Semester 4 Final Sprint

Welcome everyone to your final sprint week project! For this project you will be working as a team and individually to reimplement your final projects for Semester 3 in Java, although unlike usual we strongly suggest going for the default project this time around as we believe it has the highest learning potential. As before you can propose your own ideas with the following constraints:

* Reimplement a previous project from Semester 3 in Java.
  + Must have a backend component.
  + Must have a database component.
  + Must an object-oriented-programming style.
  + Must have a front-end component (react app).
* Develop a comprehensive set of tests to ensure that it functions identically.
  + Should include both unit tests and user stories that comprehensively cover all the features from the previous project.
* The project should be pushed to GitHub such that you use a PR workflow (trunk-based dev flow). Each team should have a repo each for the front end and back end. Teams are responsible for creating these repos.
* Implement your solution so that it can run inside a docker container (for the SpringBoot APIs). The front end can be deployed to AWS using S3 and simple web hosting.
* Deploy your container to AWS (using Elastic Bean Stock for example).
* There should be a proposed team component and a proposed individual component.
* The UI you developed in Semester 3 should be refactored to use your new APIs developed in this sprint. We will supply a base react app developed by Jamie as the base for every team if you need a starting point.
* The team should manage all work in an agile way, for example using a Trello board to work through a 2 week sprint.
* Share a “demo video” of your final, deployed, solution working! Have the full team join a Teams call and record it! **This deliverable is very important!!!**
* **NOTE:** I recommend that all data for this sprint be stored in a SQL database like MySQL. There is no need to use Mongo for this project.

# Sprint Overview

The focus of this sprint will be, as 1 **team**, taking your final projects from Semester 3 and reimplementing it in Java (using a similar approach to the SpringBoot apps we have been working with in class). For a refresher, the final project for semester 3 was a search engine with both PostgreSQL and MongoDB data in which a user could search either one or both databases for a given keyword. **Individually** however you will be working on reimplementing the binary search tree service from the same project in Java. *Attached will be the sprint document from Semester 3.*

This code should also be implemented so that it can be run inside a docker container (or cont. Once you are done with this all, you must develop a comprehensive set of tests using a combination of J-Unit and manual test specifications to ensure that the application functions identically to the original - you’ll want your test scheme to cover every single feature of the app either manually or using unit testing, so be careful to ensure that collectively your design for testing is comprehensive. After your testing and codebase is complete, you are to deploy the container(s) to AWS (using ECS for example) and record a demo video (simple way to do this is for everyone to join a Teams call and record (then post the link) the call.

***NOTE: Only the API and UI needs to be deployed to AWS, the binary search tree can be shared in a GitHub repo. For this part of the sprint unit tests are expected and it does not have to be a SpringBoot project, a simple maven based project like we did in the first half of the semester is a perfectly fine choice here.***

For reference, the Semester 3 Final Sprint document’s text can be found here:

Project One

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 postgres 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 build, 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/>

**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 if you like)

**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 in either postgres, mongo, or both (you choose). These keywords should be save with datetime stamp and user\_id.

**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>

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 I know will work well for this project is <https://www.mockaroo.com/>but 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. 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.

Graphical user interface, text, application, email

Description automatically generated

*Figure 1 - Search Page*

Text

Description automatically generated

*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 nunjucks templates 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](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');

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 my example implementation I saved records for this to both databases, but you can choose one or the other – we’ll want to run these insertions on our database(s) every time the user sends a search query, so we’ll want to make sure that’s in our code for that 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 Two

This project must be completed individually. The central idea for this project is that we’re going to create a service that takes, as a request parameter, an array of numbers, and returns as a response a JSON representation of a binary search tree featuring the numbers from the array. The summary for this project is below:

**Notes:**

* **No need to build web UI, can be a Java command line interface if you have time.**
* **You can cut out the requirement to store data in a DB, you can store it in memory or save it to a file. See:** [**https://kodejava.org/how-to-read-and-write-java-object-to-json-file/**](https://kodejava.org/how-to-read-and-write-java-object-to-json-file/) **or** [**https://www.tutorialspoint.com/how-to-convert-java-object-to-json-using-jackson-library#:~:text=Converting%20Java%20object%20to%20JSON&text=In%20it%2C%20create%20an%20object,and%20print%20the%20obtained%20JSON**](https://www.tutorialspoint.com/how-to-convert-java-object-to-json-using-jackson-library#:~:text=Converting%20Java%20object%20to%20JSON&text=In%20it%2C%20create%20an%20object,and%20print%20the%20obtained%20JSON)**.**
* **Units Tests are required over CLI**

**Step 1.** Create an interface where the user can enter a series of numbers.

**Step 2.** Create an express application that can service a request containing that series of numbers.

**Step 3.** Have your express service take those numbers, and insert them, one by one, into a Binary Search Tree. Return a JSON representation of this binary search tree as your response.

**Step 4.** Your project **MUST** include at least three unit tests in it.

**Step** **5.** You **MUST** save the input and output data to a database (Of your choice?)

**BONUS:** Make the tree your return to the user a balanced binary search tree.

So, the plan for this project is that we’re going to create a simple node/express application that has at least three routes.

The first route is going to serve some html out to the client to give them a simple interface to use, where they can enter an array of numbers *somehow*, a submit button to submit them to the second route for processing, and a show previous button which will show previous trees saved in our database.

The second route is going to take the list of numbers, and it is going to generate a binary search tree from them, by inserting them, one at a time, into the tree – it should then return this tree, in a JSON representation, as it’s response. In figure 3 below I show the output of the first route, as you can see, it is a simple interface to allow the user to enter some data, in figure 4 I show what the response JSON looks like for that input. Lastly the inputted numbers and outputted tree should be saved to a database of your choosing, which is where the third route comes in.

The third route is simply going to spit the data from your database to the webpage. This can just simply be the input array/output of the previously processed trees.

You may wish to add a homepage or improve the aesthetics of this project (particularly if you want it in your portfolio), but at a minimum that is all that is required, three simple routes, one that shows an interface, one that gives the computed results and one that shows the previously computed results.

Note: The tree that I show in my example here is self-balancing, however, you **do not** have to create a balanced version of the tree, any output that is technically a binary search tree (that is, the value of a given node’s left children are always less than its own value, and the values of its right children are always greater than its value) will satisfy the minimum requirements for this project. However, it is considered a bonus objective to have the tree be self-balancing. One final thing, you **must** include at least three unit tests for this project. I’d recommend using the Jest testing framework. You can have the tests cover whatever you’d like, but on obvious thing would be your tree insertion function.

Graphical user interface, text, application, email

Description automatically generated

*Figure 3 - Input Interface Page*

Text

Description automatically generated  
*Figure 4 - Tree Result Page*