Serial Number Database Final Write-Up

By:

Jacob Byrnes,

Jack Campanale,

Ndenda Fierro Mutsaku,

Thomas Cole Varney

Submitted To: Professor V.J. Manzo

Date: 10/12/2022

Introduction:

Our goals for this term were to clean up the current website to make it easier for future development, bolster user profile functionality, make additional UI/UX improvements, and increase overall maintainability and legitimacy as a proof of concept moving forward.

Migration to React and Spring Boot:

Starting this project, we were faced with the challenge of where to start. The last group left a functional website, but there was a need for further understanding of how it worked amongst our group. The front end made use of an HTML framework called PUG which was unknown to all our group members. The back end used Python, which we were all familiar with, but was in need of documentation as we were unsure of what did what. In our first couple of meetings, we decided that if the Serial Number Database was to move forward in future development, it needed to be reworked. In order to do this, we began a migration of this project away from a PUG and Python stack to a React and Spring Boot stack. React contains a massive library of developer-friendly features like components, hooks, react-router, etc. This would prove to not only make it easier for us to recreate the website, but to also implement improvements in how easy it is to follow. Spring Boot allowed us to step away from using an Express.js server on the front end, and instead utilize RESTful programming to connect with the Backend. This engenders a more robust and maintainable backend environment that will find more success withstanding the test of time with changing industry standards. Migrating to a RESTful Spring Boot microservice also creates an environment of easy updating, fixing, and adding of new or existing functionality. Through this migration, we were able to produce a recreation of what we were presented before that we believe to be more easily legible and includes improved documentation.

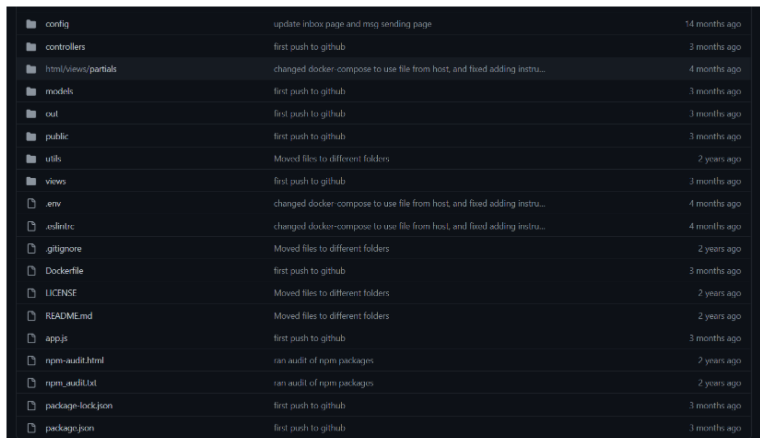

Figure 1: Front End File Structure Before

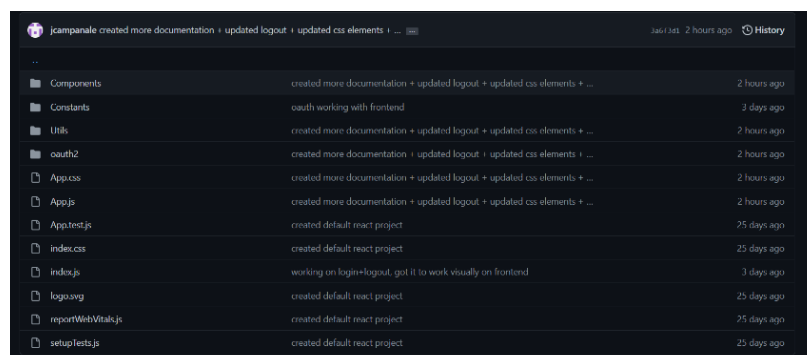

Figure 2: Front End File Structure After

React Functionality:

We created the starting React project using the [create-react-app method]() in the terminal. This gave us a good boiler plate to start with. From here, we wanted to make sure to utilize React's capabilities as much as possible in our recreation. To make routing between different pages easier, we used React's built in routing system. This allowed us to specify a route that links to a file. For example, the root route "/" links to the Home.jsx file in the /src/Components folder.

```
<Router>
  <Routes>
    {/* Set Home page (dashboard) as the index route */}
    <Route path='/' element={<Home/>}/>
    {/* Set Check page with route /check */}
    <Route path='/check' element={<Check/>}/>
    {/* Set Login page with route /login */}
    <Route path='/login' element={<Login/>}/>
    {/* Set SignUp page with route /signup */}
    <Route path='/signup' element={<Signup/>}/>
    {/* Set SignUpCompany page with route /signup/company */}
    <Route path='/signup/company' element={<SignupCompany/>}/>
    {/* Set ForgotPassword page with route /forgot */}
    <Route path='/forgot' element={<ForgotPassword/>}/>
    {/* Set Updateuser page with route /updateuser */}
    <Route path='/updateuser' element={<Updateuser/>}/>
    {/* Set Addinstrument page with route /addinstrument */}
    <Route path='/addinstrument' element={<Addinstrument/>}/>
    {/* Set Updateinstrument page with route /update/:id */}
    <Route path='/update'>
      {/* This is called a slug --> allows for dynamically changing urls (in this case :id is set to instrument serial number) */
      <Route path=':id' element={<Updateinstrument/>}/>
    </Route>
    {/* Set RedirectHandler with route /oauth2/redirect --> redirecthandler used to deal with redirecting back to home page once
    <Route path='/oauth2/redirect' element={<RedirectHandler/>}/>
  </Routes>
</Router>
```

Figure 3: React Router in App.js

Components allow developers to split the User Interface into smaller, more digestible, reusable chunks. We employed the use of Components throughout the development process. All the current pages a user can be directed to are Components. Furthermore, UI elements on these pages that might be reusable as well as more easily read on a separate page, such as an Instrument or a User Profile, are their own Components.

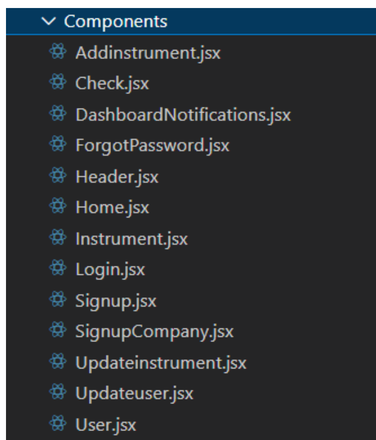

Figure 4: Component File Structure

In React, there are two types of Components: Functional and Class. We made use of Functional Components throughout the project as, stated [here](), Class Components can get confusing to read and write. Functional Components allow for the usage of React Hooks which were integral to our OAuth Implementation. The useState hook gave us an equivalence to a constructor in a Class Component, which we made use of when setting the current user and authentication in OAuth. The useEffect hook calls some code whenever a change is made to the React DOM which we used to call a load user function when the OAuth login succeeded.

Through these features of React, we were able to recreate the front end perfectly while utilizing well known and helpful features of React. Note that a wireframe of the frontend can be found [here]().

```
// Set state values (used during oauth)
const [authenticated, setAuthenticated] = React.useState(false)
const [currUser, setCurrUser] = React.useState(null)
const [loading, setLoading] = React.useState(false)
```

Figure 5: React useState Hook Usage in App.js

```
React.useEffect(() => {
  loadCurrUser();
}, [])
```

Figure 6: React useEffect Hook Usage

Spring Boot Functionality:

Our backend is built using the REST (Representational State Transfer) microservice architecture. How this works is, our backend sits on a server port and exposes a set of REST endpoints that the front-end service can reach through the http protocol. The front end sends a request to an endpoint, the backend server processes the request and sends a response back with the result of the process, which the front end will then use to repopulate the information portrayed to the user.

Control Flow Diagram:

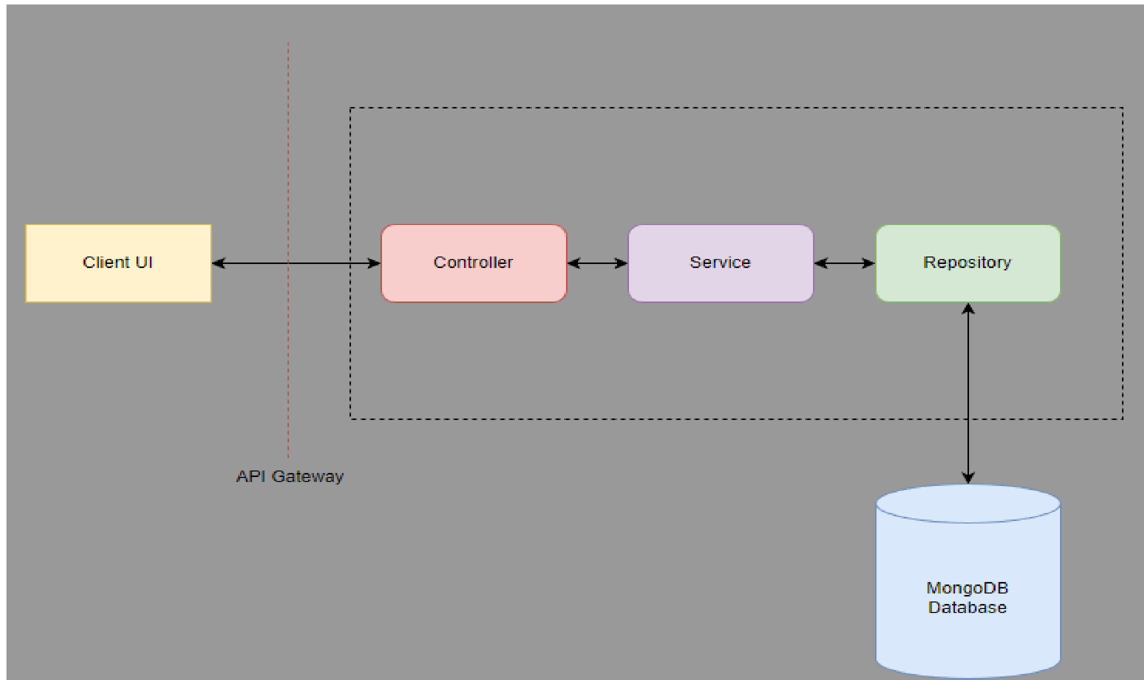

Figure 7: Control Flow Diagram

Improvements to Previous System:

.gitignore	added .vscode/ to .gitignore	2 years ago
Client.py	first push to github	3 months ago
Crawler_Manager.py	first push to github	3 months ago
Dockerfile	first push to github	3 months ago
Instrument_DB.py	first push to github	3 months ago
Job_Cleaner.py	first push to github	3 months ago
Job_Manager.py	first push to github	3 months ago
Server_Manager.py	first push to github	3 months ago
ebayCrawler.py	first push to github	3 months ago
ebayScraper.py	first push to github	3 months ago
requirements.txt	Service: completely cleaned up python requirements	2 years ago
sndb_service.dockerfile	Service: containerized with Docker	2 years ago
workflow.png	Moved files to different folders	2 years ago

Figure 8: Previous Backend Structure

..		
configuration	Oauth working with sample frontend	5 days ago
controller	Merge pull request #1 from varneycole00/oauth	30 minutes ago
entity	Merge pull request #1 from varneycole00/oauth	30 minutes ago
errors	Oauth working with sample frontend	5 days ago
repository	Oauth working with sample frontend	5 days ago
response	Initial commit	27 days ago
security	Oauth working with sample frontend	5 days ago
service	Oauth working with sample frontend	5 days ago
serviceimpl	Oauth working with sample frontend	5 days ago
utility	Merge pull request #1 from varneycole00/oauth	30 minutes ago
SndbApplication.java	Initial commit	27 days ago
postman.http	oops i forgot	5 days ago

Figure 9: Current Backend Structure

OAuth2 Integration

Instead of building out a user account system within our 7-week time frame, we opted to instead use OAuth2 for our user data management and authorization. Currently, the application supports signing in with a user's google account with the scope allowing us to see their profile and their email address. When a user signs in with google for the first time, there will be an entry added into our user table in the database where we could in the future keep track of more information if need be. As far as instruments are concerned, they are linked to the user themselves through an owner field in the instrument database where their google email address will be tagged. This allows us to find instruments owned by a single user.

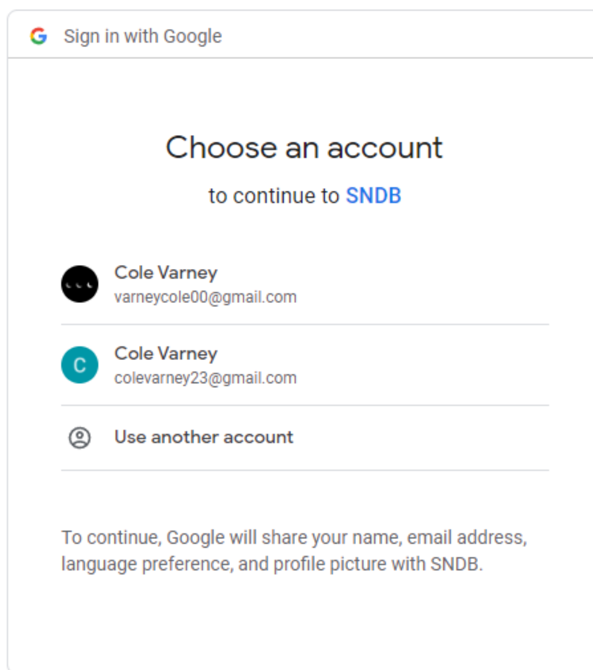

Figure 10: Google OAuth Redirect Page

Signing in with OAuth will then allow users to see their profile and dashboard, as well as introduce the ability to add their instruments to the site.

AWS Integration for Email Notification

We decided to use AWS Simple Email Service (SES) as our email service to notify users about instrument checks and to allow users to change their password if they forget it. AWS and Gmail accounts (both use email: serialnumberdatabase22@gmail.com password: SNDB_rework22) were created for the project. An AWS Lambda function was created that can call AWS SES and send emails to users. The function was then linked to an API Gateway so that our backend code could connect to AWS without requiring AWS libraries to be installed locally.

sendRecoveryEmail

▼ **Function overview** Info

sendRecoveryEmail

Layers (0)

API Gateway

+ Add trigger

APIs (1)

Find APIs

	Name	Description	ID	Protocol
○	Email		gbx9o9vdoa	REST

The Lambda function calls were linked into our backend so that they can be called by the front-end using REST API calls the same way it calls all other back-end functions.

```
jtbyrnes-wpi
@GetMapping("/sendPasswordRecoveryEmail")
public ResponseEntity<SNDBResponse> sendPasswordEmail(@RequestBody Email email) throws IOException {
    EmailUtils e = new EmailUtils();
    e.sendEmail(email.getAddress(), email.getTitle(), email.getBody());
    SNDBResponse response = new SNDBResponse( message: "Email Sent",  statusCode: "200");
    return new ResponseEntity<>(response, HttpStatus.OK);
}

jtbyrnes-wpi
@GetMapping("/sendNotificationEmail")
public ResponseEntity<SNDBResponse> sendNotificationEmail(@RequestBody Email email) throws IOException {
    EmailUtils e = new EmailUtils();
    e.sendEmail(email.getAddress(), email.getTitle(), email.getBody());
    SNDBResponse response = new SNDBResponse( message: "Email Sent",  statusCode: "200");
    return new ResponseEntity<>(response, HttpStatus.OK);
}
```

Future Work:

We completed most of our goals by the end of the project, by completing the migration to React and Spring boot and adding the OAuth login and AWS email setup. Some of the steps that need to be taken for future work are:

- Functional image uploading for adding instruments
  - Future teams should look into utilizing GridFS with the mongoDB
- Pull users current instruments from backend
  - We push instruments to the database with a field “owner” being their Gmail. This should streamline the process of pulling a user’s instruments and populating the dashboard on creation
- Functioning notifications that add to user dashboard and email them
  - Our email service is set up in AWS with a functioning URL endpoint in our backend code. Notifications should go through to users when their instruments are “Checked”. This should be as easy as when an instrument is checked, take its user field (email) and send both a notification and an email using their respective URL endpoints.
- Web Crawlers and Scrapers
  - Look into current APIs for eBay/Craigslist/etc. Also perform research on rules for crawling and scraping for specific sites as some may not want to be crawled/scraped
  - This could work with email/notification service if this get’s set up. Once a crawler finds an instrument with a matching serial number, notify user.
- Find place to host front end and back end
  - These run separately from one another, need to find places to host both