



COMP 3005

PROJECT

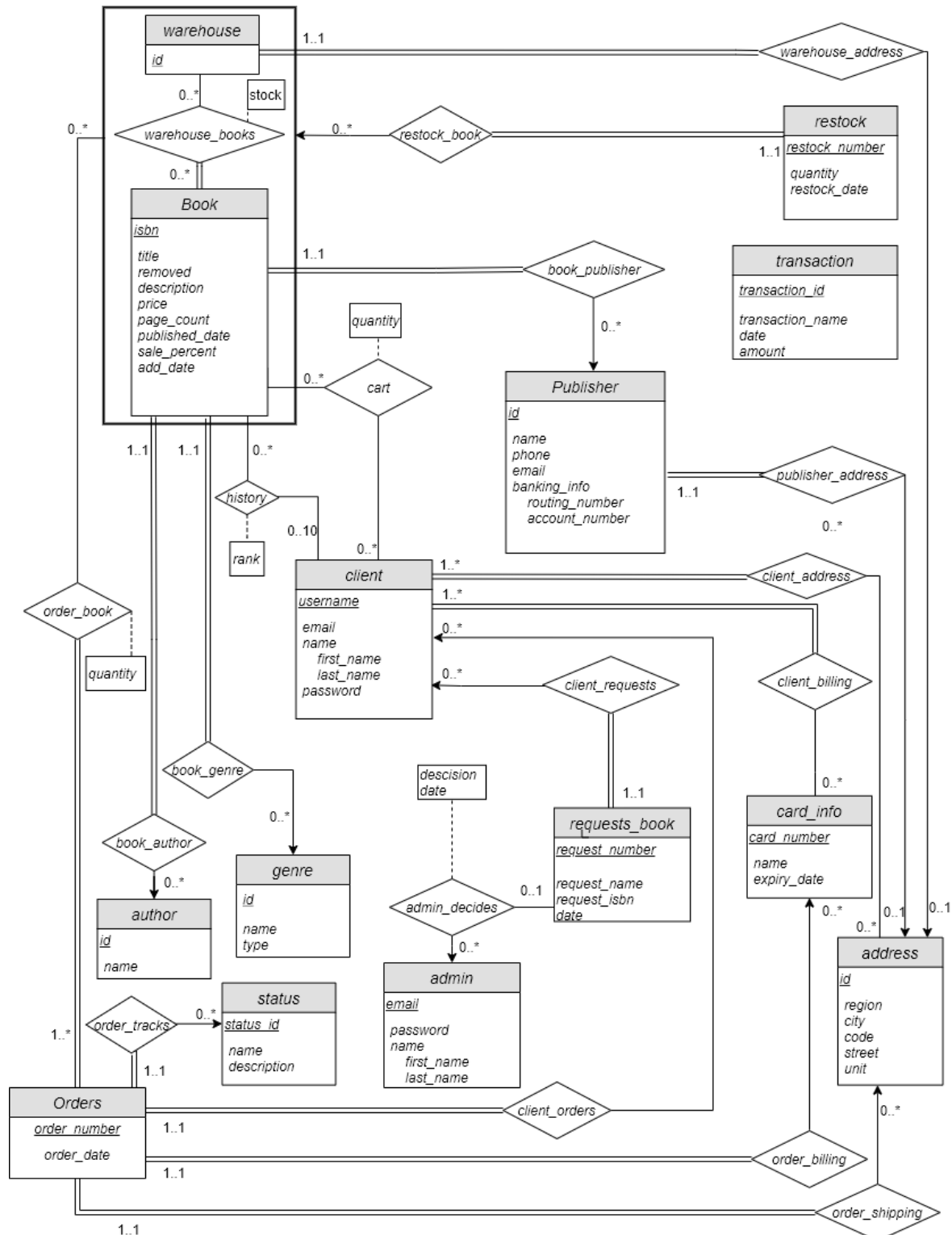
APRIL 6, 2020
COMP3005 WINTER 2020
Sharjeel Ali #101070889

Contents

Contents	1
1. Conceptual Design	3
1.1. Book	4
1.2. Author	4
1.3. Genre.....	4
1.4. Publisher	4
1.5. Client	5
1.6. Admin.....	6
1.7. Transaction.....	6
1.8. Warehouse.....	6
1.9. Orders	7
1.10. Restock.....	8
1.11. Request_book	8
1.12. Address.....	8
2. Reduction to Relation Schemas	9
3. Normalization of Relation Schemas.....	10
3.1. author.....	10
3.2. genre	10
3.3. publisher(id, name, phone, email, address_id, routing_number, account_number)	11
3.4. book(isbn, title, description, price, page_count, published_date, add_date, rating, rating_count, sale_percent, author_id, genre_id, publisher_id)	11
3.5. orders(order_number, username, order_date, status_id, card_number, address_id).....	12
3.6. status(status_id, name, description).....	12
3.7. order_book(order_number, isbn, warehouse_id, quantity)	13
3.8. cart(username, isbn, quantity).....	13
3.9. client(username, email, first_name, last_name, password).....	14
3.10. admin(email, first_name, last_name, password)	14
3.11. transaction(transaction_id, transaction_name, amount, date)	15
3.12. warehouse(id, address_id).....	15
3.13. warehouse_books(warehouse_id, isbn, stock).....	15
3.14. address(id, region, city, code, street, unit)	16
3.15. client_address(username, address_id).....	17

3.16. card_info(card_number, name, expiry_date).....	17
3.17. client_billing(username, card_number)	18
3.18. restock(restock_number, isbn, warehouse_id, quantity, restock_date).....	18
3.19. request_book(request_number, username, request_name, request_isbn, date).....	18
3.20. admin_decides(email, request_number, decision, date)	19
3.21. history(username, isbn, rank)	19
4. Database Schema Diagram	21
5. Implementation	22
5.1. Overview	22
5.2. Design.....	22
5.3. Client	26
5.4. Owner.....	39
6. Bonus Features.....	48
6.1. Fuzzy search	48
6.2. Bestsellers	49
6.3. Recently viewed	49
6.4. Recently released.....	52
6.5. Requesting books.....	52
6.6. GUI Webpage	53
6.7. Transactions	56
6.8. Restocking.....	55
6.9. Filter sales between dates	57
6.10. Restock fees	Error! Bookmark not defined.
6.11. Warehouse	54
7. Github Repository	58
7. Website link	58

1. Conceptual Design



The ER diagram that was created is shown above. For this section, an entire walkthrough that includes all explanations and assumptions about the design shown above will be explained below.

1.1. Book

The book entity contains all the relevant book information that can be derived from the primary key (isbn). These attributes are the title, description, price, page_count, published_date, rating, rating_count, sale_percent, removed and add_date. The reason for why these attributes are all put into the single book entity is because they are all unique to the book (isbn).

Book has three main “has a” relationships, where the book derives attributes from other entities. These are the author, publisher and genre (Relationships being book_author, book_publisher and book_genre). To make matters simpler in terms of designing the book entity, the assumption was made that each book must have an author, genre and publisher (Total participation), and contain at most one author, publisher and genre (So each book has a single genre, author and publisher). The book entity itself derives the ids of each of the three tables (Ids will be explained further below in genre and publisher sections). The publisher, author and genre all share a many to one relation (From their point of view) to the book, as an author, publisher or genre could potentially have more than one book that it has a relation with (a “fantasy” genre could have thousands of books, a publisher could publish many books, and an author could write many books).

1.2. Author

The author entity is very simplistic and contains just an id and name attribute. The reason for name not being a composite type (i.e. first_name, last_name) is due to it being not important. For the bookstore vision, the client or admin (owner) does not need to access only the author’s first or last name, and needs both whenever they access the author, therefore name is not a composite type. The same can be said of why the author entity does not contain any extra information, such as email, phone, address, etc. As all this information is once again irrelevant for the bookstore and would realistically be things that the publisher would know, not the store. The reason for why an id attribute exists as the primary key is that there could be books written by authors who share the same full name (Although highly unlikely). As explained above, the author has a partial participation relation with a book, where the author can write many books (many to one).

1.3. Genre

The genre entity is very similar to the author entity described above, in that they both share an id and a name. Although the bookstore would have a unique genre name (i.e. fantasy, sci fi, etc.), the reason for the id attribute is to make the author, genre and publisher entities all very similar, which would be a great benefit in terms of overall efficiency when implementing the schema. This entity also contains another attribute, being type. Type in this case refers to it being fiction or nonfiction (Which as an avid book reader is important to state, as both fiction and nonfiction contain a wide plethora of genres such as history, sciences, fantasy, sci fi, etc.)

1.4. Publisher

The publisher entity is also structurally like genre and author described above, in that it contains an id as a primary key, as well as name (Being the publisher name). However, as the requirements stated, the bookstore must store details about each publisher such as name, address, email address, phone number(s) and banking account, so therefore the publisher entity is much more complex than author and genre. Each publisher entity contains a name, phone, email and direct deposit information. The phone number is a part of the publisher entity due to the assumption that a publisher can have at most one phone number, so there are no multi valued phone numbers described here. The email address is also unique to each publisher and can also be put inside the publisher entity.

As for banking account information, this was done through a realistic standpoint. Due to the bookstore/owner needing to give a percentage of each book sale to the publisher, this means the publisher receives money from the

bookstore. In a real-world scenario this is done through direct deposits. The original design of the ER diagram included the direct deposit information as being a separate entity that publisher would have a total participation relation with. However, due to each direct deposit entity being tied to only a single publisher through another total participation relation, and the direct deposit only being used for publisher, this meant having a separate entity was redundant. Therefore, the direct deposit was transformed into a composite attribute and placed inside the publisher entity. Each direct_deposit attribute contains the transit number (Unique to each bank location), intuition number (Unique to each bank, i.e. CIBC, RBC, TD), and the account number (Unique to each account within the bank location and bank itself).

The final attributes that a publisher requires is the address. This was done through creating a separate entity address. The publisher has a total participation “has an associated” relation with address, where it derives the address id from the address entity. The reason for this association type relation is due to a publisher only requiring a single address (One to many). Due to this relationship being similar to the earlier model for direct deposit, the reason for why publisher address isn’t a composite attribute in the publisher is due to the address being used by many other entities, so it would be more organized to have it be a separate entity and not composite.

1.5. Client

The client entity is the entity that contains all the user account information. In this case, this entity contains the username, email, the name and password. The reason for having both a username and password is due to scalability and personalization. A large aspect about schemas is their scalability, so having a username attribute could be useful if in the future, the bookstore allowed clients to add book reviews, etc. The name attribute is a composite attribute in that it contains both the first name and last name, which sometimes both the full name or either the first name are required (i.e. when ordering the full name could appear, however the account page could simply display the user’s first name).

Per the requirements, each client has their own checkout/cart, which is defined as the relation between book and client, and as a result, this relation stores all the books and their quantities that the client adds to their cart. Because the requirements state that the client needs to be registered to checkout, this was interpreted in this segment; The client can only add books to their carts if they’re signed in (As the cart relationship requires both isbn and client username). The client can also choose to not have any books in their cart, which means they share a partial participation with book, and the same can be said of book, in that it does not need to have a book in every client’s cart. As a result, a client can add as many books as they wish (As well as their quantities), and a book can appear in an unlimited number of client’s carts. Therefore, this is defined as a many to many relationship.

Another very similar relationship to cart that client has with book is the history relationship. This is a bonus feature that was added. Essentially, the client can view their viewing history (The books they viewed). Each book entity can be viewed by any number of clients, and so book partially participates with client. However, to conserve space, the history only stores a client’s ten most recently viewed books, therefore the client’s relationship with history is defined as at most 10 books.

The client must also contain address and billing information, this means client must have a total participation relation with both address and billing. This was made under the assumption based on the requirements that since the client inserts shipping and billing information on registration, that each client must be totally participating with these. However, the requirements also state that upon checkout, the client can choose another shipping or billing information than the one used in registration. This therefore meant that the client should be able to add an unlimited amount of shipping or billing information, and therefore the client shares a many to many relationship with both. The billing information was interpreted as payment information (i.e. card number). Both the payment information and address can belong to any client entity as a many to many relationship. This is due to the possibility of multiple clients living in the same address or using the same card numbers.

The client can also order an unlimited amount of book orders, and this is shown through the orders – client relationship through client orders relation, where a client can have a many to one relationship with orders. As another bonus feature, the client can also request a book (Described in detail below) through a book request. In this case, a client can request an unlimited number of books to be added to the bookstore, meaning this is a many to one relation (Similar to orders), in that a book request can only belong to one client.

1.6. Admin

The admin entity stores the admins or owner of the bookstore. The admin exists as a separate entity to client due to admins not being able to order books, have a cart, or book viewing history. The admin entity is also fairly like the client entity, in that both share an email, password and a composite name attribute (Formed of first_name and last_name). However, the admin entity does not contain a username attribute, instead using the email attribute as the primary key, contrary to the client entity. This is due to there being no need for admin scalability, as admins would not be anonymous (through usernames) and would not write book reviews (As a future addition example). Therefore, username is not needed. Each admin has the option of adding or removing a book, per the requirements. However, this is unnecessary to show as a relationship with book due to no data being shared between them (i.e. author entity shares author id with book, etc.), as well as this add/remove relationship just being an action, therefore no need to add any new relationship in this case.

The admin however still has two relationships, both being bonuses. The first being that an admin can approve a book request from a client, and therefore, by approving a book request, they can add the book into the bookstore (Once again, this is an action so its not being shown as a relationship), or they can reject the request (This is seen in admin_decides relation, with the decision and date). Each admin entity can decide on any number of book requests, making it a many relationship (0..1), and they do not need to participate in these book request decisions, therefore it is a partial participation relationship.

1.7. Transaction

The transaction entity is another bonus feature. This entity stores all transactions that the admin has entered. However, the transaction entity does not store a book sale transaction or sending money to the publisher transaction. This is due to both scenarios being stored within the order entity relationships themselves, so a query can be performed to extract the necessary information from them, therefore making it redundant to add book and publisher sales into the transaction entity. However, the reason for the existence of the transaction entity is for cases where the admin wishes to enter other sales or fees, such as taxes or rent, which would be included in the sales reports. The reason for why there is no relation between transaction and admin is the same reasoning used to describe the admin adding or removing books from above. The entity itself has a transaction_id, name, date and amount. The way each transaction entity would differentiate between debit or credit transactions is through the amount (i.e. if amount < 0, it is debit, removing money from the bookstore).

1.8. Warehouse

The warehouse entity was created due to future scalability for the bookstore. Per the requirements, a book needs to be shipped from a warehouse that also contains a quantity (In this case stock). So therefore, the most efficient option was creating a warehouse entity to store all the books and their stocks. The warehouse itself is defined by just a single attribute, being its id. However, it has multiple relationships with other entities. The first relationship that warehouse has is with address through warehouse_address. Logically, a warehouse needs to have an address, and therefore warehouse totally

participates with the address table, and derives the address id to include within the warehouse entity (Described in section 2). The warehouse can also have at most one address logically, and a specific address can have at most one warehouse as well, therefore this is a one to one relationship.

The second relationship that warehouse contains is warehouse_books with books. This relationship defines each book stored in the warehouse through isbn and warehouse id, as well as the stock for each book. Because some warehouses could potentially be empty, the warehouse entity partially participates with book, but can store an unlimited number of books (Though unlikely, for obvious reasons). A book can also belong to multiple warehouses and must belong to a given warehouse as well, meaning a book totally participates with warehouse. However, the requirements state that all books are shipped (and stored) in only one warehouse. Although the design allows books to be stored in multiple warehouses, the actual implementation will only use a single warehouse to meet the requirements listed.

Finally, the warehouse and book entities need to be aggregated into a single entity for use with the order and restock entities (This is the bold outline/box around warehouse and book). Aggregation means treating a relationship (In this case warehouse_books, and its entities warehouse and book) as a single entity, that can be used by other relations, giving the other relationships and entities access to the specific isbn and warehouse id.

1.9. Orders

The orders entity stores all orders done by the client. The entity contains just two attributes, being the order_number, and order_date, however it has many relationships between other entities. The most important being the client_orders relationship. In this case, orders totally participate in this relationship, to derive the username of the client that submitted the order (As every order must be tied to a client). This also means order can have at most one user to be tied to, and is therefore a 1..1 relationship, making client_orders one to many (From orders point of view).

Another two important relationships that order has is order_shipping and order_billing between address and payment info. This is due to the requirement that states a client must insert shipping and billing information into the order, and as such it is assumed that the order entity would totally participate with both these entities as well. And similarly, to client_orders relationship, order could have at most one address and payment information attached to it (Whereas payment info and address can belong to many different orders). This once again makes these two relationships one to many (From orders point of view).

Another requirement was showing tracking information. This is done via the status entity and order_tracks relationship. The status entity itself contains the status id, the name of the status (i.e. shipped), and the description of the status. The order must participate with status as every order needs tracking information, making order a totally participating relation, and once again a one to many with status. The status id of an order would however update and change as the order progresses (i.e. shipped, out for delivery, delivered, etc.). Due to making the status and tracking simple, the order will not actually track the location (As this is different book to book), and the assumption was made that this tracking can simply be where the order is currently in the stages of ordering to delivery.

The 2nd most important relation for order to have is with book, in that each order must contain the books the client ordered as well as their quantity, which is defined as the relationship between order and the aggregated warehouse_books described above. The set of books and quantity that the client ordered is taken directly from their cart, but this is not shown as a relationship between order_book and cart due to the cart being dynamic and could potentially contain different books than what the order_book contains. Each order must be totally participating with a book (i.e. each order contains at least one book), whereas the aggregated warehouse_books does not need to participate in order (As there are certain books that are never ordered from a warehouse). In this case, order_book would contain the order_number, isbn, warehouse_id and quantity ordered from the relationship. The reason for having a relationship with the aggregated warehouse_book is simply due to making sure each book's quantity ordered actually exists in the warehouse (i.e. It is not possible to order 10 books if only 4 exist in the warehouse_books), therefore aggregation is

needed. Per the requirements stating that to assume all books are shipped from only one warehouse, the implementation will only include just one warehouse to order books from, fulfilling the requirement, however due to scalability it is still possible for an order to contain books from different warehouses in the future. Therefore, the order_book relation is a many to many relationship.

1.10. Restock

Per the requirements, a book must be restocked if its quantity reaches a value below a certain threshold, as well as email the publisher. The reason for restock being an entity instead of just an unlisted action is due to restock sending a publisher email, which the assumption is that the admin should be able to see when an email has been sent to a given publisher, as a record. The restock entity itself contains a restock_number, the quantity to restock, as well as the restock date. However, it also has a “has a” total participation relationship with the aggregated warehouse_books, where it requires the isbn and the specific warehouse id of where to restock to. This also means that the aggregated warehouse_books doesn’t need to participate with restock.

1.11. Request_book

Request book is the bonus feature described above. Essentially, a client can request a book to be added to the bookstore by entering either the book isbn to be added or the book title. This means both must be stored directly in the request_book entity, as well as the request_number and date of request. It also must have a “has a” total participation relation with client, as request_book needs the username of the client who requested the book. And it also has a relationship with admin through admin_decides, in that an admin can approve/reject a book request. Due to requests not being answered until an admin manually approves/rejects the book, request_book partially participates with this relationship, but can have at most one decision being made (Meaning one to many from the request_book side).

1.12. Address

Due to describing all the address entity’s relationships in the previous section, this will only refer to the actual attributes stored within address itself. This is not an assumption on address, however the entity itself does not contain a country attribute. This was made on purpose since the assumption is that the bookstore is for Canadians only. This is due to the overall complications that occur when adding an address attribute and storing constraints, etc. Such as for example, Canada uses provinces and territories, US uses states, other countries use regions. Same for postal codes, where different countries have different formats or none. Therefore, to make the address entity simple, country is removed. This also means all publishers and warehouses will now be pure Canadian branded.

2. Reduction to Relation Schemas

Relations:

1. *author*(id, name)
2. *genre*(id, name, type)
3. *publisher*(id, name, phone, email, address_id, routing_number, account_number)
4. *book*(isbn, title, description, price, page_count, published_date, add_date, sale_percent, author_id, genre_id, publisher_id)
5. *orders*(order number, username, order_date, status_id, card_number, address_id)
6. *status*(status id, name, description)
7. *order_book*(order number, isbn, warehouse_id, quantity)
8. *cart*(username, isbn, quantity)
9. *client*(username, email, first_name, last_name, password)
10. *admin*(email, first_name, last_name, password)
11. *transaction*(transaction id, transaction_name, amount, date)
12. *warehouse*(id, address_id)
13. *warehouse_books*(warehouse id, isbn, stock)
14. *address*(id, region, city, code, street, unit)
15. *client_address*(username, address id)
16. *card_info*(card number, name, expiry_date)
17. *client_billing*(username, card number)
18. *restock*(restock number, isbn, warehouse_id, quantity, restock_date)
19. *request_book*(request number, username, request_name, request_isbn, date)
20. *admin_decides*(email, request number, decision, date)
21. *history*(username, isbn, rank)

3. Normalization of Relation Schemas

For the following section, all relations will be reduced to BCNF form. For this section, before I reduce, and normalize my relations, the following relations from section 2 above cannot be normalized any further due to them containing only two attributes, where the primary key can determine the other attribute.

Each relation that can be normalized will follow under the following three steps:

- a. Check if R is in BCNF Using the simplified test (If R has not been decomposed yet)
 - I. Find a non-trivial dependency $a \rightarrow b$ such that it causes a violation of BCNF
 - II. Computer a^+
 - III. Verify that a^+ includes all attributes of R , proving it is a superkey of R .
- b. Decompose R into BCNF form

3.1. *author(id, name)*

Functional dependencies:

$F = \{$
 $id \rightarrow name$
 $\}$

Background:

There exist authors with the same name, so an id is needed to identify. Because it is a unique id, it can determine the name.

Normalization:

This relation is already normalized due to the following rule:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Because there is only two attributes, it cannot be normalized any further.

3.2. *genre(id, name, type)*

Functional dependencies:

$F = \{$
 $Id \rightarrow name, type$
 $name \rightarrow id, type$
 $\}$

Background:

Each genre is assigned a unique id, therefore the id can determine everything else. However, the genre name is also unique, so it can also be used to determine everything else. However, type cannot determine anything as type (Fiction or nonfiction) can be assigned to many different genres.

Normalization:

This relation is already normalized due to the following rule:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Because there are only two attributes, it cannot be normalized any further.

3.3. *publisher(id, name, phone, email, address_id, routing_number, account_number)*

Functional dependencies:

```
F = {
  Id → name, phone, email, address_id, routing_number
  phone → name, Id, email, address_id, routing_number, account_number
  email → name, Id, phone, address_id, routing_number, account_number
  address_id → name, Id, phone, email, routing_number, account_number
  routing_number, account_number → name, Id, phone, email, address_id
  name → id, phone, email, address_id, routing_number, account_number
}
```

Background:

Each publisher has a unique id and email, as well as phone number, address, and banking information and finally name to them (Due to legal reasons, you can't find two publishers with the same name and you won't find two publishers under the same address).

Normalization:

We know that by looking at the attributes, that each of them can be used to identify the other in this case (As there only exists a single phone, email, address and direct deposit bank information for a given publisher). Therefore, each functional dependency contains a superkey. Using the following rule, we can prove it is normalized:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Therefore, each a^+ of every $a \rightarrow b$ includes all attributes of R . Therefore, we can say publisher is normalized.

3.4. *book(isbn, title, description, price, page_count, published_date, add_date, sale_percent, author_id, genre_id, publisher_id)*

Functional dependencies:

```
F = {
  isbn → title, description, price, page_count, published_date, add_date, sale_percent,
  author_id, genre_name, publisher_id
}
```

Background:

The rating and ratings_count. These two although one would assume are dependent upon each other, and therefore not a superkey, these are not functional dependencies of book, due to multiple books having the same count but different ratings and vice versa. Because page, count, published_date, add_date, etc. are all determined based upon the book (ISBN), isbn can determine everything. But anything else, like for example price, which many books have the same price, or add_date, etc. cannot determine anything.

Normalization:

Another important note is author_id and publisher_id. Author_id does not determine publisher_id. This is made under the assumption that although rare, some authors switch publishers during their writing careers, so this functional dependency cannot exist.

We know that by looking at the attributes, that the only attribute that can determine another is isbn, which determines the rest of the attributes. Using the following rule, we can prove it is normalized:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Therefore, each a^+ of every $a \rightarrow b$ (In this case just the isbn \rightarrow superkey) includes all attributes of R. Therefore, we can say book is normalized.

3.5. orders(order_number, username, order_date, status_id, card_number, address_id)

Functional dependencies:

$F = \{$
 $Order_number \rightarrow username, order_date, status_id, card_number, address_id$
 $\}$

Background:

Each book order has a unique number which can be used to determine everything else, and some users ordering multiple, username cannot be used, same thing with card number and address id, these can be assigned to multiple orders.

Normalization:

The important realization about this is the card_number, address_id and username. These three attributes do not share a functional dependency (i.e. $card_number \rightarrow username, address_id \rightarrow username$). This is under the assumption that there could be the possibility of multiple orders sharing the same address but being ordered by different users (i.e. family members or roommates). Therefore, requiring username in this relation is important and is then as a functional dependency, but $card_number \rightarrow username, address_id \rightarrow username$ are not however.

We know that by looking at the attributes, that the only attribute that can determine another is order_number, which determines the rest of the attributes, making it a superkey. Using the following rule, we can prove it is normalized:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Therefore, each a^+ of every $a \rightarrow b$ (In this case just the order_number \rightarrow) includes all attributes of R. Therefore, we can say orders is normalized.

3.6 status(status_id, name, description)

$F = \{$
 $Status_id \rightarrow name, description$
 $Name \rightarrow status_id, description$
 $Description \rightarrow name, status_id$
 $\}$

Background:

A unique status id is assigned to each status, but the status name and description are also unique, so these values can be used to identify the relation as well.

Normalization:

This one is simple and similar to genre normalized above. We know that by looking at the attributes, that every attribute is defined as the a in $a \rightarrow b$, and that every b contains the rest of the attributes ($R-a$), meaning each attribute determines every other attributes. We can use the following proof to check:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Therefore, each a^+ of every $a \rightarrow b$ includes all attributes of R (They are all superkeys). Therefore, we can say status is normalized.

3.7. order_book(order_number, isbn, warehouse_id, quantity)

$F = \{$
 $Order_number, isbn \rightarrow warehouse_id, quantity$
 $\}$

Background:

Each order contains a book order, which is specified as the order number and isbn of the book, which determines the warehouse_id as well as the book quantity ordered. However, an order can contain books of the same quantity, and from the same warehouse. The warehouse_id is not a primary key and isn't on the left side due to this meaning a book can be ordered from two warehouses, which should not be allowed per the requirements.

Normalization:

We know that by looking at the attributes, that there is only one functional dependency which is just order_number, isbn. Which determines everything else, making order_number, isbn a superkey. Using the following rule, we can prove it is normalized:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Therefore, each a^+ of every $a \rightarrow b$ (In this case just the order_number, isbn \rightarrow) includes all attributes of R . Therefore, we can say order_book is normalized.

3.8. cart(username, isbn, quantity)

$F = \{$
 $username, isbn \rightarrow quantity$
 $\}$

Background:

A user can have a cart that contains a book by its isbn, however they can have multiple items with the same quantity, so quantity doesn't determine anything.

We know that by looking at the attributes, that there is only one functional dependency which is just username, isbn. Which determines everything else, making username, isbn a superkey. Using the following rule, we can prove it is normalized:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Therefore, each a^+ of every $a \rightarrow b$ (In this case just the *username, isbn* \rightarrow) includes all attributes of R . Therefore, we can say cart is normalized.

3.9. *client(username, email, first_name, last_name, password)*

$F = \{$
Username \rightarrow *email, first_name, last_name, password*
Email \rightarrow *username, first_name, last_name, password*
 $\}$

Background:

The client account contains a username which is unique to their specific account, as well as an email which they can use either to log into the database, but there could be multiple accounts with the same name or password.

Normalization:

We know that by looking at the attributes, that there are only functional dependencies which are username and email that determine everything else. Using the following rule, we can prove it is normalized:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i , meaning it is a superkey

Therefore, each a^+ of every $a \rightarrow b$ includes all attributes of R (all superkeys). Therefore, we can say clients is normalized.

3.10. *admin(email, first_name, last_name, password)*

$F = \{$
Email \rightarrow *first_name, last_name, password*
 $\}$

Background:

An admin has a unique email which determines their name and password for the same reasons as described for client above.

Normalization:

We know that by looking at the attributes, that there are only two functional dependencies which is just email. Which determines everything else, making email a superkey. Using the following rule, we can prove it is normalized:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Therefore, each a^+ of every $a \rightarrow b$ (In this case just the *email* \rightarrow is a superkey) includes all attributes of R . Therefore, we can say admin is normalized.

3.11. transaction(transaction_id, transaction_name, amount, date)

F = {
 $Transaction_id \rightarrow transaction_name, amount, date$
 }

Background:

A transaction is determined by a unique transaction id, which can determine the transaction name, amount and date, however some transactions could share the same date or amount or name, so these cannot be used to determine anything.

We know that by looking at the attributes, that there is only one functional dependency which is just transaction_id. Which determines everything else, making transaction_id a superkey. Using the following rule, we can prove it is normalized:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Therefore, each a^+ of every $a \rightarrow b$ (In this case just the transaction_id \rightarrow is a superkey) includes all attributes of R. Therefore, we can say transaction is normalized.

3.12. warehouse(id, address_id)

F = {
 $id \rightarrow address_id$,
 $address_id \rightarrow id$
 }

Background:

This relation is similar to publisher from above in that all attributes can determine the author (As a warehouse can have at most one address, and that address can contain just the warehouse).

Normalization:

We can also use the following rule to prove it is normalized:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Therefore, each a^+ of every $a \rightarrow b$ includes all attributes of R (They are all superkeys). Therefore, we can say warehouse is normalized.

3.13. warehouse_books(warehouse_id, isbn, stock)

F = {
 $Warehouse_id, isbn \rightarrow stock$
 }

Background:

A warehouse can contain a specific isbn, which determines a specific set stock at that warehouse that contains an isbn, but some warehouses could contain books with the same stock, so that isn't used to determine this.

Normalization:

We know that by looking at the attributes, that there is only one functional dependency which is *Warehouse_id, isbn*. Which determines everything else, making *Warehouse_id, isbn* a superkey. Using the following rule, we can prove it is normalized:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Therefore, each a^+ of every $a \rightarrow b$ (In this case just the *Warehouse_id, isbn* \rightarrow) includes all attributes of R . Therefore, we can say *warehouse_books* is normalized.

3.14. address(id, region, city, code, street, unit)

$F = \{$
 $Id \rightarrow region, city, code, street, unit,$
 $Code \rightarrow city$
 $\}$

Background:

The following assumptions were made for these functional dependencies, that each postal code in Canada is unique, as well as that there can exist multiple cities with the same name (i.e. Victoria exists in both BC, and Ontario). However, each postal code number is unique to each region within Canada. The reason why postal code doesn't determine province, is due to one specific outlier, being the Northwest territories and Nunavut. These two both share the same postal code letter (Starting with X), therefore code cannot determine city.

Normalization:

As seen above, this relation address has multiple functional dependencies, this means there is a possibility of it not being in BCNF, and therefore not being normalized. Once again, we check if there exists any functional dependency within address, such that a^+ in $a \rightarrow b$ is not a superkey of R , and therefore does not include all attributes.

We know that id^+ is a superkey, due to id determining every attribute in address as shown above.

The next one is code. We can compute $code^+$:

$Code^+ \rightarrow city$ (Using functional dependency from above)

$Code^+ \rightarrow city, code$ (Using Armstrong's axiom, reflexivity rule.)

$Code^+$ has now been computed, and it determines the city and region attributes, however it does not determine id , unit or street, therefore we can say that code is not a superkey. Therefore, we must decompose address on Code. (Note: the reason for not using the simplified test in this case is that the final normalized result makes a bit more sense).

To decompose a relation, it is the following:

$Result = (result - R_i) \cup (R_i - b) \cup (a, b)$

This can be defined for address as follows:

$Address = (id, region, city, code, street, unit) - (id, region, city, code, street, unit) \cup (id, region, city, code, street, unit - city) \cup (code, city)$

$Address = (id, region, code, street, unit) \cup (code, city)$

Now we have decomposed the two relations into the following:

$Address_main (id, region, code, street, unit)$

$Address_second (code, city)$

Now once more, check if both main and second address relations violate BCNF, and if so, further decompose the relations.

$Address_main (id, code, street, unit)$

$F = \{$

$Id \rightarrow region, code, street, unit$

$\}$

Now check if it is normalized using the following rule:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

We know that main only contains one functional dependency, being id. We also know that id determines every other attribute in the relation. Therefore, we can say that address main is normalized.

$Address_second (code, city)$

$F = \{$

$Code \rightarrow city$

$\}$

Now check if it is normalized using the following rule:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

We know that main only contains one functional dependency, being code. We also know that code determines every other attribute in the relation. Therefore, we can say that address main is normalized.

Therefore, we can reduce the relation to the following:

$Address_main (id, region, code, street, unit)$

$Address_second (code, city)$

3.15. client_address(username, address_id)

Normalization:

This relation has no functional dependencies. This is because a username cannot determine an address (As a user can have multiple addresses), and an address cannot determine user (As an address can have multiple users listed).

3.16. card_info(card_number, name, expiry_date)

$F = \{$

$Card_number \rightarrow name, expiry_date$

$\}$

Background:

A card number is unique to that specific card and can determine the card user's name and expiry_date. However, there are cards that share the same expiry date and people can own multiple cards, so those two cannot be used to determine this.

Normalization:

We know that by looking at the attributes, that there is only one functional dependency which is just *card_number*. Which determines everything else, making *card_number* a superkey. Using the following rule, we can prove it is normalized:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Therefore, each a^+ of every $a \rightarrow b$ (In this case just the *card_number* \rightarrow) includes all attributes of R. Therefore, we can say card_info is normalized.

3.17. client_billing(username, card_number)**Normalization:**

This relation has no functional dependencies. This is because a username cannot determine a card number (As a user can have multiple cards), and a card cannot determine user (As a card can have multiple users listed).

3.18. restock(restock_number, isbn, warehouse_id, quantity, restock_date)

$F = \{$
Restock_number \rightarrow *quantity, restock_date*
 $\}$

Background:

A restock is given a unique restock number, which can be used to determine everything else. However, warehouses can have multiple book restocks, and a book (isbn) can be restocked multiple times, so these cannot determine anything. The same logic used in previous relations can be used to explain quantity and date.

Normalization:

We know that by looking at the attributes, that there is only one functional dependency which is *Restock_number*, which determines everything else, making *Restock_number* a superkey. Using the following rule, we can prove it is normalized:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Therefore, each a^+ of every $a \rightarrow b$ (In this case just the *Restock_number* \rightarrow) includes all attributes of R. Therefore, we can say restock is normalized.

3.19. request_book(request_number, username, request_name, request_isbn, date)

$F = \{$
Request_number \rightarrow *username, status, request_name, request_isbn, date*
 $\}$

Background:

Each book request is given a unique request number, which can be used to determine everything else. However, a user can request multiple books, so that cannot be used to determine this as it isn't unique. As for name and isbn, these cannot be unique as the admin can just reject duplicates and sometimes the client can have spelling errors (So it would bypass the unique constraint check postgres uses anyway), so the admin would still have to check these.

Normalization:

We know that by looking at the attributes, that there is only one functional dependency which is just *Request_number*. Which determines everything else, making *Request_number* a superkey. Using the following rule, we can prove it is normalized:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Therefore, each a^+ of every $a \rightarrow b$ (In this case just the *Request_number* \rightarrow) includes all attributes of R . Therefore, we can say request_book is normalized.

3.20. admin_decides(email, request_number, decision, date)

$F = \{$
Email, request_number \rightarrow *decision, date*
 $\}$

Background:

An admin can decide on a specified book request, but that decision can also be applied to multiple books. However, for future scalability, maybe multiple admins need to decide on a book request for a final decision to be made, therefore, email is important for being a primary key.

Normalization:

We know that by looking at the attributes, that there is only one functional dependency which is just *Email, request_number*. Which determines everything else, making *Email, request_number* superkey. Using the following rule, we can prove it is normalized:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Therefore, each a^+ of every $a \rightarrow b$ (In this case just the *Email, request_number* \rightarrow) includes all attributes of R . Therefore, we can say admin_decides is normalized.

3.21. history(username, isbn, rank)

$F = \{$
username, isbn \rightarrow *rank*
 $\}$

Background:

A username and an isbn can determine the viewing order for that specific user, but there are books in viewing history that share the same viewing number across multiple users (i.e. user A viewed book with rank 1, same for user B, etc.), therefore it cannot determine anything and is not a functional dependency.

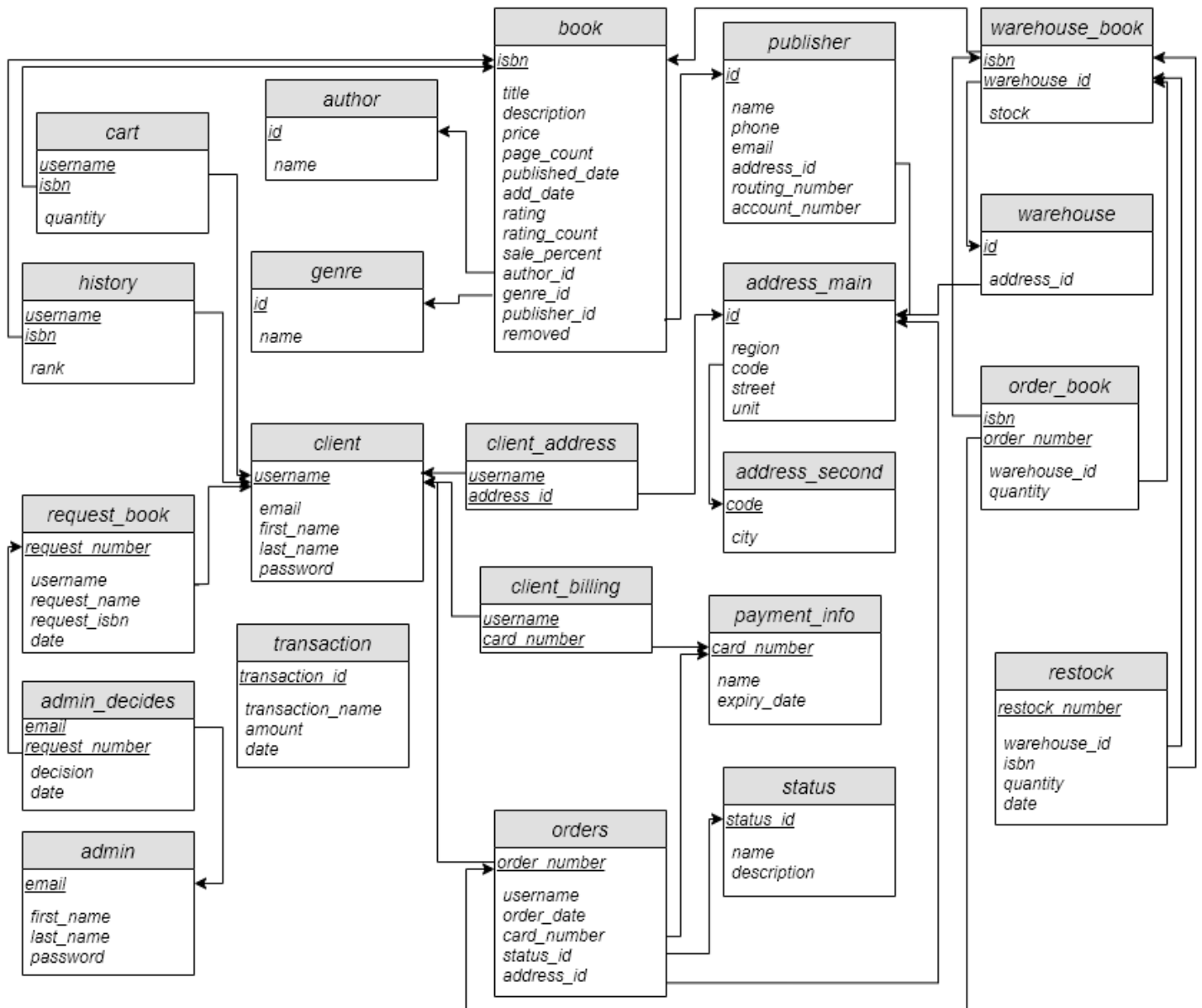
Normalization:

We know that by looking at the attributes, that there is only one functional dependency which is just *username, isbn*. Which determines everything else, making *username, isbn* a superkey. Using the following rule, we can prove it is normalized:

For every set of attributes, $a \subseteq R_i$, check that a^+ (the attribute closure of a) either includes no attribute of $R_i - a$, or includes all attributes of R_i

Therefore, each a^+ of every $a \rightarrow b$ (In this case just the *username, isbn* \rightarrow) includes all attributes of R. Therefore, we can say history is normalized.

4. Database Schema Diagram



5. Implementation

5.1. Overview

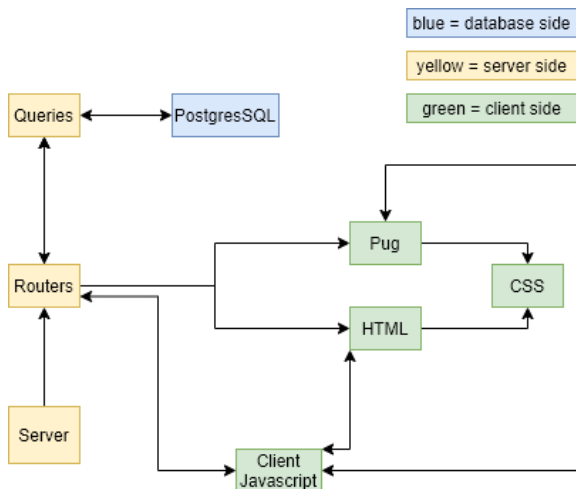
The project program was implemented in JavaScript as a web-based application. In this section, the main implementations of the project will be discussed as well as the different user interfaces and scenarios that have been added to the website.

5.2. Design

5.2.1. Design Overview

This section of the report will discuss the main design and architecture of the web application, the modules used and how the codebase is structured.

5.2.2. Data flow



The above diagram showcases the data flow of the web application. The web application itself, due to being web based implements a 3-level structure as seen above.

The first level is the database, which is the back end of the application and contains all the necessary data that the application must use for the bookstore. This level communicates with the server side, which is where the actual server is stored, which in this case is the queries block.

The queries block is the segment of the server-side level that communicates directly with the Postgres server using queries and is the only one to do so (I.e. if another level or block must communicate with the server, they must communicate through the queries block first). **Note: If you must see the queries, they are located here.**

The queries block is called from the Routers block, which is used to send and receive requests from the client side using specific URL parameters. The Routers are the most important aspect of data flow, as they route the requests to their respective locations (I.e. routing to a new webpage or routing a new query to receive data). These routers are all stored and imported into the Server block, which is the actual server that is running.

The routers are able to send files to the client side, which can either be Pug (More on this in section 5.2.3.) or HTML files. These pug or HTML file blocks are essentially the same thing. Both Pug and HTML files implement CSS for front end webpage styling. They can also load client-side JavaScript.

The JavaScript files are able to communicate back and forth with their respective Pug or HTML files (Depending on where the JavaScript file was loaded from) for front end actions such as a user clicking a button or selecting an option from a drop-down menu. They are then able to send requests to the server side which the routers receive and handle on the server side.

5.2.3. Modules

The web application uses a wide variety of modules. In this section, each of the modules used will be discussed, as well as how they're used and their implementations into the web application.

Pg:

The Pg module is used to connect to a PostgreSQL database server directly, and therefore able to submit queries and receive data back from the server in the form of JSON objects. To use the Pg module, first it must be imported into the server-side JavaScript, then a pool must be created from it (A Pool in this case is how JavaScript connects to a specific PostgreSQL database using built in parameters defined as the following:

```
const pool = new Pool({
  user: "apqsznwhludbct",
  host: "ec2-35-172-85-250.compute-1.amazonaws.com",
  database: "d66prsg7g28r53",
  password: "161239fa8d874dbc62119103682c4b1e4bd64c313a1535ddcd98f406301a262f",
  port: "5432",
  ssl: true
```

```
});
```

In the above, it defines a const pool variable as storing all the credentials and database access information for a particular database server. Then to query to the database server, a simple pool.query is used which is defined as the following:

```
pool.query("select isbn from book where removed=false order by isbn limit $1",
  [size], (err, result) => {
    if (err) {
      return console.error('Error executing query', err.stack)
    }
    //console.log(result.rows);
    res.json(JSON.stringify(result.rows));
  });
```

The above pool.query uses an asynchronous function to return the result of the queries, which can be console.logged using the rows attribute of the result as shown above. To input a parameter to a query, the array defined before the (err, result) as [size] is used. In this particular case, size is the input parameter. Each parameter within the array must be present in the query string as a \$#. In the above case, size is present as the \$1 in limit \$1. Therefore, it can be determined that the above query returns all isbn of the book table, but only returns the given size defined by the input parameter *size*.

Express:

Express is used as the actual server of the application. It is responsible for handling all get requests that are received and is able to handle different types of data routes that the client sends to the express server. The Express server implements three important functions which are used throughout the server-side application, being the `app.set`, `app.use` and `app.get` functions. The `app.set` functions sets the view engine (In this case pug), the `app.use` is also very important. The `app.use` connects the different routers to their designated route url in the format of:

```
app.use('/urlRoute', router);
```

Finally, the `app.get` is used to receive requests from the client side and is located in each router file. The `app.get` is responsible for sending and receiving data to and from the client side JavaScript.

Express-session:

Express session is used for storing session data about a client. This is very important as it allows scalability for the web application, mainly with user logging in. Normally, without implementing express session data, the program can only log in one user at a time, this is due to the inherit flaw with web based applications, being that the back end is separate from the front end, in this case the server side and client side are two different entities, and the server has a one to many relation with the front end. This means that if a second user were to log in, they'd be using the first user's server-side stored information when querying from the PostgreSQL database, which for obvious reasons is not ideal. Therefore, session data was implemented. The main purpose of session data is to keep track of information about a particular client-side user's session. More importantly, it is to keep track of a client-side user's *session id*, which is used to differentiate between two different sessions, and therefore result in zero conflicts occurring between users. In this case, whenever a user logs in their login information (i.e. email/username) is stored in an object as a key value pair, where the key is their unique session id, and the value is their login information. Therefore, this eliminates any conflicts with multiple users and allows the web application to be far more extensible in the future.

Pug:

Pug is used as the template engine for a few of the web pages. What a template engine is first, however is a template engine that converts a simplistic front-end html syntax into actual html source code. Pug also has lighter syntax than HTML code, therefore making it easier to write in. However, the main benefit of Pug over HTML is the ability for the server to both send a file (Pug) as well as JSON data for the client side to load up. Normally without the use of pug, this must be done in two separate requests, the first to request the new file, and the second to request the file's data. This would not have been an issue for this project as the goal of this project is not to create the most efficient program. However, for greater accessibility, the following should yield a result:

```
localhost:3000/search?title=Dragon&author=Chris
```

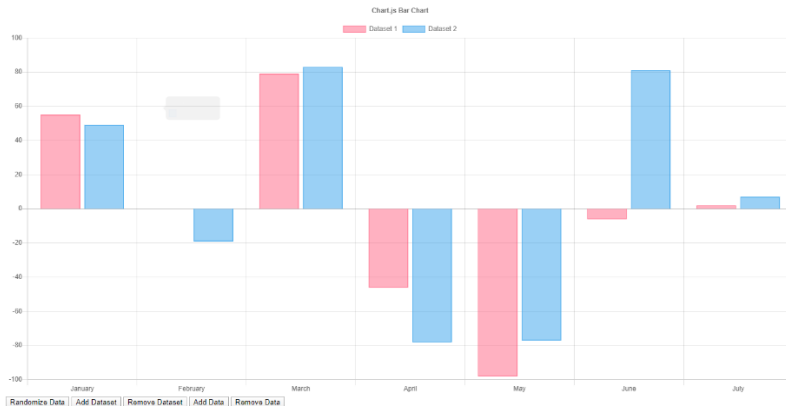
In the above scenario, the server should be able to recognize that a search is being done with input values "Dragon" for the book title and "Chris" as the author, and therefore return a page showing the results from the PostgreSQL query. However, this is an issue, as in order to do this efficiently, the server must send back both the url and the file in the same response, so PUG was used. What Pug is able to do in this case, is receive the query result and create html elements with it right inside the pug file, therefore increasing efficiency. Pug in this case was used for the two pages that involved unique search query results, being the book page found at `/book` and the search page found at `/search`.

Chart.js:

The Chart.js module is a front-end module and is not really "installed" or imported unlike the other modules described above but is instead included as a JavaScript file source. The source for the Chart.js file can be found at:

```
https://cdn.jsdelivr.net/npm/chart.js/2.7.3/Chart.min.js
```

What Chart.js allows for, is to create visually pleasing charts (More on this in section 6.6.3 below) that can store a wide variety of data, such as:



These charts are specifically used for the sales report page of the web application.

5.2.4. Code Design

This section will now describe the overall code design and in more detail, how everything connects together and how certain features are called and used within the program, as well as a more detailed look into the data flow described in section 5.2.1 above.

File system hierarchy:

The code is arranged into a hierarchy which separates out the server side files, and the client side files. The actual code base can be found in the git repo located in the /code/book_sanctum directory. Once in here, the topmost files are the server.js file as well as a data.js file (Which stores the user log in information). The web application also makes use of a package.json for module information. The directories inside the main directory include a routers directory, source directory and a sqlQueries directory. The routers directory contains all the router files used, where as the sqlQueries directory contains all the query files used. Finally, the source directory contains the front-end client-side source files. This directory is divided into two segments, being the client and pages directory. The client directory contains only client-side JavaScript files, whereas the pages directory includes all the front-end page files, which is further divided into three more directories, being the CSS, html, pug and themes directory. The CSS directory contains all the CSS stylesheets implemented into the html and pug directories, where each of those two include their respective file types. Finally, the themes directory contains the graphics of the application, mainly being the book sanctum logo and various other graphics used.

Server:

The server sets up the main Express server by implementing each of the routers using the app.use methods as well as the various page directories (For client, css, and html), and finally it implements the JavaScript file directories. Due to the server mainly implementing the routers, there is not much else to discuss for this segment.

Routers:

The app implements many different routers which each route to a specific URL defined in the server file described above. Each router file implements an express router to enable routing to different URLs using the main URL path in server, which is done through the module. Exports function. Each router also includes its corresponding query file, which can be found in the queries directory. The router file uses the query file's functions within its get functions to send and receive data from and to the postgres database server.

Queries:

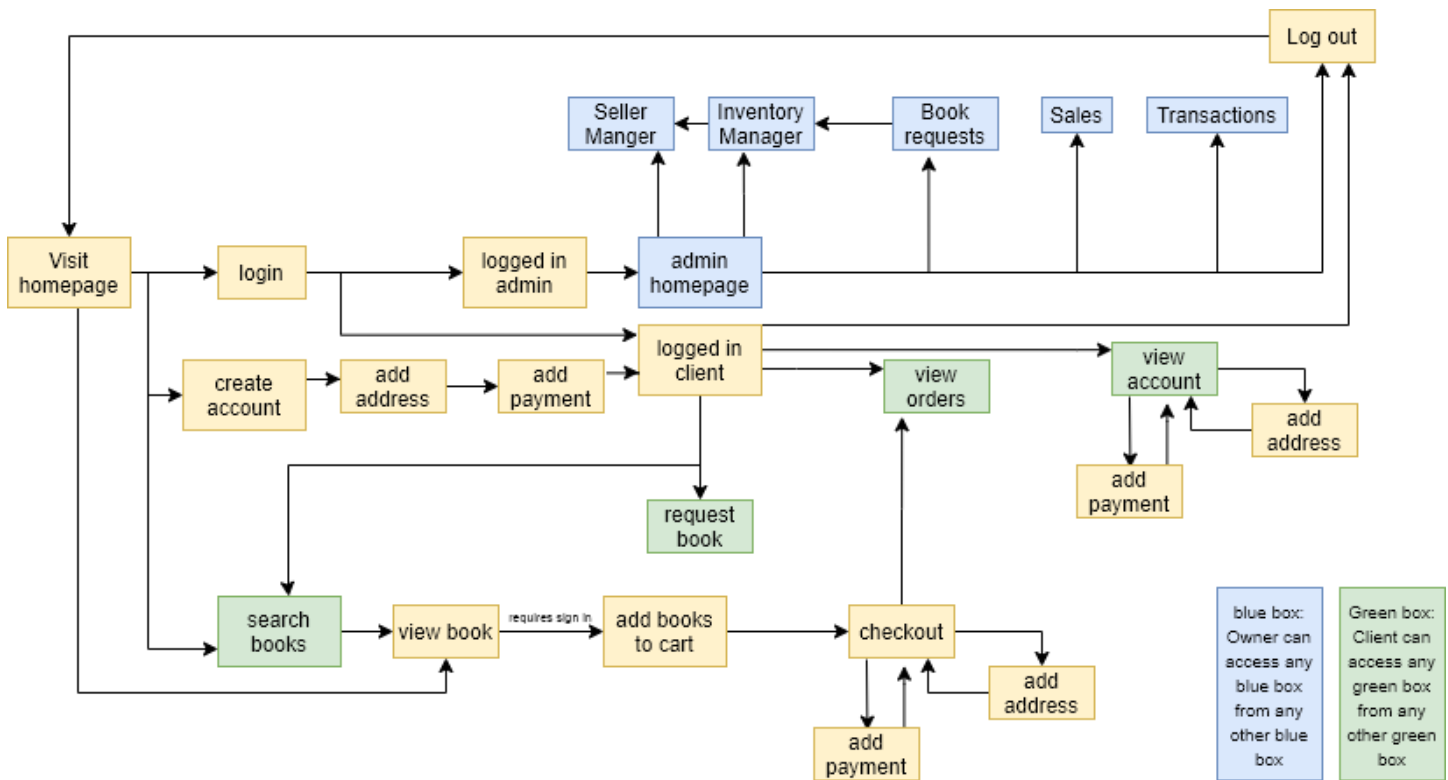
Each query implements the pg pool module described in section 5.2.3 above. As well as this, the structure of every query file is the exact same. Essentially, each file contains a function that contains a plethora of query functions (i.e. `addAddress()`, `getBooks()`, etc.). Each of these query functions is used for a specific query and goal in mind, which are all organized by their specialization type (i.e. getting book data, getting sale data, checking out, account information, etc.). Each query function can either be a promise (Meaning it sends data back to its router that called it), or send data directly to the client side using the response object parameter, if need be (As some functions do not need to send data back to the router but others do. I.e. getting books from the server does not require returning the books object to the router, so it is just send as a response to the client side, however the username information is needed when logging in by the router, so the query function must return the username information). For more information on the queries used, see the SQL directory in the git repo.). **Note: If you must see the queries, they are located here.**

Client Side:

The client side is relatively simple. It receives data from the server side as well as send data through get requests, which are then populated into HTML elements using innerHTML using elements by the creation of strings.

5.3. Workflow

This section showcases the workflow diagram which is used as a concise overview for what the user of the webpage can do.

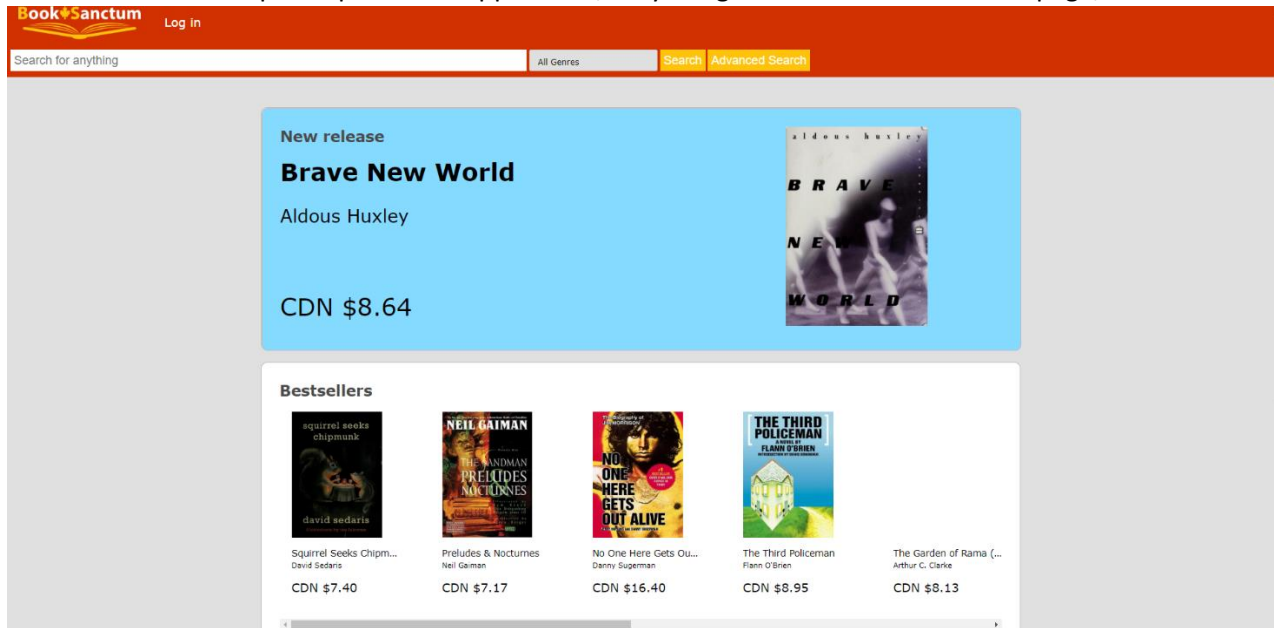


It should be noted however, that the user has the option of going to their specific homepage if they click on either the blue or green boxes described above.

5.4. Client

5.3.1. Home page


When the user first opens up the web application, they are greeted to the main home page, as seen below:



As seen above, and described in section 5.3, the user is able to either search for a book or view a book by clicking on either a new released book (The giant header in the center), or the bestsellers below the header. They are also able to log in to the website. When viewing newly released books, it automatically changes to another book in the header every 5 seconds (Like how real-world online stores do). The bestsellers list can also be scrolled horizontally to view more bestselling books. These two lists both get the book information using queries defined by order by (i.e. order by add_date for new releases and order by most sales for bestsellers). Note: Clicking on the Book Sanctum logo on the top left will reroute the store back to the home page seen above.

5.3.2. Log in


If the user decides to login, they are greeted to the following page:


Log in
Username

Password

☐ Log in as User

In this page, they either have the option of logging in as a user or as an admin/owner (See section 5.4 below). Once they press the login button, it sends a query to the postgres database which returns the username if it exists. If it doesn't, the page shows the following:


Log in
Incorrect username or password
Username


Password

☐ Log in as User

They can also create an account if they press the “create account” button.

5.3.3. Register Account

If the user decides to create an account, they are greeted to this page:



Create Account

Your first name

Your last name

Your username

Your email


Your password

Confirm password

Next

In here, they can type in the relevant user information such as their full name, username, email as well as password. And to make it a bit more realistic, the password must be confirmed through typing it in twice.

To satisfy one of the requirements of the application, being that the user must add their shipping and billing information on registration, the registration forces the user to add these two fields. Therefore, once this is submitted, they are then asked to add an address:



Add Address

Country/Region

State

City


Postal/Zip Code

Street Address

Apartment Number

Add Address

The address once again asks for all relevant information such as country, state, city, postal code, street address and apartment number. Once this is added, the application asks for a card to be added as payment:



Add Payment

Name

Card number

Expiry Date

Add Card

The card number also asks for relevant details. Once the user has added their card payment info, they are then greeted by the following page:

Book Sanctum

Review your Registration

Account info	Shipping	Payment
First name: Sharjeel	#9	Mike Myers
Last name: Ali	234 Mongo Drive	*****7392
Email address: LordofArbiters	Ottawa 54DL53	Expires: 02/22
Password: ****	Ontario Canada	

Complete Registration

Which showcases their account information they used to register. Note: For any of these “form pages”, if the user clicks on the Book Sanctum logo, they are brought back to the home page of the store shown in section 5.3.1. This is the only time the registration interacts with the postgres database, which in this case, it executes multiple queries to insert a new client, and their address and payment information. After they successfully create an account, they are logged into the store.

5.3.4. Logged in

Once the client logs in, they are welcomed back to the home page as seen below:

Book Sanctum Log out Account Orders Book Requests

Search for anything All Genres Search Advanced Search

New release

Breaking Dawn

Stephenie Meyer

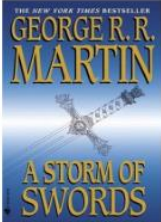


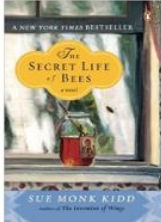

CDN \$22.23

Bestsellers

<p>Squirrel Seeks Chipmunk...</p> <p>David Sedaris</p> <p>CDN \$7.40</p>	<p>Preludes & Nocturnes</p> <p>Neil Gaiman</p> <p>CDN \$7.17</p>	<p>No One Here Gets Out...</p> <p>Danny Sugerman</p> <p>CDN \$16.40</p>	<p>The Third Policeman</p> <p>Flann O'Brien</p> <p>CDN \$8.95</p>	<p>The Garden of Rama (...)</p> <p>Arthur C. Clarke</p> <p>CDN \$8.13</p>
--	--	---	---	---

Note: The different book appearing in the new release header, as well as the top navigation bar information changing to “log out”, “account”, “orders”, and “book requests”, which they are now able to click on any one of these, as well as searching for books. They can also see a bonus feature below the bestsellers list, which is the recently viewed books (Described in section 6.3. below):

Recently viewed

				
A Storm of Swords George R.R. Martin	Neverwhere Graphic ... Mike Carey	The Stand Stephen King	The Secret Life of Bees Sue Monk Kidd	Eragon Christopher Paolini
CDN \$12.84	CDN \$11.99	CDN \$7.02	CDN \$6.48	CDN \$7.90

Once again, they are able to click on any of the books shown above to view their respective book pages.

5.3.5. Searching

The user can search for a book through two different methods. The first is by typing directly into the search bar as seen in the home page in the above sections. When the user types into the above search field, the application runs a query by matching the input text with either a book author or a book title using a fuzzy search implementation (See section 6.1 below). This means a user can type in a string such as “Tolkien Rings”, and the results will yield the Lord of the Rings trilogy for example, as well as J.R.R. Tolkien’s books. The other method of searching is through the Advanced Search button, to the right of the search button.

Book Sanctum

Advanced Search

ISBN

Title

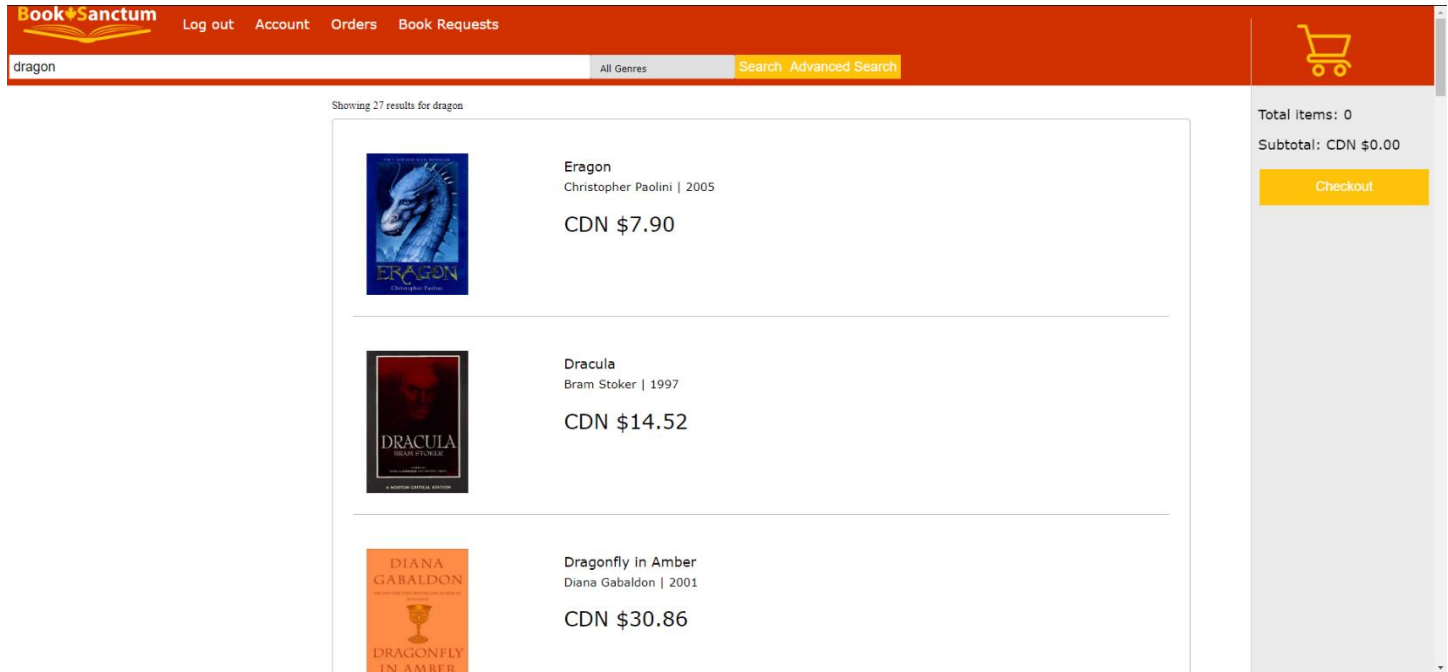
Author

Publisher

Published Date

In this advanced search page, the user has the option to search a book by one of the fields identified above, so if they choose they can search through a combination of the fields, such as searching for an ISBN and title, or searching for a publisher and genre, or an author. This allows greater flexibility for the user in terms of viewing the book catalogue.

Once they have submitted their search criteria, the application will show them the search page as shown below:



As shown above, the application retains its navigation bar, although the user is now on a different page. An empty cart can be seen to the right-hand side as well (Which will only display books if the user is logged in). The user is able to scroll through the outputted list of books on the screen and open a book's information page by either clicking on the book image cover or the book title. This also satisfies the requirements being that a user can search a book by author, genre, etc. and then they can select (Or in this case view) a book to show more information about it.

5.3.6. Book Information Page and adding to Cart

Once a book has been opened, the server executes a query to return every piece of data attached to a book (publisher, genre, and author information). The user can then view important information about the book, such as:

The screenshot shows the Book Sanctum website interface. At the top, there is a navigation bar with links for 'Log out', 'Account', 'Orders', and 'Book Requests'. Below this is a search bar with the placeholder text 'Search for anything' and buttons for 'All Genres', 'Search', and 'Advanced Search'. The main content area displays the book 'Squirrel Seeks Chipmunk: A Modest Bestiary' by David Sedaris, published in 2010. The book cover features a squirrel and a chipmunk. The price is listed as CDN \$7.40, and there is an 'ADD TO CART' button. A description of the book is provided, mentioning its unique blend of hilarity and heart. To the right, a shopping cart summary shows 'Total items: 0' and 'Subtotal: CDN \$0.00', with a 'Checkout' button.

Book Sanctum Log out Account Orders Book Requests

Search for anything All Genres Search Advanced Search

Squirrel Seeks Chipmunk: A Modest Bestiary
David Sedaris
2010

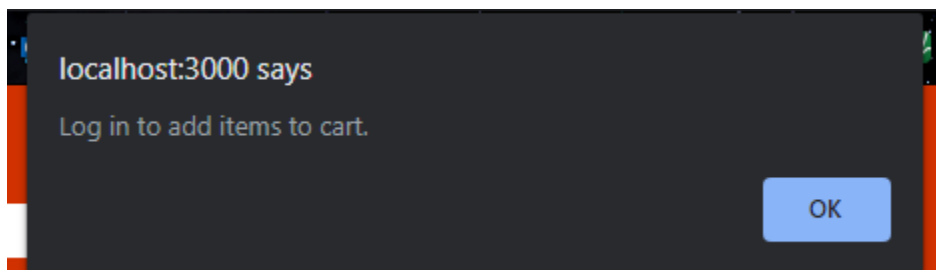
Featuring David Sedaris's unique blend of hilarity and heart, this new collection of keen-eyed animal-themed tales is an utter delight. Though the characters may not be human, the situations in these stories bear an uncanny resemblance to the insanity of everyday life. In "The Toad, the Turtle, and the Duck," three strangers commiserate about animal bureaucracy while waiting in a complaint line. In "Hello Kitty," a cynical feline struggles to sit through his prison-mandated AA meetings. In "The Squirrel and the Chipmunk," a pair of star-crossed lovers is separated by prejudiced family members. With original illustrations by Ian Falconer, author of the bestselling Olivia series of children's books, these stories are David Sedaris at his most observant, poignant, and surprising.

CDN \$7.40
- 1 +
ADD TO CART

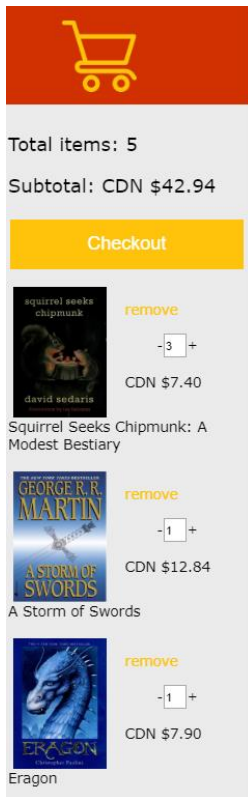
ISBN: 9780316038393
Publisher: Little, Brown Young Readers
Genres: short-stories
Pages: 176
Date added: 2020-04-07

Total items: 0
Subtotal: CDN \$0.00
Checkout

As shown above, this page displays the book title, author, published year, description, price, isbn, publisher, genre, pages and the date that it was added into the catalogue. They are also able to add a book to their cart by clicking on the "Add to cart" button on the left, as well as specifying the quantity to add. However, they must first be logged into the bookstore to add books to their cart, which was another requirement of the project. If they are not logged in, the following appears as an alert box:



Which tells them they must log in to add books to their cart. However, once a book has been added to the cart, the cart tab on the right will show the following:






They can also remove books from their cart (Which sends a query to set the quantity of their book cart to 0, and then a trigger removes the book from the cart if its quantity is 0), or increase or decrease their quantity, which changes the subtotal price automatically. They are also able to open the book information page by once again clicking on the book image shown above. If they wish, they can click the “checkout” button above to checkout and complete their order or continue to browse and add books to their cart.

5.3.7. Checking out

If the user decides to checkout, they will see the following page:

Review your order

By ordering you agree that your items will never arrive to the destination.

	<p>The BFG</p> <p>CDN \$10.99</p> <p>Qty: 2</p>	<h3>Shipping & Billing</h3> <h4>Shipping options</h4> <div>212 La'phreu Drive Add new Shipping</div> #12 212 La'phreu Drive Ottawa M4G5L6 QC Canada <h4>Payment options</h4> <div>Card ending in 5342 Add new Payment</div> Sharjeel Ali *****5342 Expires: 02/22
	<p>Eragon</p> <p>CDN \$7.90</p> <p>Qty: 1</p>	
	<p>The Tale of Peter Rabbit</p> <p>CDN \$7.71</p> <p>Qty: 3</p>	

Order Review


Total Items:	6
Total Price:	\$53.01
Shipping:	\$3.00
Subtotal	\$56.01
Tax	\$7.28
Total	\$63.29

ORDER NOW

On this page, they will see all relevant information about their new order such as the books ordered, their quantity and their shipping and payment options. They can also add new shipping or payment information, which fulfills the specified requirements where the user can insert different shipping or billing information than what was used from the registration. If they do decide to add new shipping/payment, they will see the same form which was shown in the user register section above for either adding an address or payment, but now once they click the “add” button, it will take them right back to the checkout page. Once the user is content with their order, they can press the “order now” button to complete their order, which sends a query to the postgres server to insert a new book order, then they are redirected to the order page described below.

5.3.8. Orders

The orders page shows a record of each book that has been ordered and with what order, as well as tracking information as seen below:


[Log out](#)
[Account](#)
[Orders](#)
[Book Requests](#)



Squirrel Seeks Chipmunk: A Modest Bestiary
 David Sedaris
 CDN \$7.40
 Qty: 3



Eragon
 Christopher Paolini
 CDN \$7.90
 Qty: 1




A Storm of Swords
 George R.R. Martin
 CDN \$12.84
 Qty: 1

As seen, the order shows 3 books that have been ordered as well as the order date, total price, the shipping address, the card that was used, the order number and the status, which is the tracking. The tracking number which is the order is assigned a status number, so to keep the tracking updated. This also fulfils another requirement, being that each order must show tracking information (Which in this case is the status).

5.3.9. Account

The user also has the option of viewing their account information (Currently, it is just their shipping and payment information) shown below:


[Log out](#)
[Account](#)
[Orders](#)
[Book Requests](#)

Add new address

#12
 212 La'phreu Drive
 Ottawa M4G5L6
 QC Canada

#43
 654 Lamberson Drive N
 Ottawa K1S4L5
 ON Canada

Add new card

Sharjeel Ali
 *****5342
 Expires: 02/22

Sharjeel Ali
 *****3233
 Expires: 09/30

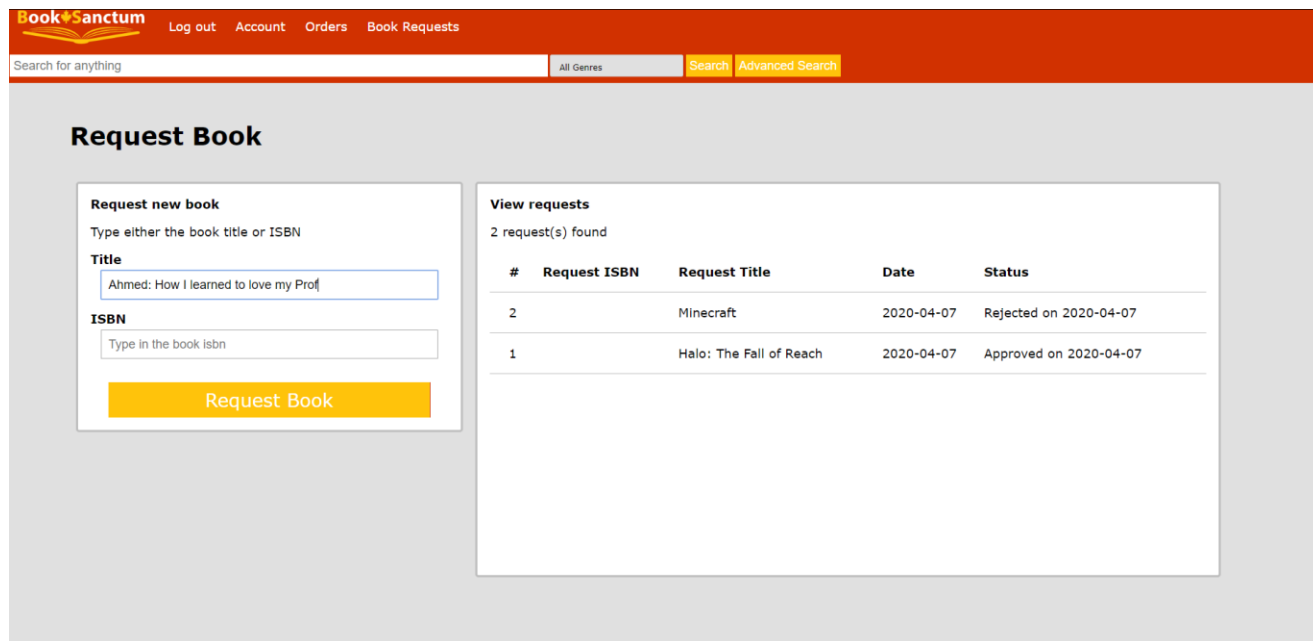
Sharjeel Ali
 *****2222
 Expires: 08/22

Sharjeel Ali
 *****3232
 Expires: 06/34

They once again have the option of adding a new address or card to their account, which when added using the same forms described above, will take them directly back to the account page.

5.3.10. Requesting a book

As a bonus feature, the client is also able to request a new book to be added to the store catalogue by specifying a book's isbn number or a book's title. This is then able to be viewed by the bookstore owner (Described in section 5.5.). For more information on requesting a book, see section 6.5. below.



Request Book

Request new book
Type either the book title or ISBN

Title
Ahmed: How I learned to love my Prof

ISBN
Type in the book isbn

Request Book

View requests
2 request(s) found

#	Request ISBN	Request Title	Date	Status
2		Minecraft	2020-04-07	Rejected on 2020-04-07
1		Halo: The Fall of Reach	2020-04-07	Approved on 2020-04-07

As seen above, the client can see their past book requests as well as the current status of them, such as the book being rejected (i.e. Minecraft), or approved (Halo). They can also request a new book on the left where they input the book title or isbn. Once a new request is added and an insert done through executing a query function, the client will see the following:

View requests
3 request(s) found

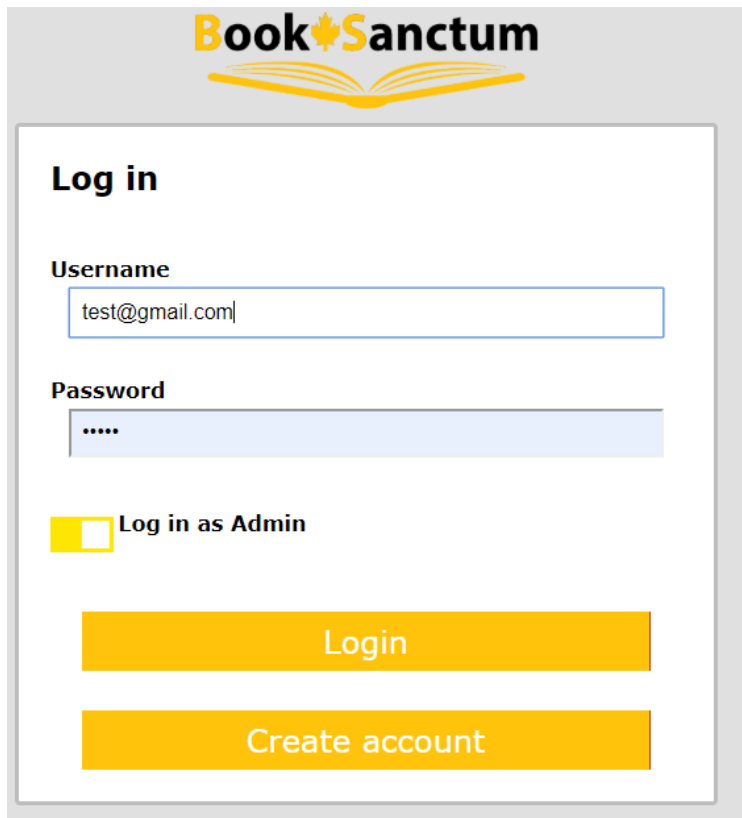
#	Request ISBN	Request Title	Date	Status
3		Ahmed: How I learned to love my Prof	2020-04-09	N/A
2		Minecraft	2020-04-07	Rejected on 2020-04-07
1		Halo: The Fall of Reach	2020-04-07	Approved on 2020-04-07

Now the new book *Ahmed: How I learned to love my Prof* shows a status as N/A, as it has not been decided yet to add or reject the book by the owner.

5.5. Owner

5.5.1. Logging in

The admin is able to log right from the client home page by clicking the log in button showcased above in section 5.4. They are then able to toggle the “log in” switch to show the following:

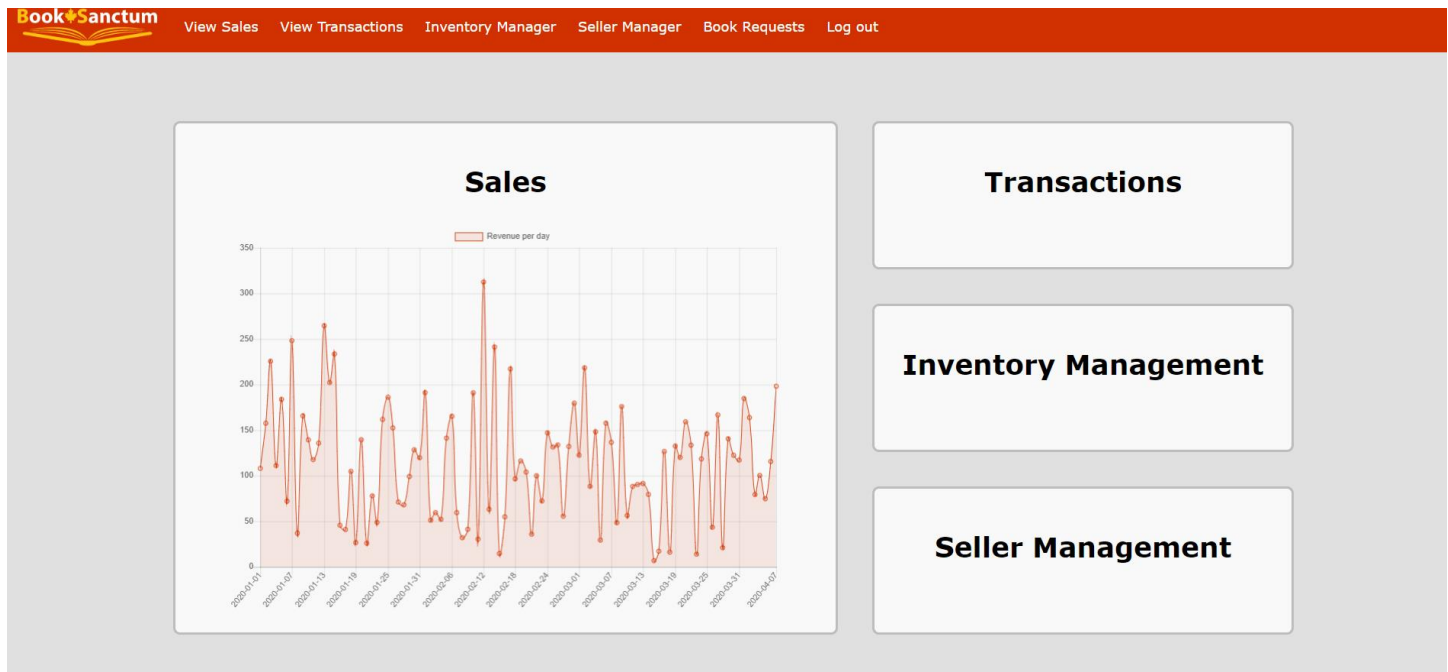


The screenshot shows the 'Book Sanctum' logo at the top, featuring the text 'Book Sanctum' with a yellow book icon. Below the logo is a 'Log in' section. It contains a 'Username' label followed by a text input field containing 'test@gmail.com'. Below that is a 'Password' label followed by a password input field with six dots. There is a checkbox labeled 'Log in as Admin' which is currently unchecked. At the bottom of the form are two yellow buttons: 'Login' and 'Create account'.

One key difference between an admin and client in terms of logging in is that an admin does not have a username, where as a client does, therefore an admin must only use their email, unlike a client which can use both. This also enables easy access to both admin and client (I.e. for debugging, the postgres database has an admin and client account that use the same email, so it is faster to switch between them). Once an admin has logged in, they are greeted to the home page specifically made for the admin side.

5.5.2. Admin Home Page

Once the admin logs in or clicks on the Book Sanctum logo on the top left of the navigation bar, they are redirected to the following page listed below:

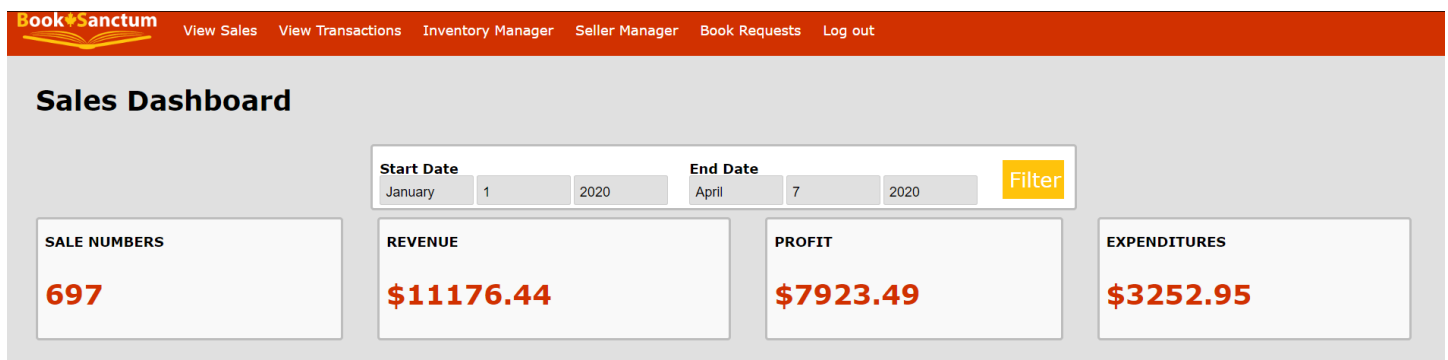


As seen above, the admin home page displays four categories they are able to view, which are the sales, transactions, inventory management, and seller management. The sales chart shown above displays the total sales for the entire duration of the bookstore's history. The admin can also choose to also click on the navigation bar on the top to select between different menu interactions and options.

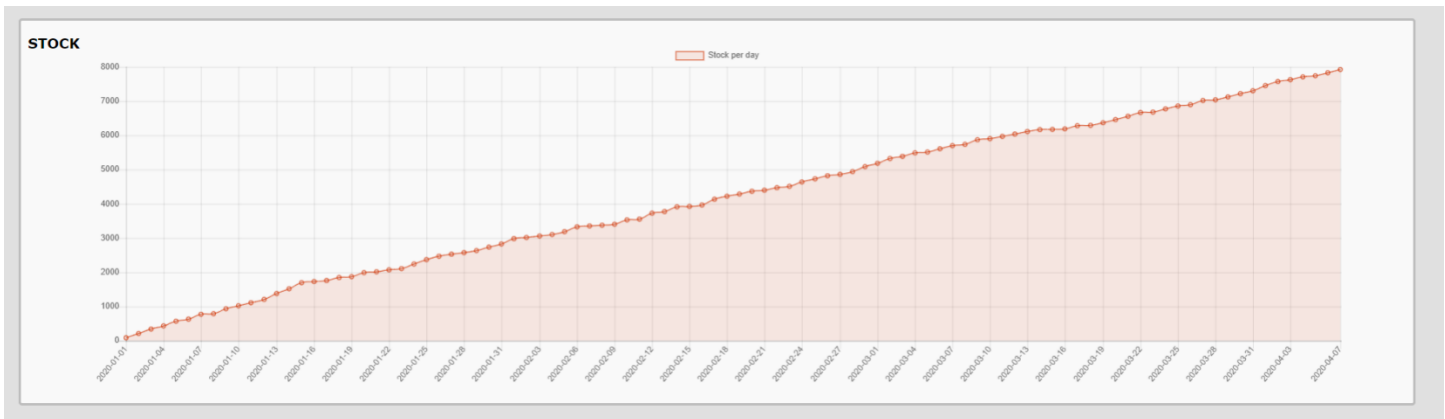
5.5.3. Sales

If the publisher clicks on the sales page shown above, they are brought to a full in-depth page listing of all the books sales and various charts shown below in a nice concise sales report. The admin can also filter the sales between two different dates (start and end), so for example if the admin wishes to view the book sales from March 10th to March 15th, they are able to do so (Which is also described as a bonus in section 6.12.) This was implemented by the creation of a view which contains all the various types of transactions an admin can view (See section 6.10 for a more in depth explanation).

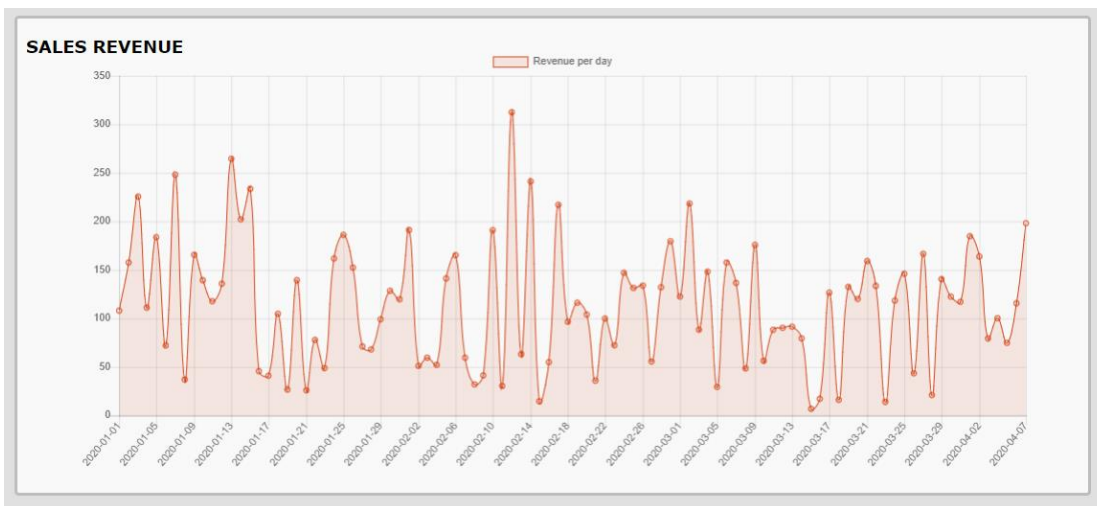
The first thing the admin will see is the following four sales numbers:



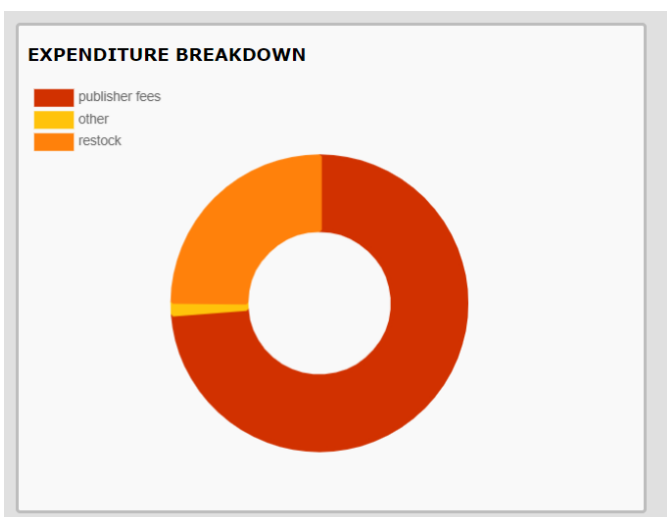
If they scroll down, they can also view the total book sanctum stock to date:



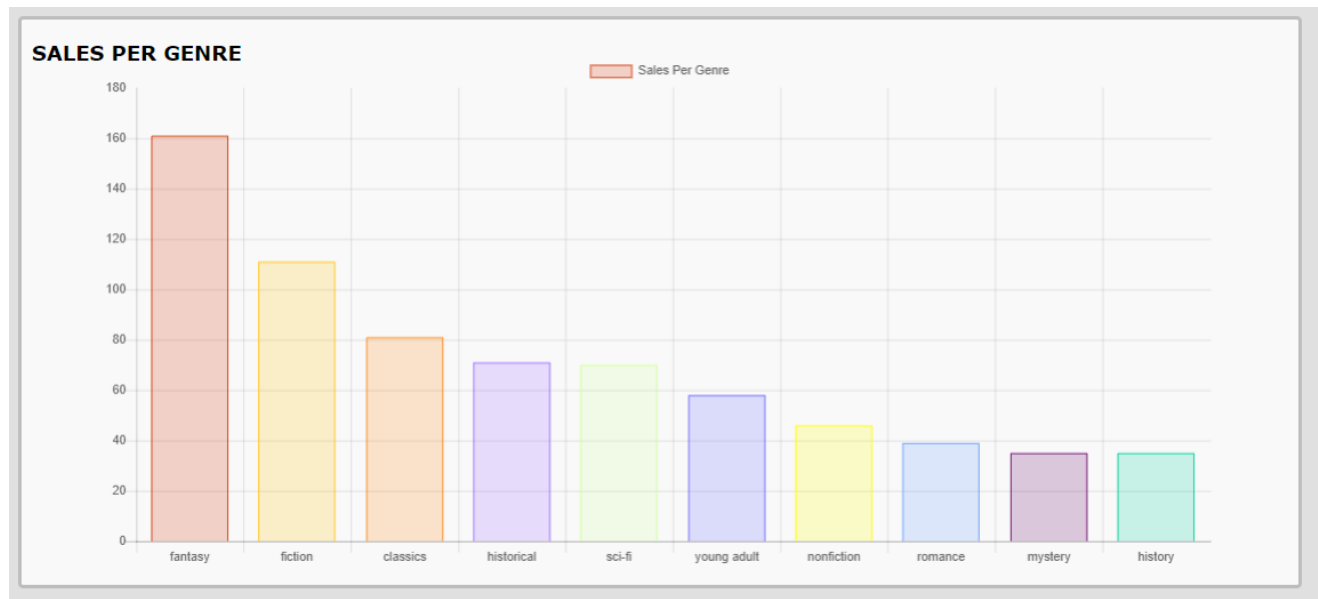
Note: it is a linear graph due to creating a script to add a set random amount of orders with a set random number of books each day, so the linearity is expected. Below this chart is another chart which was the exact same shown in the home page, which showcases total sales revenue per day:



There is also a breakdown chart of all the various types of expenditures that have been done (Restocking fees, more on this in section 6.9 and section 5.5.4, and other transactions, more on this in section 5.5.4):

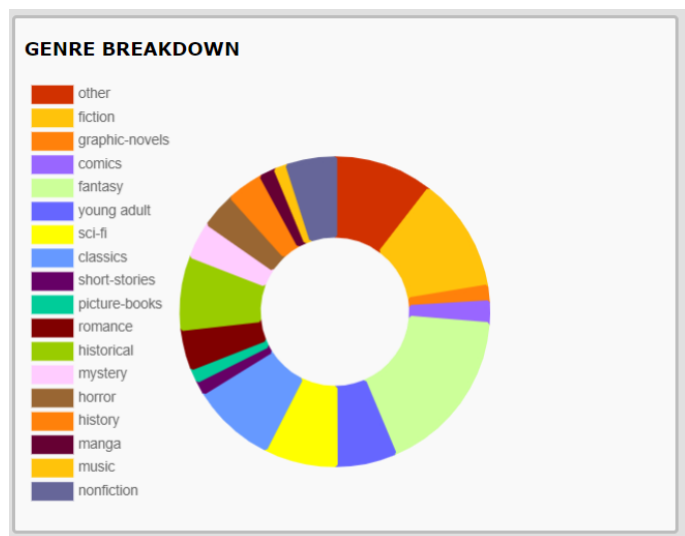


Below this is once again, another chart showcasing sales per genre (Which was a suggested requirement of the application), in this case showing the top 10 most sold genres by number of book sales. This was done by a query that groups book orders by the quantity of orders completed:



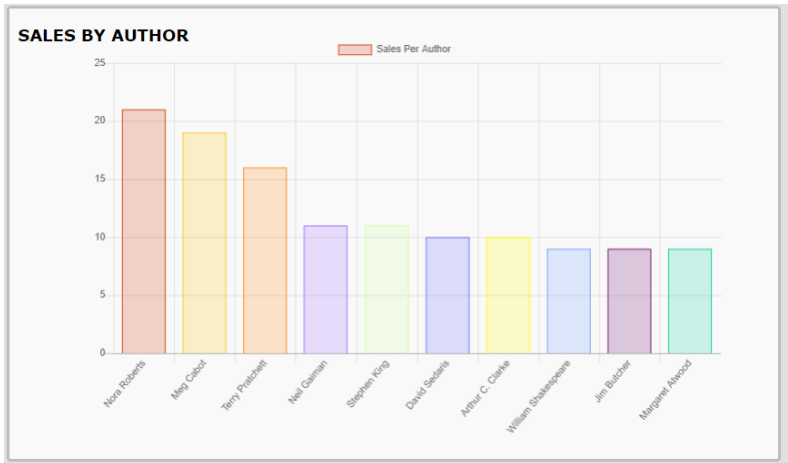
Note: This data is not truthful, this is due to the database containing mostly fantasy, and with the random insert orders script, it is likely to favor fantasy more.

The other chart beside genre, is the genre breakdown chart, showcasing the distribution of book sales by genre, which uses a similar query to the one described above:

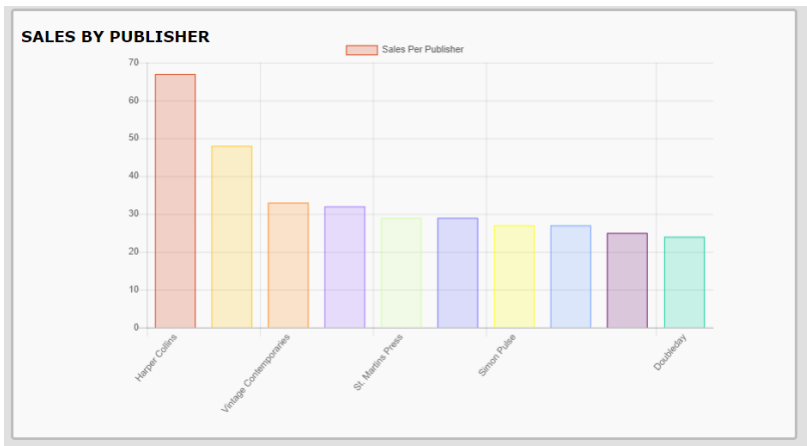


As seen above, note the other field. This specifies any genre where the distribution of genre sales is <1% (i.e. only <1% of the books sold were this genre).

Below this is the last row, showcasing sales by author, which uses the same query used for genre above, but this time arranged to select authors instead:

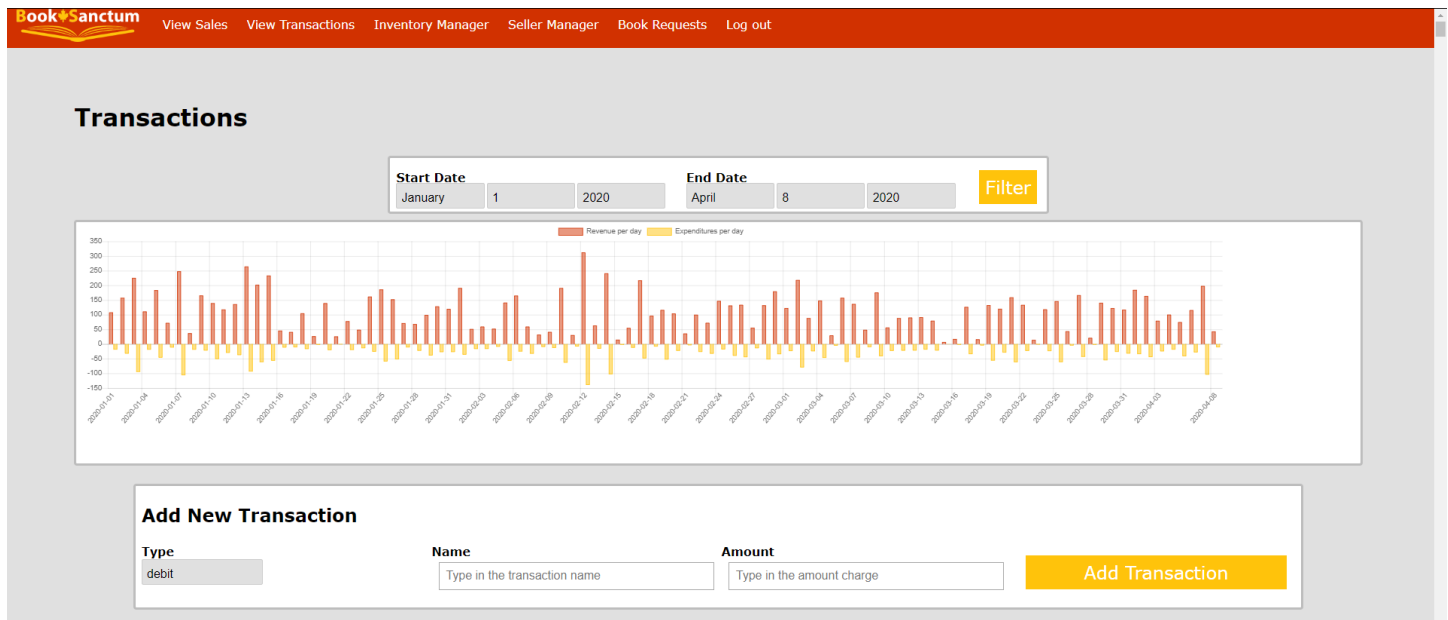


And the last, being publisher:



5.5.4. Transactions

This page is a bonus page. It is meant to showcase every transaction that the bookstore has ever gone through, once again using the same view described in section 5.5.3 above as well as the section 6.10. below. When the admin opens this page, they are greeted to the following:



The giant chart in the middle above showcases both expenditures and sales per day, similar to how online banking showcases each day's purchases as debit or credit transfers of money (Such as cheques or paystubs). The admin can once again filter through the start and end dates to see specific sales on certain dates, as well as add a new transaction as either a credit or debit transaction. Once the admin clicks the "add transaction" button, the server executes a query to insert data into the transaction table, defined in section 2, 3 and 4 above, this is also described in more depth in section 6.8. as a bonus feature of the application. If the admin scrolls down, they can see an actual list of their transactions:

697 Transactions listed				
Date	Transaction	Debit	Credit	Profit-to-Date
2020-04-07	PUBLISHER FEES: Grove Press	-16.75		7940.24
2020-04-07	RESTOCK: Arrow: 9780099297703 x6	-9.94		7950.18
2020-04-07	BOOK SALE: 9780872205543		36.36	7913.82
2020-04-07	BOOK SALE: 9780802136831		69.80	7844.02
2020-04-07	PUBLISHER FEES: David Fickling Books	-7.95		7851.97
2020-04-07	BOOK SALE: 9780099297703		43.20	7808.77
2020-04-07	BOOK SALE: 9780879234621		33.12	7775.65
2020-04-07	PUBLISHER FEES: Arrow	-9.94		7785.59
2020-04-07	BOOK SALE: 9780062076106		15.68	7769.91
2020-04-07	OTHER: rent	-45.00		7814.91

As shown above, the design was based off of online banking, which showcases each transaction similarly. Also take note of the various types of transactions such as "other", "book sale", "publisher fees" (Used to transfer a percentage of sales to the publisher, which are declared randomly upon book search), and "restock", which showcases the "fake" method of how the bookstore emails the publisher requesting an order of the quantity shown (In this case, it shows a restock order

for publisher Arrow for 6 books of the specified ISBN, which costed the bookstore \$9.94, therefore fulfilling the restock requirements, as this also means 6 books were ordered in the previous month). (More on restock in section 6.9. below).

5.5.5. Inventory Management (Adding books/removing books)

The admin can also view their inventory or book catalogue instead of relying on searching it in the public interface. This is also where the admin can add new books to their catalogue or remove books, which fulfills both of the add/remove requirements required for the project. As seen below:

Book Sanctum View Sales View Transactions Inventory Manager Seller Manager Book Requests Log out

Inventory Management

Add book

Title
Halo: The Fall of Reach

ISBN
Type in the book ISBN

MSRP
Type in the MSRP

Seller Deduction %
Type in the %

Pages
Type in the page coun

Year
Type in the book year

Select Genre
animals

New Genre
Type in the genre

Select Author
A.A. Milne

New Author
Type in the author

Select Publisher
47North

Add Publisher

Description
Type in the book description...

Add Book

Book Information

Rise of a Merchant Prince
Raymond E. Feist
1996
Added: 2020-04-07
ISBN: 9780006497011
Publisher: Harper Collins
Genres: fantasy
Pages: 479
Stock: 10
CDN \$26.01

Surviving the wrath of the fearsome Sauur-a hideous race of invading serpents-noble Erik and cunning Roo have delivered a timely warning to the rulers of the Midkemian Empire, and are now free to pursue their separate destinies. Erik chooses the army-and the continuing war against Midkemian's dread enemies. Roo lusts for wealth and power-rising high and fast in the world of trade. But with luxury comes carelessness and a vulnerability to the desires of the flesh. And a beautiful seductress with her ruthless machinations threatens to destroy everything Roo has built and become-summoning catastrophe into his future... and terror into his world.

Remove Book

2699 books listed

9780002314572

9780006473299

9780006479758

9780006479901

9780006480099

9780006486015

9780006497011

9780006513223

9780006551812


9780007118892

9780007119356

As seen above, the admin can browse every single book in their collection sorted by the book ISBN, they can then click on the ISBN to open the book information which displays the same information a user will see in the public page, however, this page also shows the book stock (For the book above it is 10). The admin can then choose to remove a book from their catalogue by clicking the "remove book" button. In this case, removing a book was an issue in terms of design. The original design included deleting a book entirely from the database, however this would also mean deleting orders that contained the same book, so this was an issue that had to be fixed. Therefore, the solution was to create an attribute *removed* which allows the server to update the specified book by isbn's removed bool to be true, meaning the book is now hidden from the public search and will not appear. As for adding a new book, the admin can type all the necessary information in the above field. They can then either select a genre or add a new one, and same for author. However, for publisher, due to the extensive information required, a publisher must be added in a separate page (Seller page) . After the button to add is clicked, an insert query is executed to insert the book information.

5.5.6. Seller Management (Adding publishers)

This page allows the admin to view the list of sellers/publishers that the bookstore presumably has a contract with:


[View Sales](#)
[View Transactions](#)
[Inventory Manager](#)
[Seller Manager](#)
[Book Requests](#)
[Log out](#)

Seller Management

Add Publisher

Name

Phone

Email

Banking Routing #

Banking Account #

Unit #

Street

Postal Code

City

Region

Add Publisher

Publisher Information

Name
 Adamant Media Corporation

Phone
 918-702-4538

Email
 AdamantMediaCorporation@worldpublishers.com

Direct Deposit

Routing #: 130508085

Account #: 307601526

Address
 206 8234 Fairfield Drive
 T9A8T6 Wetaskiwin
 AB Canada


277 Publishers listed

Name
47North
Abacus
Ace Books
Adamant Media Corporation
ADV Manga
Akashic Books
Aladdin Paperbacks
Alfred A. Knopf
Algonquin Books
Allison & Busby
Amistad
Anchor Books

As seen above, it is very similar in design to the inventory manager page described above. The admin can view different publishers as well as insert a new one into the database (Which is done through multiple queries for adding address, then publisher itself).

5.5.7. Admin book requests

(Bonus described in section 6.5. below) The admin has the ability to also view all book requests done by the bookstore clients. As discussed in section 5.3.10 above, the client can request for a book to be added, which the admin can see:


[View Sales](#)
[View Transactions](#)
[Inventory Manager](#)
[Seller Manager](#)
[Book Requests](#)
[Log out](#)

Book Requests

View requests

4 request(s) found

#	Username	Request ISBN	Request Title	Date	Status
4	LordofArbiters		Dark Web Chronicles	2020-04-09	Approve Reject
3	LordofArbiters		Ahmed: How I learned to love my Prof	2020-04-09	Approve Reject
2	LordofArbiters		Minecraft	2020-04-07	Admin Ali rejected on 2020-04-07
1	LordofArbiters		Halo: The Fall of Reach	2020-04-07	Admin Ali approved on 2020-04-07

The admin then has the option to either reject the book or add it into the database. If the book is rejected, the status will change for both client and admin to “rejected”, however if the admin clicks the approve button, they will be redirected back to the inventory manager page:

The screenshot shows a web form titled "Add book". It contains several input fields and buttons. The "Title" field is pre-filled with "Halo: The Fall of Reach". The "ISBN" field has a placeholder "Type in the book ISBN". The "MSRP" field has a placeholder "Type in the MSRP". The "Seller Deduction %" field has a placeholder "Type in the %". The "Pages" field has a placeholder "Type in the page coun". The "Year" field has a placeholder "Type in the book year". The "Select Genre" dropdown menu is open, showing "animals" as the selected option. The "New Genre" field has a placeholder "Type in the genre". The "Select Author" dropdown menu is open, showing "A.A. Milne" as the selected option. The "New Author" field has a placeholder "Type in the author". The "Select Publisher" dropdown menu is open, showing "47North" as the selected option. There is an "Add Publisher" button. The "Description" field has a placeholder "Type in the book description...". At the bottom of the form is a large yellow "Add Book" button.

However, the important thing is the title. If for example, the admin wants to approve the book request with title “Halo the fall of reach”, the title will be prefilled and unable to be changed due to it being tied to a book request. The same can be said for ISBN as well. Then the admin can type in the book information normally. After this, the book will appear as “approved” under the book request page for both client and admin.

6. Bonus Features

The website contains a wide variety of bonus features that have all been implemented in the codebase described above. In this section, a detailed overview of each bonus feature will be described.

6.1. Fuzzy search

The website implements a fuzzy search variant for searching for books through both the regular search bar and the advanced search page. This feature is very important for the search algorithm due to enabling related book searches. I.e. if the user searches “Lord of the Rings”, the book search will yield results for the Lord of the Rings trilogy; Fellowship of the ring, the two towers and Return of the King (Assuming the book titles contain words from “Lord of the Rings”). Another feature of this fuzzy search mechanism is it allows non exact character searches, i.e. if the user searches “Lord of the Rings”, then it will also yield results for book titles containing words such as “Load, Rim, etc.”. This also has another benefit, being searching for authors. Due to names sometimes being complicated to spell out, it enables the query to yield results for the intended author name. I.e. If the user searches for books written by author “J. R. R. Tolkien”, however they type “J. Tolkin” Instead into the search bar, the query will still return “J. R. R. Tolkein” as a result.

There is also another benefit for fuzzy searching, for this particular implementation of the bookstore. Due to the store only containing 2699 books by default, this means many searches will yield very few results, and therefore will not replicate a good user experience that is expected when searching through an online store. Therefore, implementing fuzzy searching allows the store to be more “populated”, although it is not.

The fuzzy search implementation is done through the following query:

```
select book.isbn,book.title,book.price,book.published_date,
author.name as author, genre.name as genre_name from book
inner join author on author.id = book.author_id
inner join genre on genre.id = book.genre_id
inner join publisher on publisher.id = book.publisher_id
where genre.name like '%fantasy%'
and book.published_date like '%2005%'
and (length('') = 0 or similarity(book.title,'') > 0.15 )
and (length('chris') = 0 or similarity(author.name,'chris') > 0.15 )
and (length('') = 0 or similarity(publisher.name,'') > 0.15 );
```

The important aspects of the above query are the similarity(PARAMATER_1, PARAMATER_2)>num function. The fuzzy search algorithm implements an extension known as pg_trgm, which uses trigrams to do fuzzy searching. Trigrams are very useful for approximate string comparisons, as they break apart strings into groups of characters (i.e. splitting “Ring” would result in “Ri”, “Rin”, “ing”). These trigram groups are then compared against each other, presumably using exact string matches for each group. Postgres then returns a number between 0 and 1, which describes how similar the trigram groups are (i.e. if 2 out of 3 trigrams groups are exact, it will return 0.6). Using this with the > operator can be used to determine approximate string matches. For the query shown above, 0.15 is used as the threshold to compare to. So, for a similarity() match, if the returned result is larger than 0.15, the query will include the result in the final search.

Another fuzzy search type is the % % matching search. This search type works relatively similar to *string.contains* in other programming languages. The reason why another trigram search wasn’t used for genre and published_date is due to the

way the website implements the input for published_date and genre name. Due to genre name being an input from a drop down select box, it already has predefined values (So no approximate trigram needed), and for year input, it is just a 4-digit number. Another important note is the reasoning for why a fuzzy search is required for genre and year instead of exact string matching. This is in the case if the user inputted a blank string. Using fuzzy searching in the form of %, blank strings do not need to be checked (As they are for similarity trigram checking using the length()=0 value). Therefore, implementing a fuzzy search allows a much better user experience.

6.2. Bestsellers

The home page contains a list of bestselling books, which is used to fill the home page. This aspect is important for an online store to advertise easily selling products, as well as showing what items are most relevant at the given point. Another important aspect about having a bestsellers list is to show suggestions to the user, in case they do not know what exactly to search in the bookstore.

The query used to return the best-selling book results is listed below:

```
select book.isbn, book.title, book.price, author.name from
(select isbn, sum(quantity) as sales from order_book
group by(isbn) ) as sold
inner join book on sold.isbn = book.isbn
inner join author on book.author_id = author.id
order by sales desc
limit 10;
```

The query to handle bestselling books is relatively simply, and involves returning the top 10 books, ordered by their sales (Which in this case is the sum of all book orders by their quantity). It also returns other attributes that are important to the store such as isbn, title, price and name, which are all used for displaying the bestsellers on the front page.

6.3. Recently viewed

The home page, along with displaying bestselling books described in section 6.2, also displays recently viewed books that the user has viewed. Viewing a book in this scenario is defined as opening a book page located at the URL address /book?isbn=123456789012. The importance of implementing a recently viewed list is to allow the user to view their past items they have viewed, and acts as a sort of “wishlist” (Due to an actual wishlist not being implemented). Another importance of a recently viewed section is to fill out the home page, otherwise it may seem blank.

The original design of recently viewed was to use javascript’s session data which is able to store information on the browser session, or use JavaScript window localStorage, which stores information permanently for the specific browser. This would have been very useful to implement as it allows a user to view their history without the need for signing into their account. However, this would not have taken full advantage of the postgres database, so this design was implemented into the schema described in section 1. The recently viewed table is named the *view_history* and is defined in the DDL as the following:

```
create table view_history(
    username          varchar(20),
    isbn              varchar(13),
    rank               numeric(1,0) not null,
    primary key (username, isbn),
```

```

foreign key (username) references client,
foreign key (isbn) references book
);

```

As seen above, the table `view_history` is defined by its two primary and foreign keys, *username* and *isbn* as well as having another very important attribute, being the *rank*. The rank itself enables the use of tracking the client's view history. The basic design behind this table is that whenever a client views a book page, the client side JavaScript will send a get request to the server, which calls the query function to insert into `view_history` with the new book's isbn and the current username of the client. Each time a book is inserted into `view_history`, its rank is set to 0, then a trigger is called which updates every other book in `view_history` belonging to the client by updating the rank by 1 level. This can be described as the following:

isbn	rank
A	0
B	1
C	2
D	3
E	4

Now, when a new book *F* has been viewed by the user, the original table's ranks are all updated by one to make space for the new book *F* to be inserted:

isbn	rank
F	0
A	1
B	2
C	3
D	4
E	5

Now, if the table looked like the following:

isbn	rank
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9

If a new book *K* is inserted, it would result in book *J* having a rank of 10, which is not allowed as the numeric DDL is described as having only 1 digit. Therefore, the book *J* is removed from the table, and all ranks updated as usual:

isbn	rank
K	0

A	1
B	2
C	3
D	4
E	5
F	6
G	7
H	8
I	9

The final case that occurs for `view_history` is when the client views a book that already exists within the `view_history` table.

Going back to the following example:

isbn	rank
A	0
B	1
C	2
D	3
E	4

If the client views the book *C*, there would be conflicts as book *C* already exists in the table. Therefore, book *C* is removed from the table, and only the books that have a rank less than that of book *C*'s original rank (Before removal) are updated:

isbn	rank
C	0
A	1
B	2
D	3
E	4

This entire algorithm is described in the function `update_view_history()`:

begin

```

    if (new.isbn in (select isbn from view_history where username= new.username)) then
        update view_history
        set rank =rank+1
        where username = new.username and rank< (select rank from view_history where username=
new.username and isbn = new.isbn);

```

```

    delete from view_history
    where username = new.username and isbn = new.isbn;

```

else

```

    update view_history
    set rank =rank+1
    where username = new.username;

```

```

    delete from view_history
    where rank >=9 and username = new.username;

```

```
    end if;  
    return new;  
end;
```

The function is split into two cases, the first being if the book previously exists in the `view_history`, and the second being if it is a new book that will be inserted. This function is called using the trigger `trig_view_history` which is called before each `view_history` insert.

6.4. Recently released

Lastly for the home page, it also displays the recently released books at the very top. Recently released in this scenario is not defined by the published date, but rather the add date of the book. This is to ensure this feature is useful in that, if the owner inserts a new book to the database using the inventory manager (Discussed in section 5.4), the clients of the bookstore are all notified of a new book addition. Another benefit of having a recently released section is for the design of the home page, as most online webstores have a large section dedicated to some of their items, in this case it was decided this would be the recently released books.

The query used to return the recently released book results is listed below:

```
select book.isbn, book.title, book.price,author.name  
from book left join author on author.id = book.author_id  
order by book.add_date DESC  
limit $1;
```

The query itself described above is very simple and just orders the book by their `add_date` value and limits the results to the input variable `$1`, which for the app is 3 (i.e. only 3 books appear in the new release header).

6.5. Requesting books

The request book feature was an idea that was thought for the purposes of giving the client a way to send feedback to the bookstore. In this case, an online bookstore would greatly benefit from allowing the clients themselves to request new book additions to the store. This also realistically gives the owner other criteria in looking at the future additions to the bookstore, as they could examine both book sales as well as what types of books users are requesting to be added. Another important reason for the request book implementation was giving a direct way for owners and clients to communicate to each other, as normally, this is done when a client places a book order, but then the owner themselves cannot send data back to the client to view. In the case of the request book implementation, the client simply requests a book, then the owner can view the request and either send back an approve/decline to the user as a way of two-way communication. Lastly, the request book feature allows the owner/admin entity described in section 1 to be connected with everything else in the ER diagram, as removing request book forces the admin entity to be singular and not have any relations with other entities (Similar to the transaction), which is not very ideal for an ER diagram.

The request book feature was implemented using multiple relations as shown in section 1. The implementation involves the client having a relation with the `request_book` entity, which then has another relation with the admin entity for when the admin is able to decide if a book should be added to the bookstore or not. The relation with admin only exists when the admin makes a decision through either clicking the “approve” and adding the book through the inventory manager page, or “reject” buttons. Once an admin-request_book relation exists, the client and admin is then able to view the decision shown in their respective request_book pages as either “approved” or “rejected”. Another feature added was the addition of the admin request book page showcasing which admin made the decision. This is useful for

extensibility in the future, if multiple admins exist as it provides a control structure (i.e. each admin knows which admin is responsible), where as the client does not need to know this information.

Another important note is that the admin is able to view every book request ever done by all clients. This is to allow better management on the admin's part, as well as the client usernames who requested the book, which allows the use of the client username attribute (As the username is not used anywhere else on the front end). However, for obvious privacy reasons, the client can only view their specific book requests.

The DDL for the request_book is defined as:

```
create table request_book(  
    request_number    serial,  
    username          varchar(20) not null,  
    request_isbn      varchar(13),  
    request_title     varchar(500),  
    date              date not null default current_date,  
    primary key (request_number),  
    foreign key (username) references client on delete cascade  
);
```

As seen above, each request_book contains the request number, as well as the username of the user who requested the book and date. The most important feature of the book request entity is the isbn and title. These have the option of one of them being null (i.e. isbn is null, title is not). This allows the client the option of either adding the book isbn, title or both in the request form, which is how the admin knows which book to add to the bookstore.

Once the admin has decided on the book request, a new insert is done on the admin_decides table, which is created using the DDL:

```
create table admin_decides(  
    request_number    serial,  
    email             varchar(40),  
    decision          bool not null,  
    date              date not null default current_date,  
    primary key (request_number, email),  
    foreign key (request_number) references request_book on delete cascade,  
    foreign key (email) references admin on delete cascade  
);
```

This DDL statement is relatively simple, and describes a new relation as containing the request_number, email, date and decision, which is the most important attribute, being true for approved, and false for rejected.

6.6. GUI Website

6.6.1. Overview

The Graphical User Interface of the project is arguably one of the most important aspects of the website. Due to it being an online bookstore, the front-end design is heavily important for the users. It also allows the user a much more pleasing method of navigating throughout the bookstore. The following sections listed below will delve more in depth about the overall GUI front end design.

6.6.2. CSS

For this particular project, the front end designed was developed using CSS, HTML and pug template engine files. The vast majority of the CSS was created by scratch, however some aspects (i.e. the navigation bar, and charts) were sourced from other websites (With sources linked) through the permission of Professor El-Roby.

6.6.3. Charts

The admin interface of the website implements a wide variety of charts, such as pie charts, bar charts and line charts. These are all very important for the admin user as it allows them access to view a more user-friendly design of the sales, therefore giving them a better understanding of how well the bookstore is doing in the market at the given time.

6.6.4. Graphics

The graphics of the bookstore website is very important as it allows the store to be better appealing. To also add, it lessens confusion for when the client is browsing books in the store, as they now have an image of the actual book they are viewing, which ironically makes the phrase *"Don't judge a book by its cover"* false in this case. It also allows each book to be distinguished from another if they share a common name (Which many books share). The book images themselves are gathered from online websites using the isbn as part of the book url, this makes the bookstore much more efficient, as now it does not need to store books locally or in the database, which containing 2699 book images would force the database to take up a large storage size.

The other graphical side involves the implementation of two relatively simple icons, the first being the bookstore *"Book-Sanctum"* logo, which is seen on the top left of every page (As well as the cover page of this report) and is used to identify the bookstore from other online stores, and the second being the "cart" symbol, used for the cart tab on the top right.

6.7. Warehouse

The warehouse of the bookstore is a very important feature that was added to the bookstore. The reason for this being described as a bonus is due to the fact that the entire bookstore design can work without the warehouse but adding one in allows for far greater scalability. The purpose of the warehouse is to store each book as well as the book stock. The importance of having a warehouse for this project is to control the design flow of an order and book purchase, as well as not implementing a stock attribute directly into book. It also allows for more realism in that real-life online stores use warehouses, and presumably have a warehouse schema for their own databases. Another important aspect of the warehouse is future scalability proofing. Although the project requirements specified only ordering books from a single warehouse, a warehouse is still required in the case where the bookstore expands to the point where multiple warehouses are needed, and therefore books can be shipped from multiple warehouses (Although for this implementation, all books are shipped from a single warehouse to satisfy the requirements specified).

The warehouse table itself is created through the following DDL statement:

```
create table warehouse(  
  id          serial,  
  address_id  serial not null,  
  primary key (id),  
  foreign key (address_id) references address_main(id)  
);
```

Where, as seen above, it only requires an id and an address (Using the address_id attribute). This keeps things relatively simple, yet still gets the main point across of what a warehouse should be. The warehouse itself is in a relation with the

book entity, forming an aggregation of warehouse_books (As described in section 1.8). This allows any book orders to use the actual warehouse_book table as a foreign key, which is described below:

```
create table order_book(
    isbn            varchar(13),
    order_number    serial,
    warehouse_id    serial not null,
    quantity        numeric(3,0) not null,
    primary key (order_number, isbn),
    foreign key (order_number) references orders,
    foreign key (warehouse_id, isbn) references warehouse_books
);
```

Where a single book ordered has an isbn, and an order_number as its primary identifying attribute keys. However, take note of warehouse_id. It is not a primary key. This is due to common logic. If warehouse_id was also a primary key, this can therefore imply that a single order can have the same book (specified by ISBN) ordered from multiple warehouses (Assuming the quantity is greater than 1), which is redundant and inefficient from a service online store point of view. Therefore, warehouse_id is not a primary key.

6.8. Transaction table

The bookstore allows the owner to add other various expenses to the bookstore such as rent, taxes, etc. whatever the owner decides is necessary to include. This feature was added through the (indirect) advice of Professor El-Roby on Culearn project discussion, where he advised it would be a better implementation to allow the owner to add other expenses besides publisher fees (Which was the default requirement) for future scalability. This also allows the bookstore to be more realistic in that a real store will always have many more expenses than simply “publisher fees”.

The transaction itself is not in any other relations due to no data sharing involved between transaction and any other relation being required and is therefore defined as a lone entity showcased in section 1 ER diagram. The transaction itself contains the date of the transaction, the amount, and the name of the transaction, as defined below in the DDL:

```
create table transaction(
    transaction_id    serial,
    name              varchar(30) not null,
    amount            numeric(6,2)not null,
    date              date not null default current_date,
    primary key (transaction_id)
);
```

The most important aspect about the transaction entity is the ability to take in both debit and credit amounts, which is defined as a negative numerical value for debit (i.e. sending money), and credit being a positive numerical value (i.e. receiving money), this allows greater scalability for the bookstore, in such cases when for example, the bookstore receives a donation, or other similar credit amounts. The transactions inputted here are also used to calculate sales, which will be described in section 6.10. below.

6.9. Restocking

One of the requirements for the project was allowing automatic book restocking if the book stock is less than a given amount. Although this in of itself is not a bonus, the restocking feature was expanded upon greatly. Due to the requirement being automatic restocking of books in the warehouse, this required no other entity or table needed,

however from a logical standpoint, the owner should be able to view all past restocks of books and which ones were restocked, and at what date. Therefore, a new entity was created, and defined below:

```
create table restock(
    restock_number    serial,
    warehouse_id      serial,
    isbn              varchar(13),
    quantity          numeric(3,0) not null,
    date              date not null default current_date,
    primary key (restock_number),
    foreign key (warehouse_id, isbn) references warehouse_books
);
```

In this case, whenever the book quantity less than the threshold given (In this implementation is <5), the postgres database will insert into this table from within the update_stock() trigger function (Called after each order_book insert). The quantity attribute is the most important field for restock, as it contains the exact number or quantity of books that must be restocked using the number sold in the previous month. The other bonus feature of restocking a book is that logically, if a book were to be restocked, there would be a fees for the order, whether this is the price the bookstore must pay to order books directly from the manufacturer, or a simple restock fees. In this case, it is the first. The actual price of the restock is simply the quantity multiplied by the book price multiplied by the publisher fees. This, however, is not stored in restock as it would be redundant (These attributes can be easily retrieved from the book table).

Although the restocking bonus does not mean much compared to the other bonuses implemented above, it is used heavily in the next section (6.10) for transactions and sales, which will be described.

6.10. Transactions View

The transactions view is the most important aspect of the postgres database for the owner. Its purpose is to give the owner a full examination of all finances for the bookstore. The transactions view itself stores transactions from multiple sources, being the following: book order sales, book order publisher fees, other transactions (From section 6.8.), and lastly restock orders (From section 6.9). The reason for why a view is needed is due to the other bonus feature of this section, being the ability to view all transactions, aside from simply book sales and publisher fees. Due to now requiring the bookstore server to call queries to receive transactions data from four different tables, a view was implemented which uses unions between the four tables to group together data into a single, unified table which can be queried upon just once.

The actual view query which creates the view transactions can be defined as the following:

```
(
select to_char(date, 'YYYY-MM-DD') as date, name, amount, 'other' as type from transaction
union
select  to_char(order_date, 'YYYY-MM-DD') as date, book.isbn as name,
round(price*quantity::numeric, 2) as amount, 'book sale' as type from orders
inner join order_book on orders.order_number = order_book.order_number
inner join book on book.isbn = order_book.isbn
union
select  to_char(order_date, 'YYYY-MM-DD') as date, publisher.name as name, round(-
price*quantity*sale_percent::numeric, 2) as amount, 'publisher fees' as type from orders
```

```

inner join order_book on orders.order_number = order_book.order_number
inner join book on book.isbn = order_book.isbn
inner join publisher on book.publisher_id = publisher.id
union
select to_char(date, 'YYYY-MM-DD') as date, concat(publisher.name, ': ', book.isbn, '
x', quantity) as name, round(-price*sale_percent*quantity::numeric,2) as amount, 'restock'
as type
from restock
inner join book on restock.isbn = book.isbn
inner join publisher on publisher.id = book.publisher_id
order by date desc
);

```

As seen above, the view creation is complex, but it is much better than simply querying the data four times. What this allows, is querying a table to look like the following result:

date	name	amount	type
2020-04-07	Scholastic	-3.21	Publisher fees
2020-04-07	9780553573428	12.99	Book sale
2020-03-07	Scholastic: 9780553573428 x 14	-45.32	Restock
2020-03-29	Website fees	-14.99	Other

As seen above, the return result displays each type of transaction (defined by attribute type) nicely and organized to be used later. This view containing all the various types of sales is then used in the sales page and the transactions page (Described below in section 6.11).

6.11. Transactions Page

The transaction page was created as a way to allow the owner to manage all his transactions that the bookstore has completed. This was a necessary addition as the design of the sales reports and page did not do enough justice to allow the owner the option to view each individual book sale and the other transactions listed in the above sections. This also allows the owner to know exactly where all the finances of the bookstore are sent or received. The design of this page was based upon actual online banking pages that show the user's transactions. This page also only uses queries which select data from the transactions view described in section 6.10. above.

6.12. Filter sales/transactions between dates

The sales and transaction pages both allow the owner to filter between two different dates to view the bookstore sales and various transactions. This felt like a very important bonus feature to add to these pages due to simply viewing all sales as not being enough of an "examination" into the bookstore itself. But now with this new addition of a filter feature, it allows the owner the ability to view sales and transactions from set dates, better showcasing the bookstore transactions and sales. The actual filter component is done by the client-side JavaScript sending the start and end date values to the server to then query back on the specified query functions for the pages, which then implements the following line to filter between two dates:

where date >=\$1 and date<=\$2

where the input parameter \$1 is the start date and the input parameter \$2 is the end date.

7. Github Repository

<https://github.com/SharjeelAliBCS/comp3005W20-project>

The code is located in /code/book_sancutm

The SQL queries are located in/SQL

7. Website link

The website is hosted on Heroku as well, feel free to create an account (Just do not put personal information as the database is not secure). Also adjust the browser zoom sizes, as the CSS is not mobile friendly, and should only be viewed on a desktop:

[**http://book-sanctum.herokuapp.com/**](http://book-sanctum.herokuapp.com/)