ISYS2099

Database Applications

Name: Database Project

Lecturer: Tri Dang

Student name: Nguyen Cong Phuong **ID number**: s3804846

Student name: Nguyen Tuan Anh **ID number**: s3864077

Table of Contents

Database Design

Cart_item table:

- “**id**” (CHAR(36) NOT NULL AUTO_INCREMENT PRIMARY KEY)
- “**ItemDetail**” (JSON NOT NULL)
- “**quantity**” (BIGINT NOT NULL)
- “**productId**” (CHAR(36) NOT NULL, FOREIGN KEY referencing to “product”)
- “**createdAt**”(DATETIME)
- “**updatedAt**”(DATETIME)
- “**cartId**" (INT NOT NULL, FOREIGN KEY referencing to “**cart**(**id**)”)

Carts table:

- “**id**” (INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY)
- “**createAt**” (DATETIME)
- “**updatedAt**”(DATETIME)
- “**userId**” (CHAR(36) NOT NULL, FOREIGN KEY referencing to “**users(id)**”)

Users table:

- “**id**” (CHAR(36) NOT NULL AUTO_INCREMENT PRIMARY KEY)
- “**username**” (VARCHAR(255) NOT NULL)
- “**email**” (VARCHAR(255) NOT NULL)
- “**passwordHash**” (VARCHAR(255) NOT NULL)
- “**createAt**” (DATETIME)
- “**updatedAt**”(DATETIME)

Order_items table:

- “**id**” (CHAR(36) NOT NULL AUTO_INCREMENT PRIMARY KEY)
- “**productId**” (CHAR(36) NOT NULL)
- “**itemDetail** (CHAR(36) NOT NULL)
- “**quantity**” (BIGINT NOT NULL)
- “**createAt**” (DATETIME)
- “**updatedAt**”(DATETIME)
- “**orderId**" (CHAR(36) NOT NULL, FOREIGN KEY referencing to “**orders(id)**”)

Orders table:

- “**id**” (CHAR(36) NOT NULL AUTO_INCREMENT PRIMARY KEY)
- “**stat**” (ENUM(‘pending’,’accept’,’reject’) NOT NULL)
- “**createAt**” (DATETIME)
- “**updatedAt**”(DATETIME)
- “**userId**”(CHAR(36),FOREIGN KEY referencing to **‘users(id)’**)

Products table:

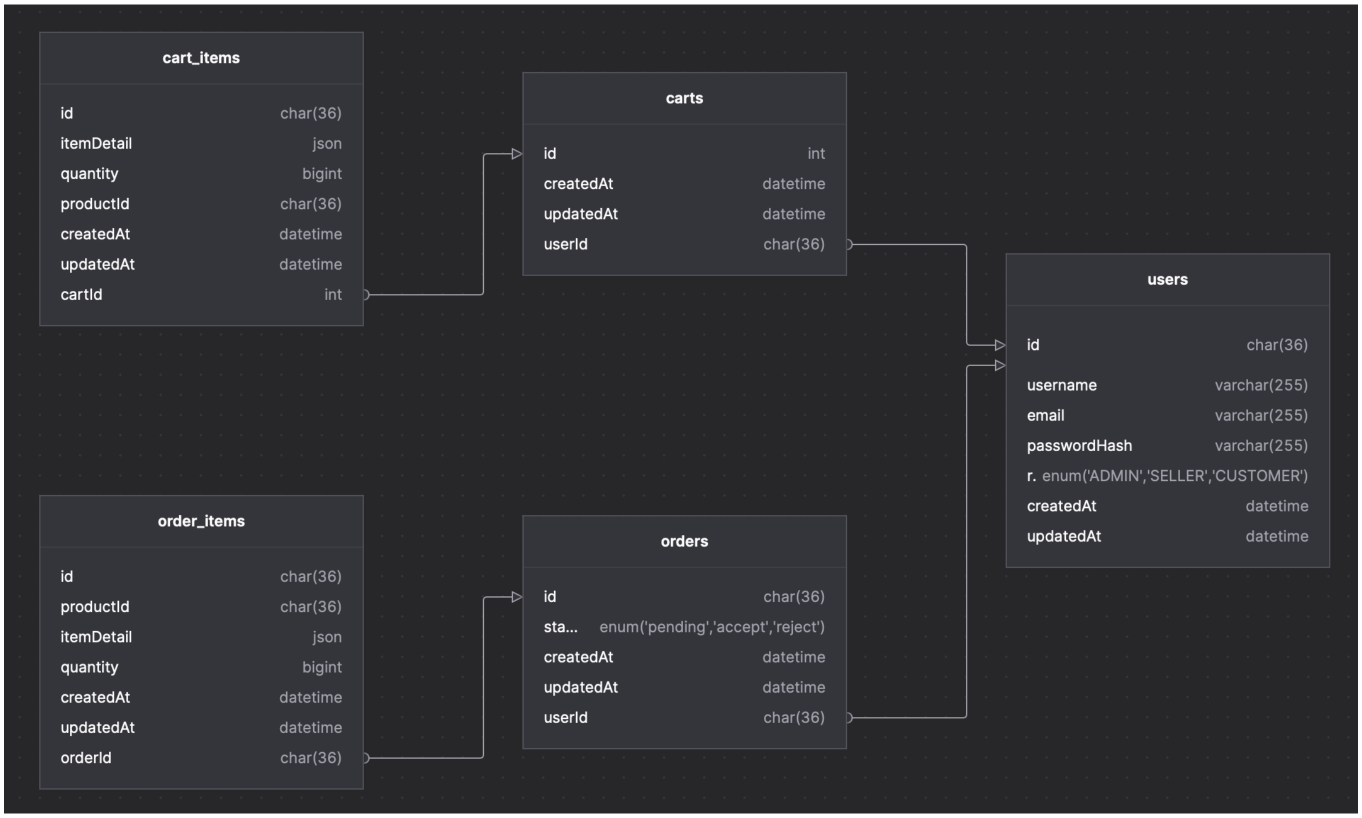
- “**id**” (CHAR(36) NOT NULL AUTO_INCREMENT PRIMARY KEY)
- “**title**” (VARCHAR(255) NOT NULL)
- “**description**” (VARCHAR(255) NOT NULL)
- “**price**” (FLOAT NOT NULL)
- “**quantityNumber**”(BIGINT NOT NULL)
- “**dimension**”(JSON NOT NULL)
- “**Image**" (VARCHAR(255) NOT NULL)
- “**addtional_attributes**” (JSON NOT NULL)
- “**categoryId**” (VARCHAR(255) NOT NULL,)
- “**createdAt**”(DATETIME)
- “**updatedAt**”(DATETIME)
- “**warehouseId**" (CHAR(36) NOT NULL, FOREIGN KEY referencing to “**warehouse**(**id**)”)

Warehouses table:

- “**id**” (CHAR(36) NOT NULL AUTO_INCREMENT PRIMARY KEY)
- “**name**” (VARCHAR(255) NOT NULL)
- “**address**” (JSON NOT NULL)
- “**volume**” (BIGINT NOT NULL)
- “**createdAt**”(DATETIME)
- “**updatedAt**”(DATETIME)

Categories table:

- “**id**” (VARCHAR(255) NOT NULL AUTO_INCREMENT PRIMARY KEY)
- “**name**” (VARCHAR(255) NOT NULL)
- “**parentId**” (VARCHAR(255) NOT NULL)



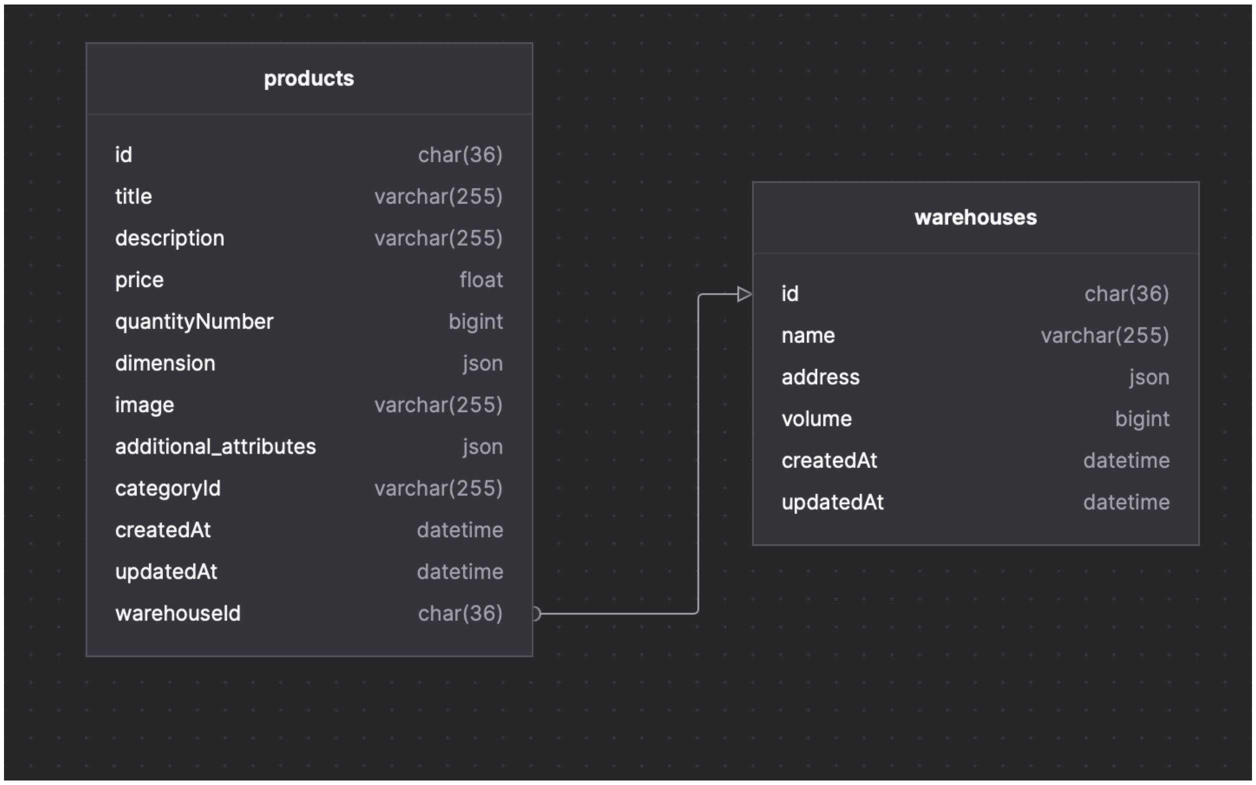
cart_items	
id	char(36)
itemDetail	json
quantity	bigint
productId	char(36)
createdAt	datetime
updatedAt	datetime
cartId	int

carts	
id	int
createdAt	datetime
updatedAt	datetime
userId	char(36)

users	
id	char(36)
username	varchar(255)
email	varchar(255)
passwordHash	varchar(255)
r.	enum('ADMIN','SELLER','CUSTOMER')
createdAt	datetime
updatedAt	datetime

order_items	
id	char(36)
productId	char(36)
itemDetail	json
quantity	bigint
createdAt	datetime
updatedAt	datetime
orderId	char(36)

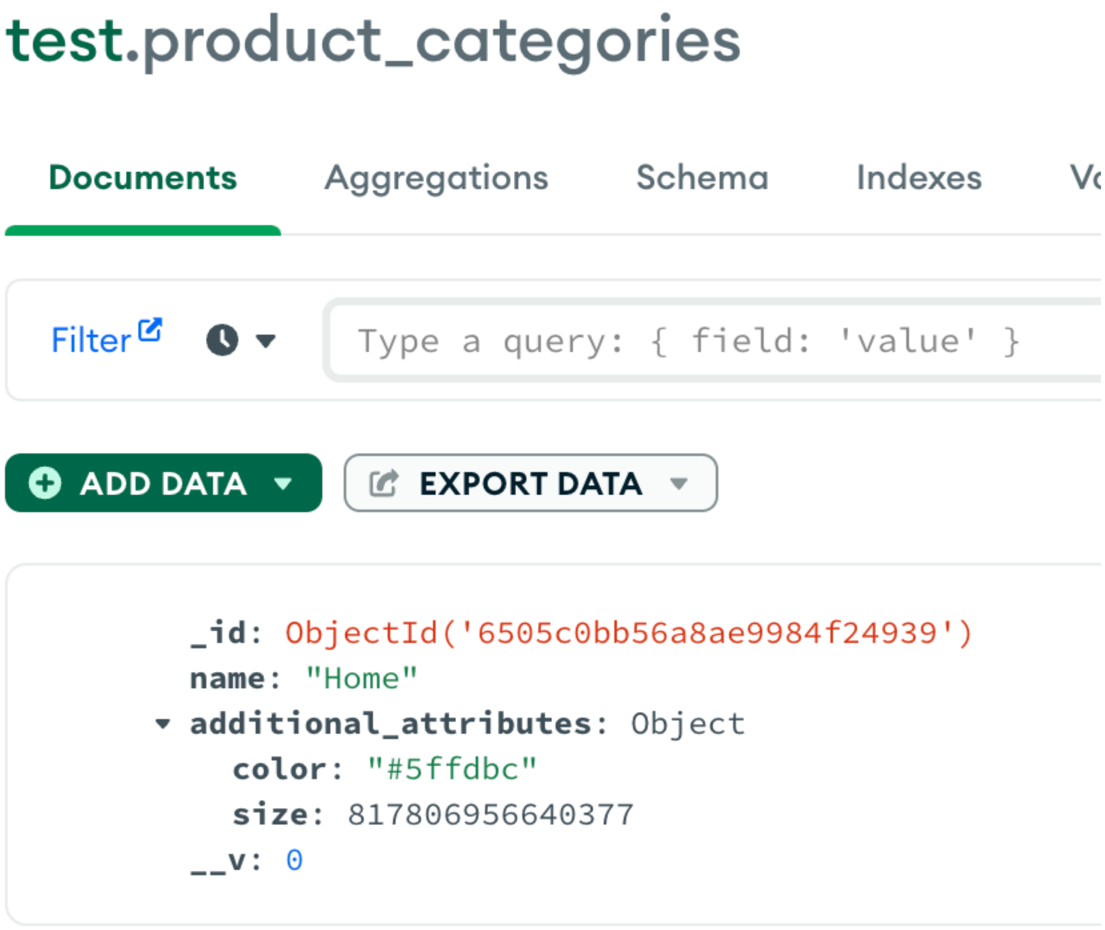
orders	
id	char(36)
sta...	enum('pending','accept','reject')
createdAt	datetime
updatedAt	datetime
userId	char(36)



products	
id	char(36)
title	varchar(255)
description	varchar(255)
price	float
quantityNumber	bigint
dimension	json
image	varchar(255)
additional_attributes	json
categoryId	varchar(255)
createdAt	datetime
updatedAt	datetime
warehouseId	char(36)

warehouses	
id	char(36)
name	varchar(255)
address	json
volume	bigint
createdAt	datetime
updatedAt	datetime

Figure 1Figure 2 MySQL Schema

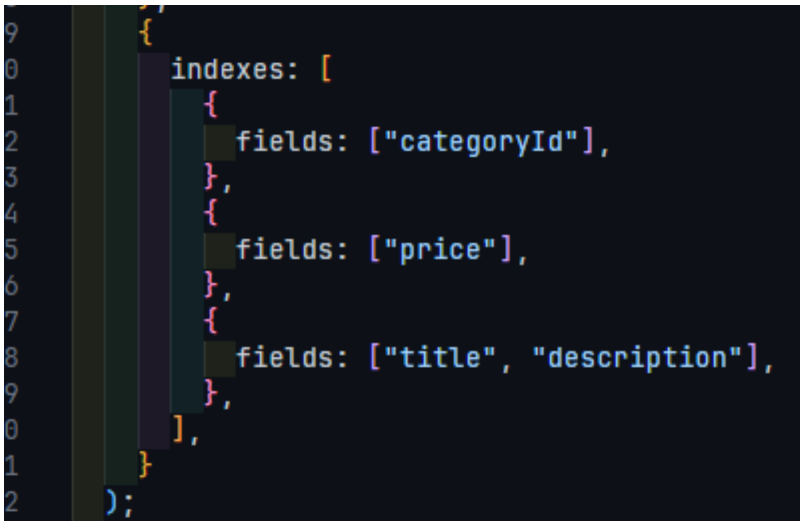


```
_id: ObjectId('6505c0bb56a8ae9984f24939')
name: "Home"
additional_attributes: Object
  color: "#5ffdbc"
  size: 817806956640377
__v: 0
```

Figure 3 MongoDB Schema

1. Database Design

Performance Analysis.



```
  {
    indexes: [
      {
        fields: ["categoryId"],
      },
      {
        fields: ["price"],
      },
      {
        fields: ["title", "description"],
      },
    ],
  }
);
```

Figure 4 product indexing

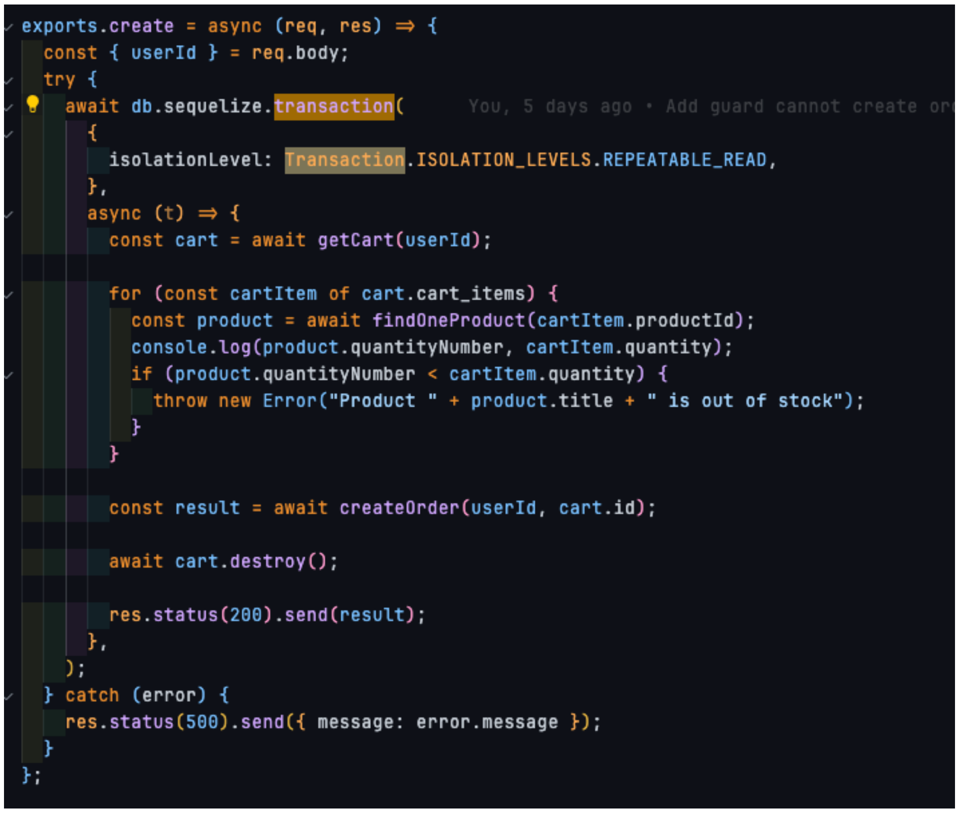
- Title and description pair partial index

For searching we always look up both title description so indexing both fields will help searching more faster

- Indexing categoryId and price for filter and sorting performance improvements

2. Data Consistency.

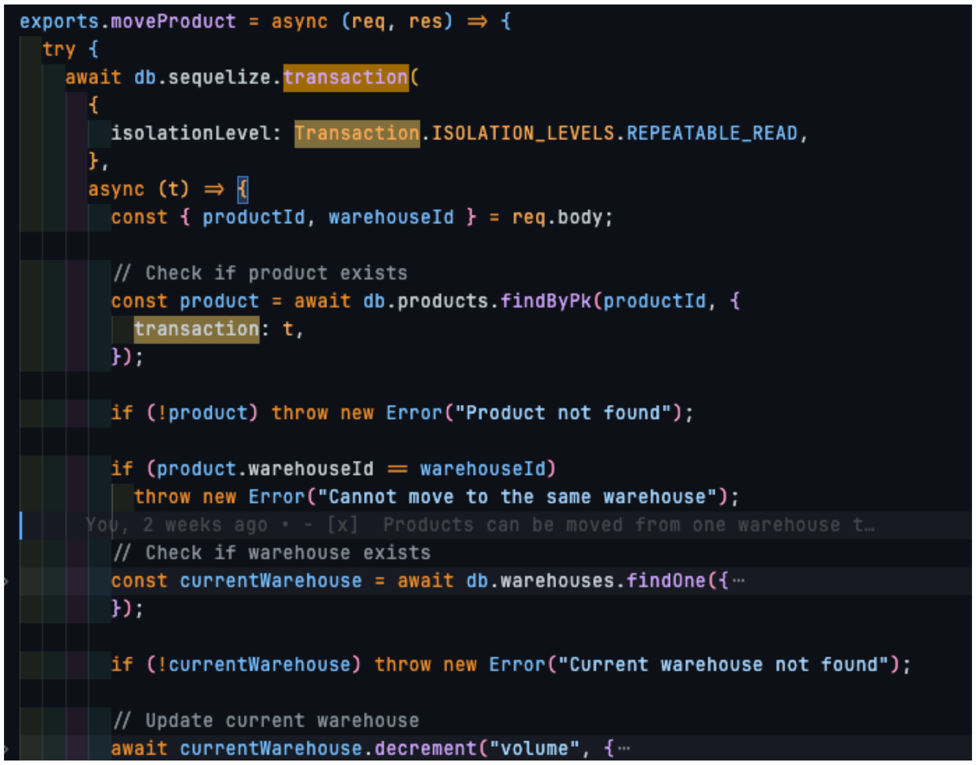
Transactions and hooks (trigger)



```
exports.create = async (req, res) => {
  const { userId } = req.body;
  try {
    await db.sequelize.transaction(      You, 5 days ago • Add guard cannot create or
      {
        isolationLevel: Transaction.ISOLATION_LEVELS.REPEATABLE_READ,
      },
      async (t) => {
        const cart = await getCart(userId);

        for (const cartItem of cart.cart_items) {
          const product = await findOneProduct(cartItem.productId);
          console.log(product.quantityNumber, cartItem.quantity);
          if (product.quantityNumber < cartItem.quantity) {
            throw new Error("Product " + product.title + " is out of stock");
          }
        }

        const result = await createOrder(userId, cart.id);

        await cart.destroy();

        res.status(200).send(result);
      },
    );
  } catch (error) {
    res.status(500).send({ message: error.message });
  }
};
```

Figure 5 Create Order transaction

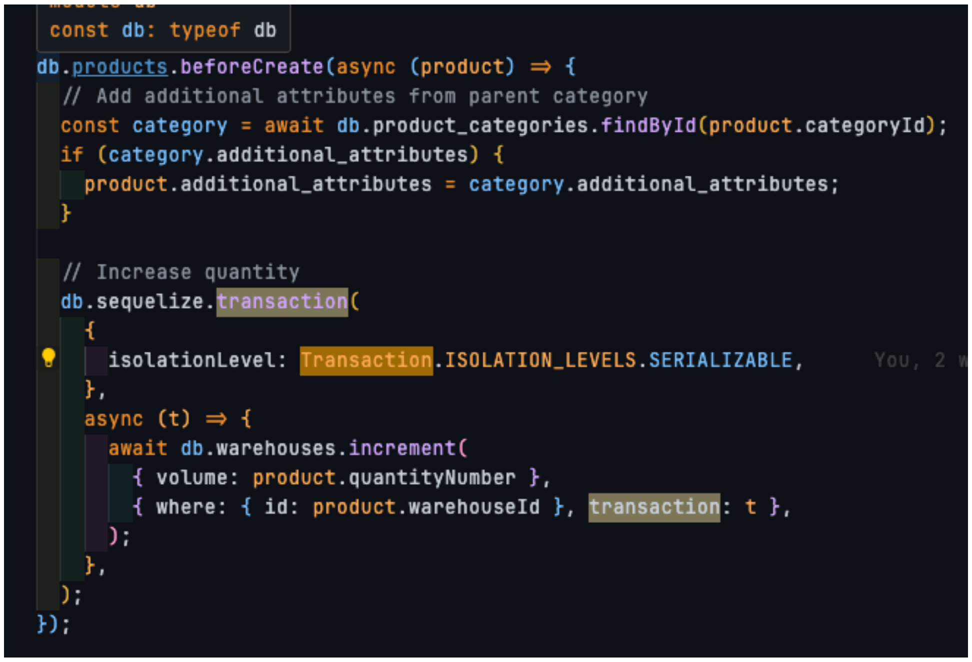
Transactions are used here to ensure data integrity and consistency, especially in a scenario involving multiple database operations. In this case, multiple operations are being performed, including checking cart items, creating an order, and potentially deleting the user's cart. Using a transaction ensures that either all of these operations are completed successfully or none of them are. This prevents situations where, for example, an order is created but the user's cart is not properly deleted.



```
exports.moveProduct = async (req, res) => {
  try {
    await db.sequelize.transaction(
      {
        isolationLevel: Transaction.ISOLATION_LEVELS.REPEATABLE_READ,
      },
      async (t) => {
        const { productId, warehouseId } = req.body;

        // Check if product exists
        const product = await db.products.findByPk(productId, {
          transaction: t,
        });

        if (!product) throw new Error("Product not found");

        if (product.warehouseId == warehouseId)
          throw new Error("Cannot move to the same warehouse");
    You, 2 weeks ago • - [x] Products can be moved from one warehouse t…
        // Check if warehouse exists
        const currentWarehouse = await db.warehouses.findOne({…
        });

        if (!currentWarehouse) throw new Error("Current warehouse not found");

        // Update current warehouse
        await currentWarehouse.decrement("volume", {…
```

Figure 6 Move product transaction.

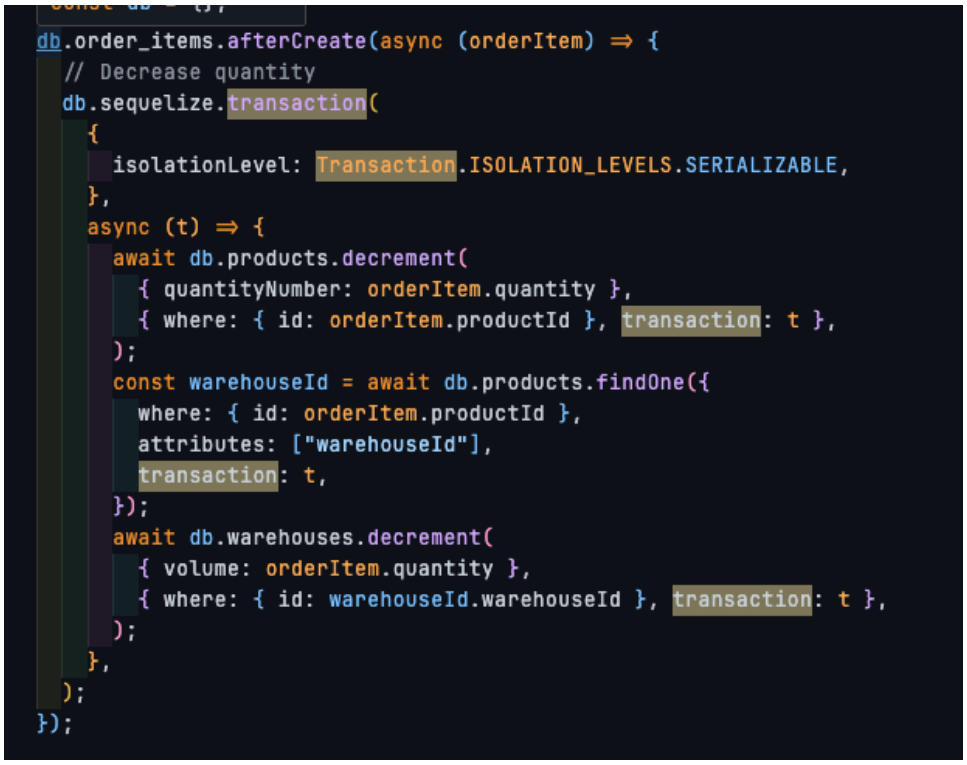
This transaction ensures prevention of race conditions when moving a product. In the event of two simultaneous product movements, one will succeed while the other will fail due to the product already being in motion.



```
  const db: typeof db
db.products.beforeCreate(async (product) => {
  // Add additional attributes from parent category
  const category = await db.product_categories.findById(product.categoryId);
  if (category.additional_attributes) {
    product.additional_attributes = category.additional_attributes;
  }

  // Increase quantity
  db.sequelize.transaction(
    {
      isolationLevel: Transaction.ISOLATION_LEVELS.SERIALIZABLE,      You, 2 w
    },
    async (t) => {
      await db.warehouses.increment(
        { volume: product.quantityNumber },
        { where: { id: product.warehouseId }, transaction: t },
      );
    },
  );
});
```

Figure 7 create product trigger.

This trigger activates upon the creation of a new product. It extends the product's attributes by incorporating all additional characteristics associated with its category. Furthermore, it augments the volume of the warehouse.



```
db.order_items.afterCreate(async (orderItem) => {
  // Decrease quantity
  db.sequelize.transaction(
    {
      isolationLevel: Transaction.ISOLATION_LEVELS.SERIALIZABLE,
    },
    async (t) => {
      await db.products.decrement(
        { quantityNumber: orderItem.quantity },
        { where: { id: orderItem.productId }, transaction: t },
      );
      const warehouseId = await db.products.findOne({
        where: { id: orderItem.productId },
        attributes: ["warehouseId"],
        transaction: t,
      });
      await db.warehouses.decrement(
        { volume: orderItem.quantity },
        { where: { id: warehouseId.warehouseId }, transaction: t },
      );
    },
  );
});
```

Figure 8 create order item trigger

This trigger is activated upon the creation of an order item. It simultaneously augments the volume in the warehouse and adjusts the quantity number of the respective product. In the event of a race condition, the correct order will be prioritized to maintain consistency in the quantity of items.

3. Data Security.

Authorization and Authentication: This project employs the JWT (JSON Web Token) authentication method to secure user access

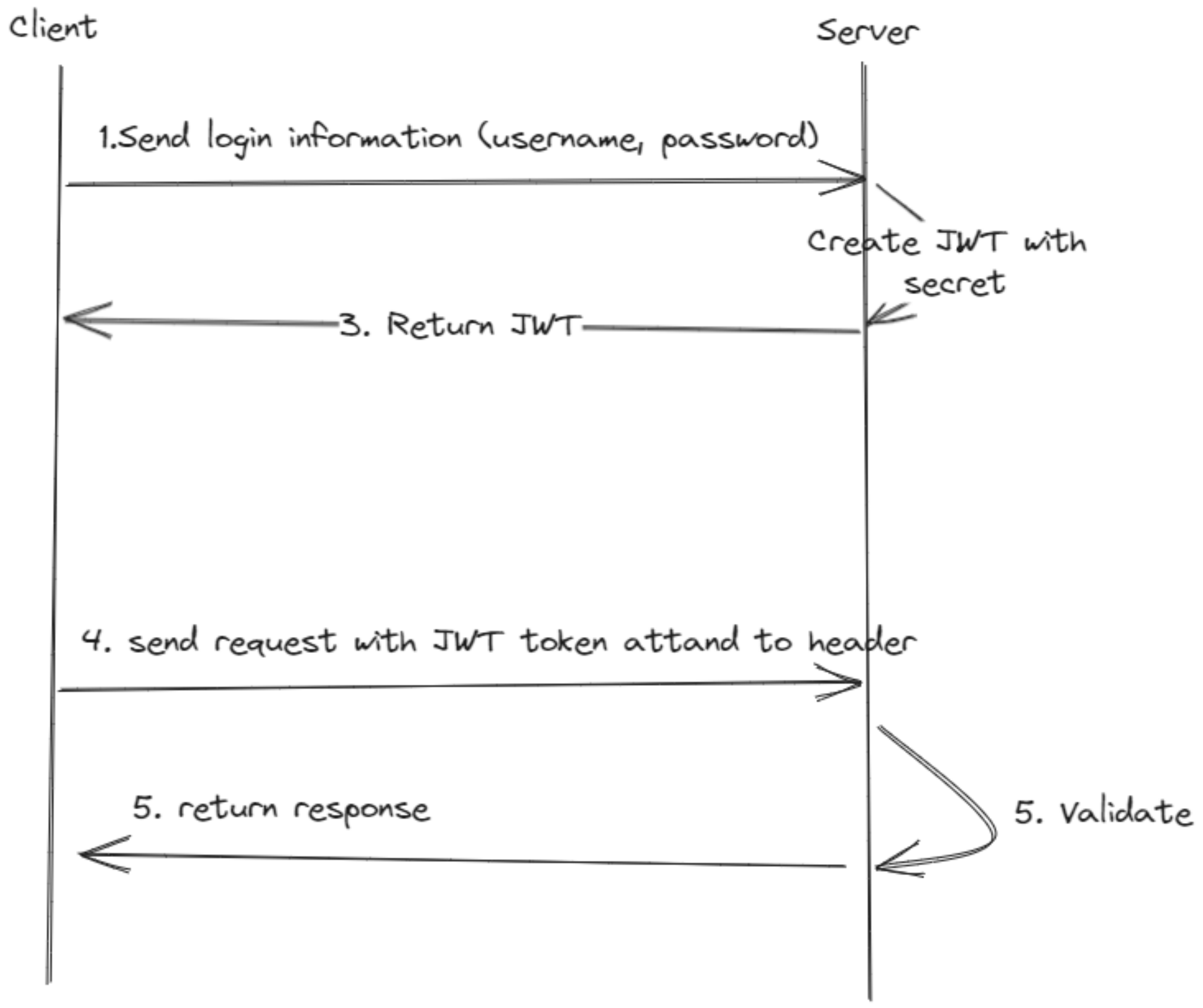

Client
Server
1.Send login information (username, password)
Create JWT with secret
3. Return JWT
4. send request with JWT token attand to header
5. return response
5. Validate

Figure 9 Login Flow

Login Flow: Begin by sending the login information to the server in order to obtain a JWT (JSON Web Token). Subsequently, append this token to the 'x-access-token' header and initiate a request to the server for validation.

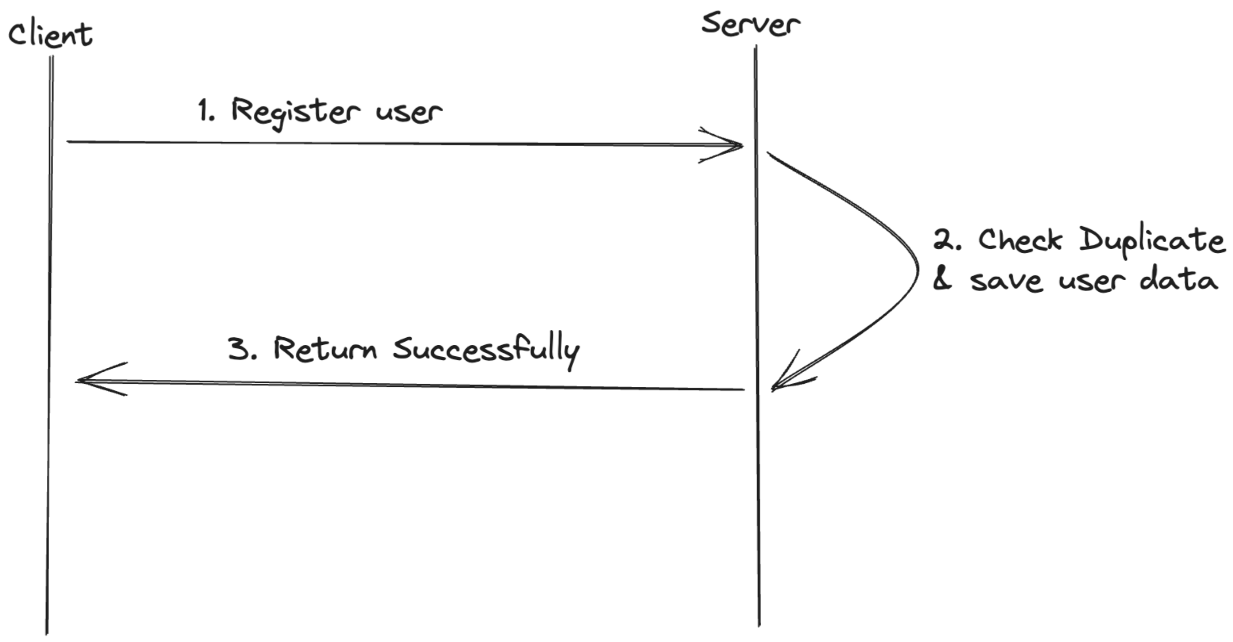


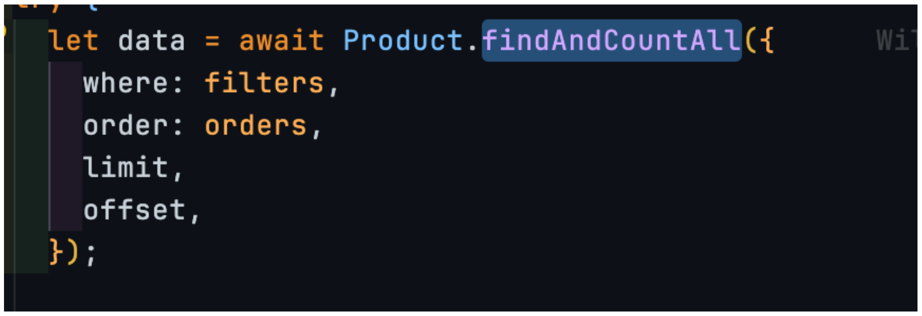
Figure 10 Register Flow

Register Flow: The client initiates a signUp request to the server, which then processes and saves the information to the database. Upon successful registration, the server sends a confirmation of the registration back to the client.

Role Verification: When the server receives a JWT token from the user, it inspects the token to retrieve the associated userId. Using this ID, the server locates the user and extracts their assigned role.

Error Handling: In every scenario, there are predefined protocols in place to address potential errors and exceptions. These measures ensure smooth operation and a secure user experience.

SQL Injection handling: by using Sequelize ORM it has been handle when using findAndCountAll with Parameterized Queries [1]



```
let data = await Product.findAndCountAll({
  where: filters,
  order: orders,
  limit,
  offset,
});
```

Figure 11 findAndCountAll method ORM

SQL Injection Handling: Leveraging the Sequelize ORM, we've implemented robust measures to mitigate SQL injection risks. Specifically, we utilize the findAndCountAll method along with Parameterized Queries. By strictly avoiding raw SQL queries, we automatically escape and sanitize user inputs. Additionally, we offer tools for defining data types and conducting validation to fortify our security measures.

4. Conclusion

In conclusion, this report outlines a robust e-commerce platform that bridges sellers, warehouse admins, and customers, ensuring seamless operations from product listing to order delivery. The frontend offers a well-defined interface for users at every level, from registration to product management, order placement, and even simulating order deliveries. It notably places a strong emphasis on user experience with functionalities like filtering, searching, and sorting.

On the backend, the focus is twofold: ensuring data integrity and optimizing performance. The use of transactions, concurrency techniques, triggers, and stored procedures guarantees that the data remains consistent and accurate across operations, especially in scenarios with simultaneous requests or inventory updates. The platform also encourages best practices in database security by creating designated roles with relevant privileges. With an architecture that thoughtfully integrates NoSQL where needed, the system is poised to deliver both flexibility and performance. The layered approach ensures that as the platform scales, it remains responsive, secure, and efficient.

5. Reference:

[1] GitHub, https://github.com/sequelize/sequelize/blob/6b58009d6b539897ae567946fa06dce5dd01e6ea/packages/core/src/utils/sql.ts#L209C7-L209C7 (accessed Sep. 18, 2023).