.

Okay, now that we have a good handle on the backend implementation, there are only a few final details to talk about. Firstly, we need to keep a log of every query that a user sends. In the example implementation we saved records for this to both databases. You can choose one database or the other, or use an npm package for logging, or create your ability to log searches to a disk file – we'll want to run these logging insertions every time the user sends a search query, so we'll want to make sure that's in our code for the searching route.

In addition, we need to make sure that the user **cannot** access the search functionality of our website unless they are logged in, so we'll need to check for that in the relevant routes.

Finally, for the first bonus, if there is an opportunity to break some logic out into a testable, reusable function, that is a good candidate for unit-testing with Jest. For the second bonus, we'll want to ensure that our database is robust against any query injection attacks. SQL injection is fairly common, so test your website using a couple of well-known attacks to make sure you're not vulnerable to it:

https://www.w3schools.com/sql/sql_injection.asp

Likewise, mongodb also has query injection vulnerabilities:

https://blog.sqreen.com/mongodb-will-not-prevent-nosql-injections-in-your-node-js-app/

Honestly, for this project, the security aspect isn't as important, and it is only a bonus, but it is something that we should be aware of when we take on real projects. SQL injection, in particular, is one of the first things people will try on public websites, so it's important that we're at least aware of it as a threat – with query injection, people can take full control of our database, and sometimes even full control of our entire server, which is obviously not good.

PROJECT DELIVERABLES

A single zip file (with your team name) that contains the following;

1. All the SQL scripting to create the database and tables.
2. All of the JSON or SQL script files to add / insert the data.
3. All of the node.js files used to implement the project.
4. All other text or json files implemented by the project.
5. All dependent .json files that reference the project dependencies (ie. package.json)

Note: do not include the node_modules folder in the zip file.