# Project Report

Bookeroo is a site for members, by people who wanted the site when they were users themselves. The vision is simple, an easy to access, lightweight, online retailer and community for book worms and casual readers alike. Currently, on the market are just general book retailers, whose main focus is selling books. Goodreads, which is a community for book readers and second-hand sale websites like Gumtree. The vision is to combine all those features into a one-stop-platform for our members to enjoy.

The vision was to create:

- A simple, intuitive interface that any user can use, making sure all skills and abilities with the online world are cared for
- Ensure that a wide variety of users are catered for with varying genres, languages and book formats
- Ensure that it also varies from other sellers through the ability to list your own books for sale, which allows users a flexibility not seen before, through integrating a facebook marketplace/gumtree feature - users can make this their one-stop platform, to not only buy affordable books but sell ones they've before read and either enjoyed, or didn't.
- Leave meaningful reviews for both the user and publisher benefit. With intuitive input and easy to view.
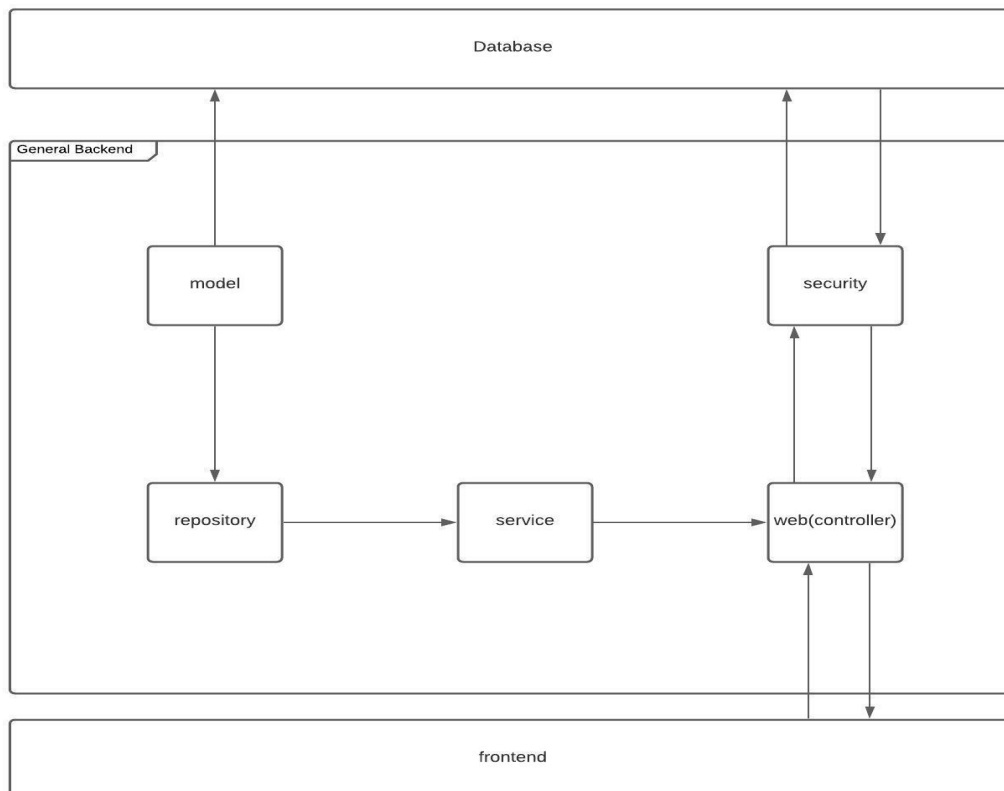
Overall Structure:
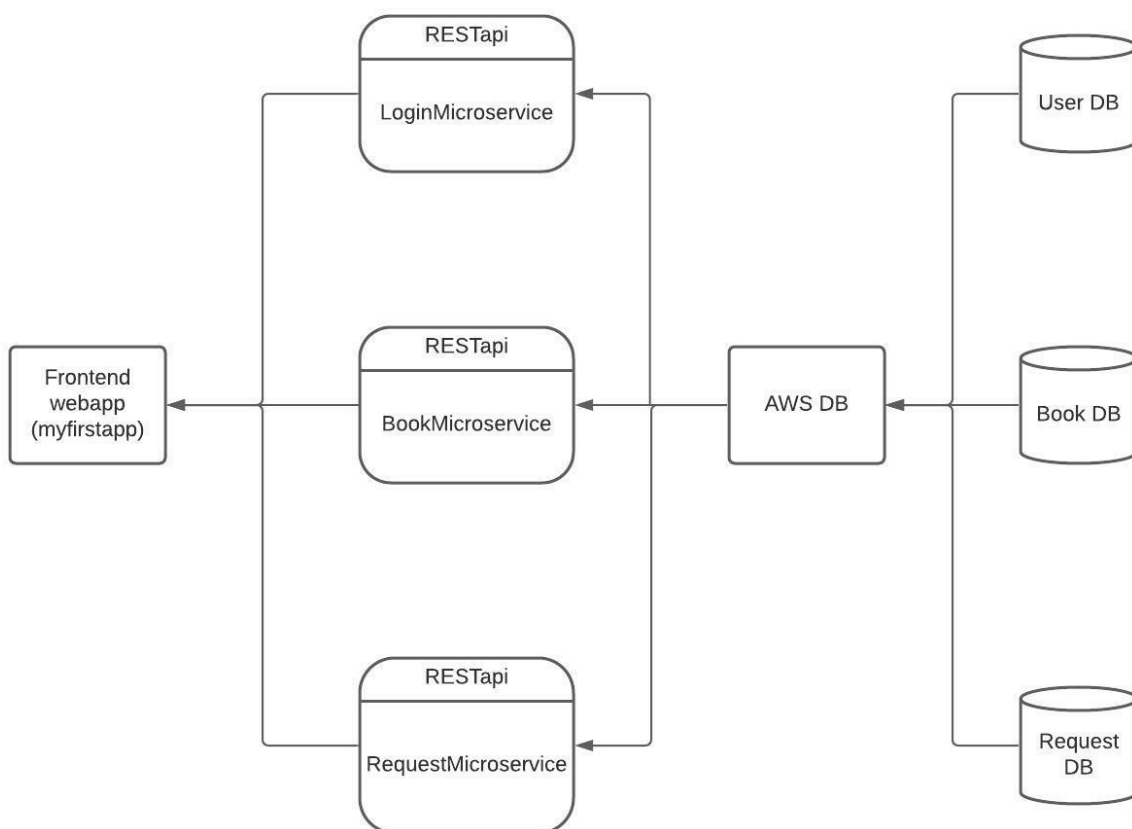
Front end:
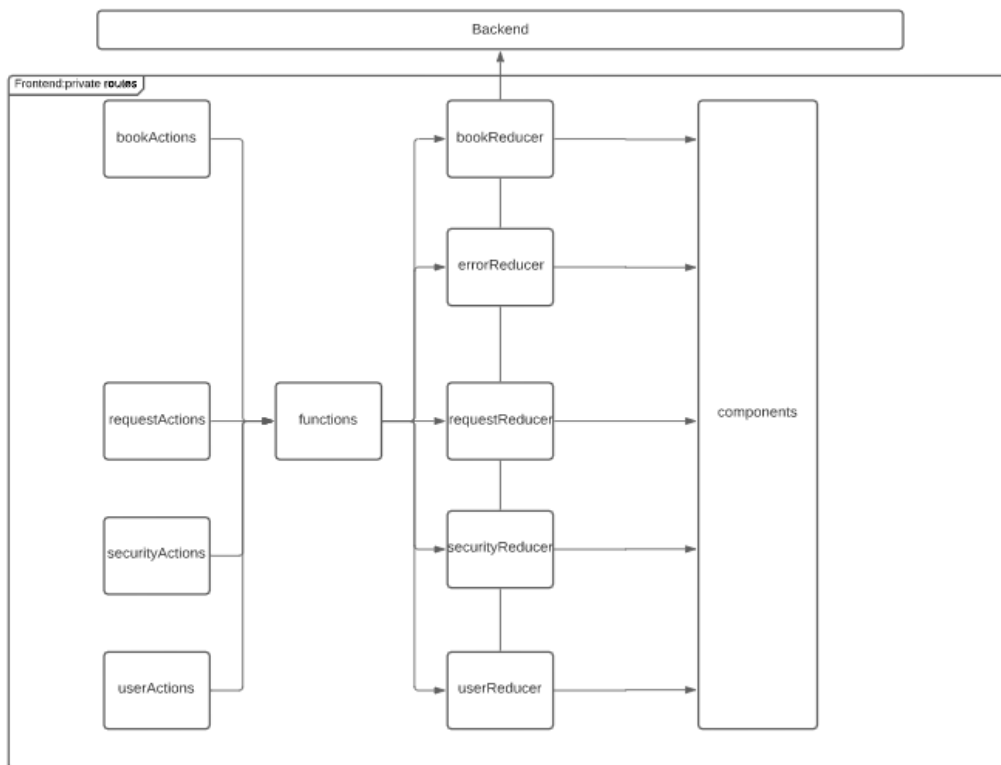
Backend:

Code Overview: Major milestone 3

This document will cover the work and code finished and unfinished in milestone 3: sprint 2 and 3, as well as cover the difficulties included with implementing it and the barriers that needed to be overcome in order to complete the user stories that were set during sprint planning or included as the scope of the sprint evolved.

Carry over tasks from previous milestones:
We improved the user profile formatting, which had little to no formatting,it now looks like so:
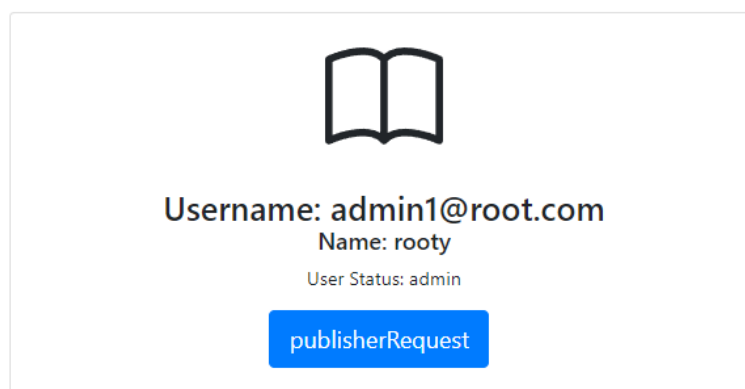
Major implementation: Book microservice (the first new microservice)
One of the major questions that was posed to us in the implementation of the book microservice was how to work on a microservice collaboratively, what could be implemented in the front end before a backend had been made and how to do so in a conducive manner that wouldn't result in multiple rewrites. Due to having been given the previous login microservice in the previous iteration, this process would require some tweaks as we proceeded as the groundwork needed to be laid in order to know what we were dealing with. That being said, given our very limited history with react js formatting and spring boot variations, what was presented forward facing looks the best we could present with our current skill set as a group.
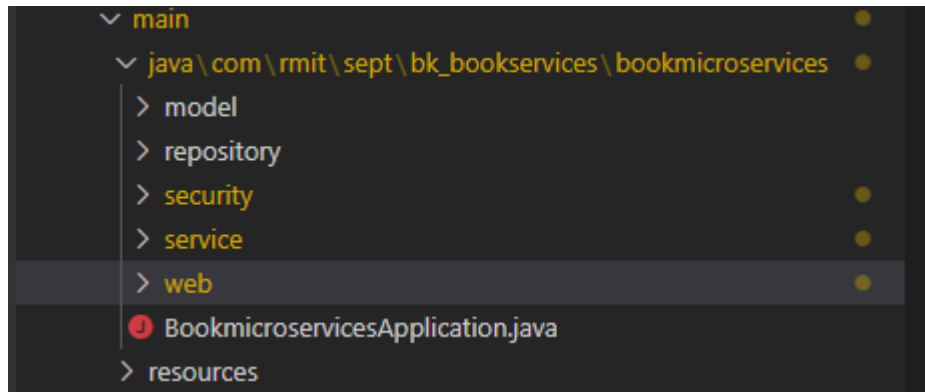
Using resources and tutorials from:
https://www.baeldung.com/spring-data-jpa-pagination-sorting
https://spring.io/guides/gs/handling-form-submission/
https://www.baeldung.com/spring-mvc-form-tutorial

As well as examples given to us through the login microservice, we were presented with the breakdown of folders needed as so:

Model represents the sql schema in our case, breaking down what information about the model (in this case: the book) is needed

The repository hosted the crudrepository: Spring Data interface for generic CRUD operations on a repository of a specific type. It provides default types to interact with the SQL

Security was mostly unused, although implemented with the spring security.

Service provides a repository of custom methods that called the repository interface

The web hosted the controllers which called the service and repository methods of which to interact with the frontend.

What is unique to the book microservice was the implementation of pageables, and several other plugins not present in other microservices, that being said, the services required the implementation of additional java classes that were outside the purview of what we're taught. Thus as we used tutorials to cover the skill difference, there is a chance that the practices engaged in may seem strange in context. That being said, what was presented, we believe, was worth the strange implementation within the purview of this project, but perhaps not so in a commercial product.

Breaking down the user stories that were worked on, on this milestone includes the following for the book microservice:

As a user, I want to search for book titles so that I can rate it and write a review

As a customer, I want to view a books details, so that I can get a better understanding of it before I purchase it.

As a user, I want to search for book titles so that I can get information on a specific books

As a user, I want to view the books table of contents, so that I can gauge whether its suitable for me

As a user, I want to search up authors so that I can buy more books written by that author

As an admin, I want to be able to upload a books details, so that customers can view it

As an unregistered website user I want to register as a publisher user so that I can add new titles

From these user stories, we identified the need for a book profile page, a way to search the book and a way to find the book and a way for publishers and admins to add a book.

There were initial attempts of searching using processed statements but we quickly found that such commands were extremely awkward to use in the context of relating it to react.js. However we found an extremely simple solution, that being the use of a jpa query:

```
@Repository
public interface BookRepository extends PagingAndSortingRepository<Book, Long>{

    @Query("FROM Book b WHERE b.title LIKE %:searchText% OR b.author LIKE %:searchText% OR b.language LIKE %:searchText% OR b.genre LIKE %:searchText% ORDER BY b.price ASC")
    Page<Book> findAllBooks(Pageable pageable, @Param("searchText") String searchText);
}
```

This in turn, when linked with the controller allowed us a simple call and receive message based on the sql command library. Essentially, this command fulfilled all the requirements for the respective search related user stories.

Book List							mandarin
Title	Author	ISBN Number	Price	Language	Genre	Actions	
adsasdasd1	asdasdasfasdasfas	121324142312	32132	Mandarin	Fantasy		

Showing Page 1 of 1   First   Prev   1   Next   Last

```
<Card className={"border border-white bg-white text-black"}>
  <Card.Header>
    <div style={{ float: "left" }}>
      <FontAwesomeIcon icon={faList} /> Book List
    </div>
    <div style={{ float: "right" }}>
      <InputGroup size="sm">
        <FormControl
          placeholder="Search"
          name="search"
          value={search}
          className={"info-border bg-white text-black"}
          onChange={this.searchChange}
        />
        <InputGroup.Append>
          <Button
            size="sm"
            variant="outline-info"
            type="button"
            onClick={this.searchData}
          >
            <FontAwesomeIcon icon={faSearch} />
          </Button>
          <Button
            size="sm"
            variant="outline-danger"
            type="button"
            onClick={this.cancelSearch}
          >
```

```
searchChange = (event) => {
  this.setState({
    [event.target.name]: event.target.value,
  });
};

cancelSearch = () => {
  this.setState({ search: "" });
  this.findAllBooks(this.state.currentPage);
};

searchData = (currentPage) => {
  currentPage -= 1;
  axios
    .get(
      "http://localhost:8081/api/books/search/" +
        this.state.search +
        "?page=" +
        currentPage +
        "&size=" +
        this.state.booksPerPage
    )
```

Due to my limited experience with react, i had to make a shortcut axios.get within the component file, which may be bad practice.

From the book controller:

```java
@RequestMapping("/api/books")
public class BookController implements Resource<Book>{

    @Autowired
    private IService<Book> bookService;

    @Autowired
    private IPageService<Book> bookPageService;

    @Override
    public ResponseEntity<Page<Book>> findAll(Pageable pageable, String searchText) {
        return new ResponseEntity<>(bookPageService.findAll(pageable, searchText), HttpStatus.OK);
    }

    @Override
    public ResponseEntity<Page<Book>> findAll(int pageNumber, int pageSize, String sortBy, String sortDir) {
        return new ResponseEntity<>(bookPageService.findAll(
                PageRequest.of(
                        pageNumber, pageSize,
                        sortDir.equalsIgnoreCase("asc") ? Sort.by(sortBy).ascending() : Sort.by(sortBy).descending()
                )
        ), HttpStatus.OK);
    }

    @Override
    public ResponseEntity<Book> findById(Long id) {
        return new ResponseEntity<>(bookService.findById(id).get(), HttpStatus.OK);
    }

    @Override
    public ResponseEntity<Book> save(Book book) {
        return new ResponseEntity<>(bookService.saveOrUpdate(book), HttpStatus.CREATED);
    }

    @Override
    public ResponseEntity<Book> update(Book book) {
        return new ResponseEntity<>(bookService.saveOrUpdate(book), HttpStatus.OK);
    }

    @Override
    public ResponseEntity<String> deleteById(Long id) {
        return new ResponseEntity<>(bookService.deleteById(id), HttpStatus.OK);
    }
}
```

```java
public interface Resource<T> {
    @GetMapping("/search/{searchText}")
    ResponseEntity<Page<T>> findAll(Pageable pageable, @PathVariable String searchText);

    @GetMapping
    ResponseEntity<Page<T>> findAll(int pageNumber, int pageSize, String sortBy, String sortDir);

    @GetMapping("{id}")
    ResponseEntity<T> findById(@PathVariable Long id);

    @PostMapping(consumes = MediaType.APPLICATION_JSON_UTF8_VALUE)
    ResponseEntity<T> save(@RequestBody T t);

    @PutMapping(consumes = MediaType.APPLICATION_JSON_UTF8_VALUE)
    ResponseEntity<T> update(@RequestBody T t);

    @DeleteMapping("{id}")
    ResponseEntity<String> deleteById(@PathVariable Long id);
}
```

Using the resource interface to declare the types of mapping and the controller, methods available begin to take shape, giving an initial idea of how a front end developer would begin using these methods.

To note, as this was our first from scratch microservice implementation, we also became somewhat familiar with the order of writing the first end. From that we entered the order of the actions -> reducers -> components (from a template that was written from one of our front end developers).

```javascript
import * as BT from "./types";
import axios from "axios";

export const saveBook = (book) => {
  return (dispatch) => {
    dispatch({
      type: BT.SAVE_BOOK_REQUEST,
    });
    axios
      .post("http://localhost:8081/api/books", book)
      .then((response) => {
        dispatch(bookSuccess(response.data));
        alert("save succesful")
      })
      .catch((error) => {
        dispatch(bookFailure(error));
      });
  };
};

export const fetchBook = (bookId) => {
  return (dispatch) => {
    axios
      .get("http://localhost:8081/api/books/" + bookId)
      .then((response) => {
        dispatch(bookSuccess(response.data));
      })
      .catch((error) => {
        dispatch(bookFailure(error));
      });
  };
};
```

As part of learning of the implementation process of backend to front end, we identified the axios.[mapping] to be the important part of the front end that takes in the information that's created for the localhost at the respective 808x port and the axios mapping methods being in charge of what to do with the information.

A Method we had to incorporate in this unit was the inclusion of multiple react functions on a single page, in this case referring specifically to booklist, which allowed for redirection to edit a page, delete a page and redirect to the pages profile. Of course, that being said, editing a books page and deleting a page was restricted to an admin usertype:

Admin user:
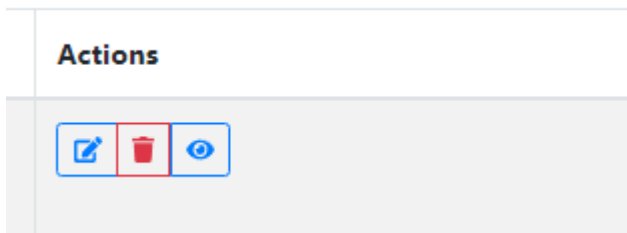
Regular user:
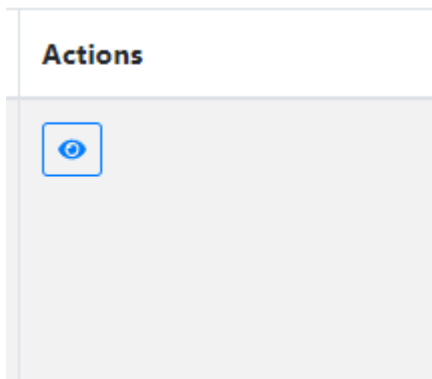
Which is represented in code like so:

```
<td>
  <ButtonGroup>
    {user.userStatus === "admin" &&
      <>
        <Link
          to={"edit/" + book.id}
          className="btn btn-sm btn-outline-primary"
        >
          <FontAwesomeIcon icon={faEdit} />
        </Link>{" "}
        <Button
          size="sm"
          variant="outline-danger"
          onClick={() => this.deleteBook(book.id)}
        >
          <FontAwesomeIcon icon={faTrash} />
        </Button>
      </>
    }
    <Link
      to={"BookProfile/" + book.id}
      className="btn btn-sm btn-outline-primary"
    >
      <FontAwesomeIcon icon={faEye} />
```

Using store as a way to determine the user type to know what icons to display.

Another implementation of note is the ability to add books in a form:

The data taken was made to match specs given to us in the assignment specification sheet, that being said, the act of writing in the form was very cumbersome. Furthermore, things like authors being unique weren't taken into consideration. If given enough time for another sprint, it could potentially be dealt with in a future iteration, however what we presented in this case was done so for brevity and to meet the deadlines/requirements of the project.Thus at this time, authors for each books when put into a profile as labeled as is, which is to say, each author is an individual entity with not connection to the authors with the same name (although semantically you could argue that by searching an author's exact name, they do count as the same person).

A book profile is shown as so:

Although a rudimentary paypal api was used, it's not fully implemented at this stage, and if given more time for a sprint, a cart system could be added.

Major implementation: Request microservice

Now we had a service to peddle, we needed more differentiation between the types of users on the website from which was established in sprint 1:
Admins and publishers are able to publish books onto the website
Users are not.

However there needed to be a system in which a user was able to suggest books, or in general contact admins if needed help, the user stories that related to this implementation are:

As a user, I want to contact the admin team, so that I can ask questions about signing up.

Furthermore, during this sprint we decided there should be some differentiation towards users being able to add books to the database. Thus the request acts as a suggestion box to add new books as well as general admin contact concerns. It's reachable from the footer and in the header as a header option.

Bookaroo   book list   suggest book

About Us	Contact Us	
Aim	Support	C. 2021 Bookaroo Ltd. All Rigths Reserved. "Bookeroo" is a trademark of Bookeroo Ltd.
Background	Connect	Activate Window

Contact Us!

If you have any issues, want to suggest a book for us to stock or anything else - submit a request below!

Alternatively, send through an email to the address below.

support@bookaroo.com.au

Request Title

Request Information

Email

Submit Request

Another distinction for the user story, is that even a non logged in user would also be able to contact the admin team, but the admin team would know if the query was from a currently logged in account.
Here is a screenshot of the admin view page:

The backend uses a standard structure learned from the book microservice and the login microservice:
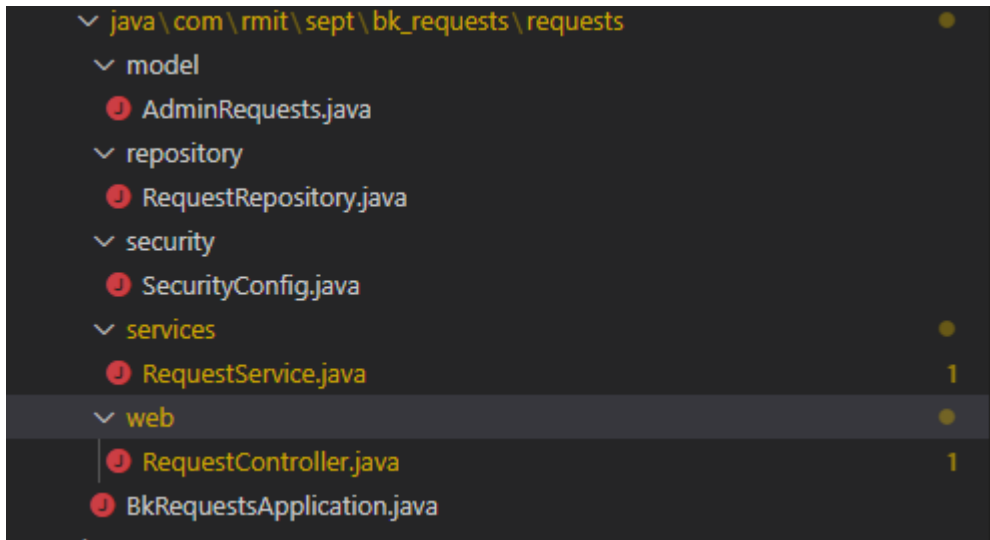
The structure is essentially the same, but the requirements are simpler than both the book and login. Thus this is the simplest implementation required to make a functioning backend, not requiring special accountremonts outside the security config.

On Top of that, given we needed more functionality for an admin, the admin is able to view request on a list like so:

id	user	title	request comment	email
180	vicarootting	ofudhvbvoiudfssf8sud0	sdnf	hey@ouefh.com
181	vicarootting	ofudhvbvoiudfssf8sud0	hey this is sort of a long request hey this is sort of a long request hey this is sort of a long request hey this is sort of a long request hey this is sort of a long request hey this is sort of a long request	hey@ouefh.com
182	vicarootting	hey	heyeyheyheyehehyh	email

AWS MYSQL:

```
1    server.port=8082
2
3    spring.datasource.url=jdbc:mysql://a-sep-up.chcksl92xamp.ap-southeast-2.rds.amazonaws.com:3306/uniwork
4    spring.datasource.username=admin
5    spring.datasource.password=aslightsepup
6    spring.jpa.hibernate.ddl-auto=create
7    spring.security.user.name=root
8    spring.security.user.password=root
9    spring.jpa.show-sql=true
10   spring.jpa.properties.hibernate.globally_quoted_identifiers=true
11   spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
12   spring.datasource.driver-class-name=com.mysql.jdbc.Driver
13   spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
```

The transition to the amazon database ended up being overall a simple one, however we stumbled on very little minute mistakes that made us believe we weren't making as much progress as we were. Declaring the "3306" into a "3306/uniwork" in itself created issues and ended up taking a day and the attention of 2 members to finally successfully connect to the AWS server. That being said, after the first connection was made, the proceeding connections were simple and easily applicable to other application configs.

We used the mysql workbench to connect and check connections in our SQL databases, we created the connection as shown below:

Paypal API:
We had an extremely rudimentary implementation of the paypal api in our project. Luckily the paypal api proved to be extremely developer friendly and quite simple to implement based on the documentation. However as we didn't want to spend money to test it, the paypal implementation doesn't receive any real information.

What was needed to implement the paypal api is as follows: the module from the paypal developer website under the name:@paypal/react-paypal-js

Then creating a line in the app.js file like so:

```
<PayPalScriptProvider options={{ "client-id": "AZNkf83ly9F9U4wjt_1FcORPFEaBD2cEJaQelNZqF4KvjwVkQZrkyKiXhAwjgyrMEyl1nYZQ6QghrQPr" }}>
```

After this was done, we needed to apply it to a page. At this point in our project, we had a book with some monetary value to it, but we didn't have a **cart** implementation at this stage. So as a placeholder function, the book page instead had just a purchase button/buy it now implementation as shown here:

And represented in code as shown here:

```
<div class="container profile" >
    <div class = "column">
        <div class="ratings d-flex flex-row align-items-center">
            <h6>Rating: {this.state.rating}/5</h6>
        </div>
        <div className="btn btn-lg btn-primary mr-2">
            <PayPalButtons style={{ layout: "horizontal" }} />
            Buy Now: ${this.state.price}
        </div>
    </div>
</div>
```

Limitations and errors:
Books implementation suffered from some code smelling that ended up breaking the code after several merges and not enough commenting. The editing of the book had to be rewritten as at one point, the state of the book would only take the previous book edit click: e.g. if i clicked on book id 3, nothing would show, and then if i clocked on book id 4, then book id 3 would show up. Thus Editbook.js was created as a bugfix, but the refactoring of the book.js function still needed to be pruned.

There were several framework implementations that were never built upon, the backend team mostly consisted of at most 2-3 people including testing, as such we could only work and implement at most 2 microservices per milestone. Thus considerations like customer

listings and carts weren't implemented but would have been the main focus of the next sprint.

Considerations like front end refactoring won't be perfect, due to our limited knowledge of js, at least in the case of complete REST api building. Redundancy may occur and functionality was prioritised

Scrum Process

The team met 2 times a week, once on Sunday night at 8pm and once before the SEPT tutorial time on Wednesday. The primary communication platform was Discord, although Slack and Teams were also used for files and other discussion. The scrum master role was split between the team, and everyone contributed to documentation, jira management, and trello updates.
The overall process relied heavily on having jira be constantly up to date and updated. This is due to the fact that we are coding from user stories defined in each sprint that we go through every 2 weeks. The sprint planning document is created before the sprint, where we plan the sprint itself and the user stories to cover, while the sprint retrospective reviews the sprint and we look back at improvements to be made, things that went well and overall review the sprint as a whole.

Deployment Pipeline Setup:
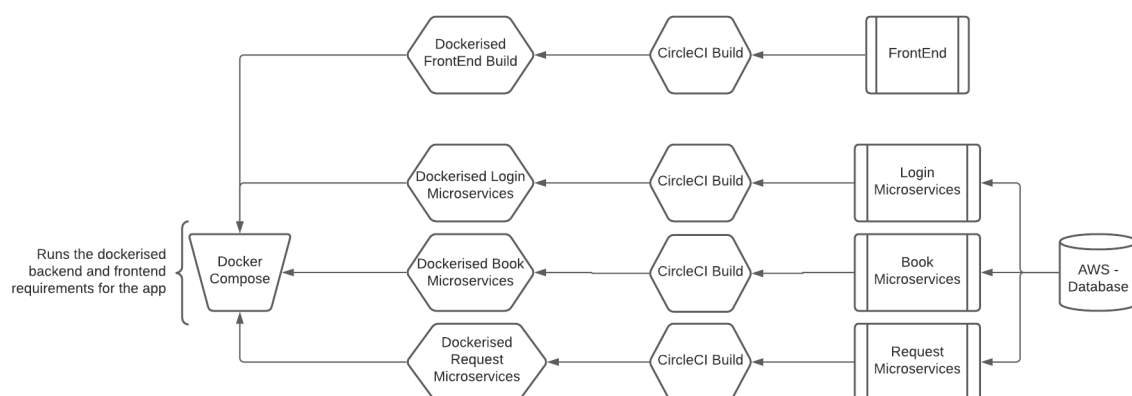
Our microservices are all built via a CircleCI setup to be Dockerized containers, which are updated after each commit on each branch within our repository. The docker compose is then called locally to launch the setup of all the dockerized microservices, and we are able to launch the app with the backend and frontend properly being connected.