COMP 3005 Project Report:

Participants: Aidan Bossart(101130220), Emad Ryan(101078668)

2.1

**ER Diagram**:

A close up of a map

Description automatically generated

**Assumptions**:

-Many publishers can publish many books

-Many books can be authored by many authors

-many books may be added to a single cart

-many books can be included in a single collection

-Each shipment will be made from an order that is formed from a cart that is based on the user

-Every owner/admin can read every report

2.2

**Reduced Relations**:

Author(author\_id(pk),name)

Book(book\_id(pk),author\_id(fk),name,ISBN,genre,pagenum,price,pub\_percent,stock,min\_stock,sold\_month,rating)

Writes(author\_id,book\_id)

Publisher(publisher\_id(pk))

Published By(book\_id,published\_id)

Owners(o\_id)

collection(book\_id,o\_id,name)

User(u\_id,name,username,password,cardnum(pk),cardtype(pk),expiry(pk),address,postalcode,country,province)

Shipment(shipment\_id(pk),courier\_id,status,deliveryAddress)

Cart(cart\_id(pk),book\_id(fk))

order(order\_id,cart\_id(fk),shipment\_id(fk),u\_id(fk),cardnum(pk),cardtype(pk),expiry(pk))

add\_to(book\_id,cart\_id)

writes(author\_id,book\_id)

reports(report\_id,total\_sales,total\_expenditures,sales\_genre,sales\_author,sales\_publisher)

2.3

**Normalization Decomposition**:

**-Author**: author\_id,name

author\_id -> name

This is already in normal form and can’t be reduced further.

**-Publisher**: publisher\_id,name,email,phone,banknum,postalcode,country,province,address

publisher\_id->name, email, phone, banknum, address, postalcode, country, province

address-> postalcode, country, province

This is not normalized as address cannot map to the complete set, therefore, we can degrade into:

Table 1: publisher\_id, name, address, email, phone, banknum

Table 2: address, postalcode, country, province

Natural joining on address allows for the original table to form again, and this is normalized

**-Book**: book\_id, name, author\_id, ISBN, genre, pagenum, price, pub\_percent, stock, min\_stock, sold\_month, rating

book\_id -> name, author\_id, ISBN, genre, pagenum, price, pub\_percent, stock, min\_stock, sold\_month, rating

name,author\_id -> ISBN, genre, pagenum

This is not normalized as name,author\_id cannot map to the full set. Therefore, we can decompose into:

Table 1: Book\_id, name, author\_id, price, pub\_percent, stock, min\_stock, sold\_month, rating

Table 2: name, author\_id, ISBN, genre, pagenum

Joining on the combination of name, and author\_id allows for the original table to reform, and it is currently normalized

**-Cart**: cart\_id, book\_id

cart\_id -> book\_id

This is already in normal form and can’t be reduced further.

**-Owner**: o\_id, u\_id

o\_id -> u\_id

This table is already in normal form and can’t be reduced further

**-Shipment**: shipment\_id, courier\_id, status, deliveryAddress, postalcode, country, province

Shipment\_id -> courier\_id, status, deliveryAddress, postalcode, country, province

deliveryAddress -> postalcode, country, province

courier\_id -> status

This is not in normal form due to courier\_ids & deliveryAddress’ closure not being the full closure of the table. Therefore, it can be reduced into:

Table 1: shipment\_id, courier\_id, deliveryAddress

Table 2: delivery\_address, postalcode, country, province

Table 3: courier\_id, status

**-User**: u\_id, name, username, password, paymentinformation, cardnum, cardtype, expiry, address, postalcode, country, province

u\_id -> name, username, password, paymentinformation, address

Paymentinformation -> cardnum, cardtype, expiry

Address -> postalcode, country, province

This is not in normal form and can be reduced into:

Table 1: u\_id -> name, username, password, paymentinformation, address

Table 2: Paymentinformation -> cardnum, cardtype, expiry

Table 3: Address -> postalcode, country, province

-**reports**: report\_id, total\_sales, total\_expenditures, sales\_genre, sales\_author, sales\_publisher

Report\_id -> total\_sales, total\_expenditures, sales\_genre, sales\_author, sales\_publisher

This is already normalized and cannot be reduced further

2.4

**Database Diagram**:

A screenshot of a social media post

Description automatically generated

2.5

Implementation:

-Architecture:

This is a web-based application built in angular, using maven and spring as a backend. It is written mainly in typescript, with java code being used in a modular fashion to add functionality to the website. We use PostgreSQL for our database, and use the postgres JDBC API to query it which provides us with result sets that we can use to populate our actual website.

Using maven and spring as the backend allows us to build our project very easily, as we can use it to generate our website template, and to manage our dependencies through a single file. For the purpose of this paper we can treat maven as hub for our website, that manages our different dependencies and allows us to develop our store in a very simple and readable format.

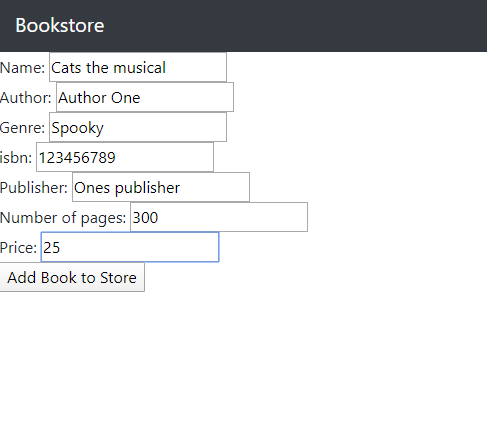
Each component that we use is separated into its own folder, where we can add functionality to the component in typescript, or even modify how the component looks in html or CSS code. Then, outside of these component folders is the general service classes, which allow the website to call on these components and use their functions with context provided from the site itself. Using these components and then the services in tandem allow for the website to function properly.

Outside of our component and service architecture we also have our java code which is used to query and interact with the database. We have a simple application properties file which configures our connection to the database, and then we have our bookstore controller file, which contains any and all queries our website would need to run on our database. These functions can be executed on GET or POST requests and will take context from the website to run the queries. An example case of this would be if we searched a book by name, the function would be called on the request from the user, and a connection would be opened to the database by leveraging the JDBC. From there, the result set will be converted into a JSON and returned to the site, so that the information can be displayed to the user.

Database query implementation can be observed in the BookstoreController class, and database connectivity parameters can be viewed in the application.properties class.

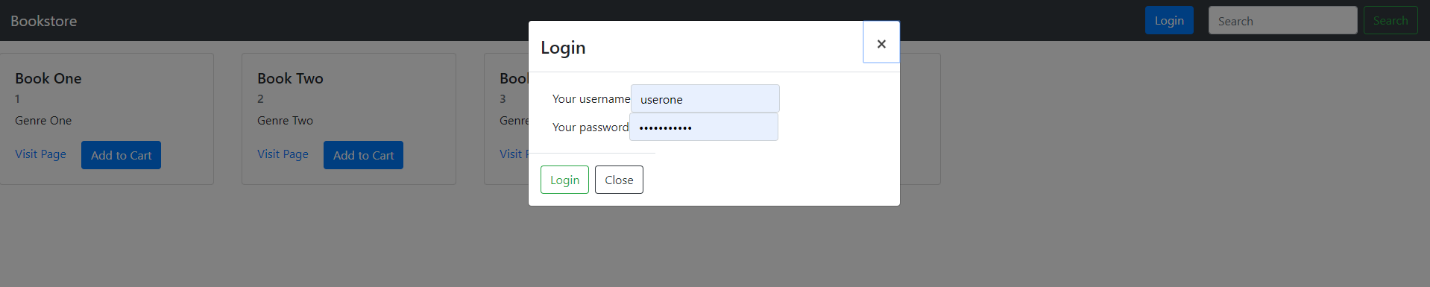
Example use cases:

**Admin page**:



We have an admin page where the user can add books to the bookstore. These trigger the JDBC to run executeUpdate functions, which add the book to the database.

**Logging in**:

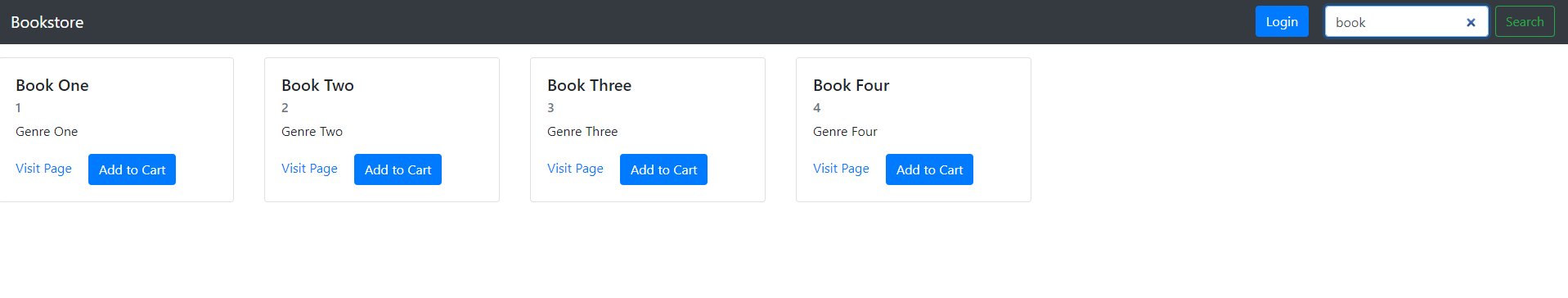


Here the user is prompted to log on to their bookstore account.



After the user successfully logs in the login button deactivates to show that the user has been successfully authenticated, by comparing their inputs to the tuples in the user table inside of the database.

**Searching for books**:



(searching by case “book”)

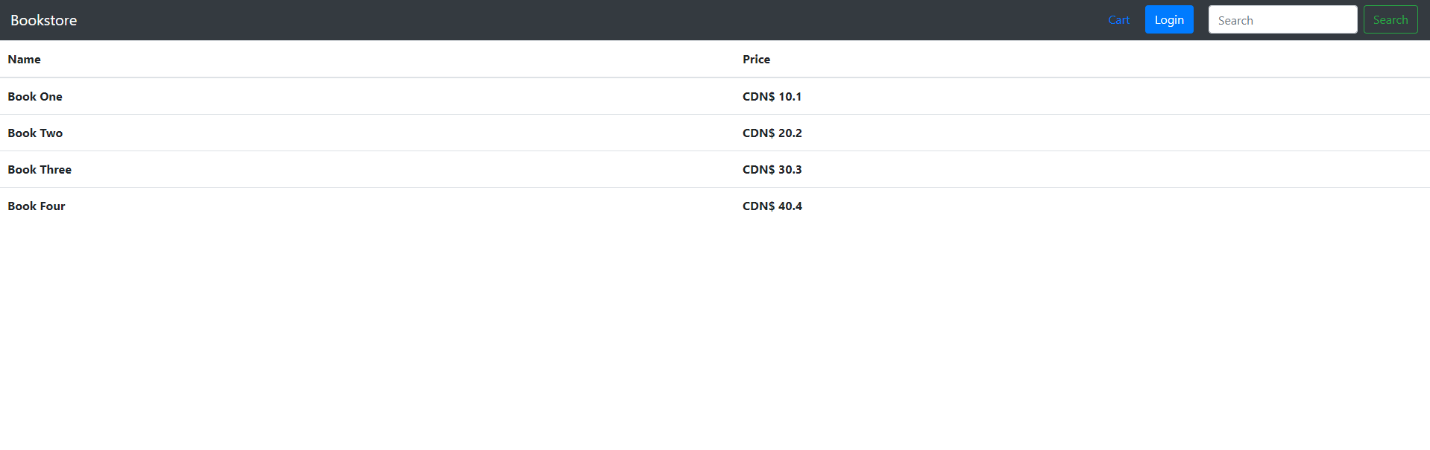


(searching by case “one”)

Here the user can search for books that they want to purchase. This demonstrates approximate matching as well, in that the user can search for partial cases, and the database will return all books that contain the case provided.

This is achieved by taking the search cases and comparing them to the database tables with the LIKE parameter.

Adding books to cart to purchase:



After browsing the collection of books after searching for some titles, the user can add as many books to their cart as they want before moving forward with purchasing them.

2.6

Bonus features:

Approximate match: Can search for a book by anything contained in the book name. This varies from a single letter to the whole title.

2.7

GitHub repository: https://github.com/aidanbossart/COMP3005-Project

2.8

Appendix:

April 16th 1:30 onwards.