01219231 Database Systems for Software and Knowledge Engineers

2024/1st

member : Jirakorn, Phubet, Adithep

The supermarket system

This supermarket database system is aimed for manager and human resource in supermarkets to use. The web-app helps these people by letting them create a table update and delete freely. Also, we have 9 queries for them to conveniently see the data without them looking in the read feature and writing down. The data that are stored here are the customer's profile, supplier detail, product, employee, promotion and bill to help with finance management.

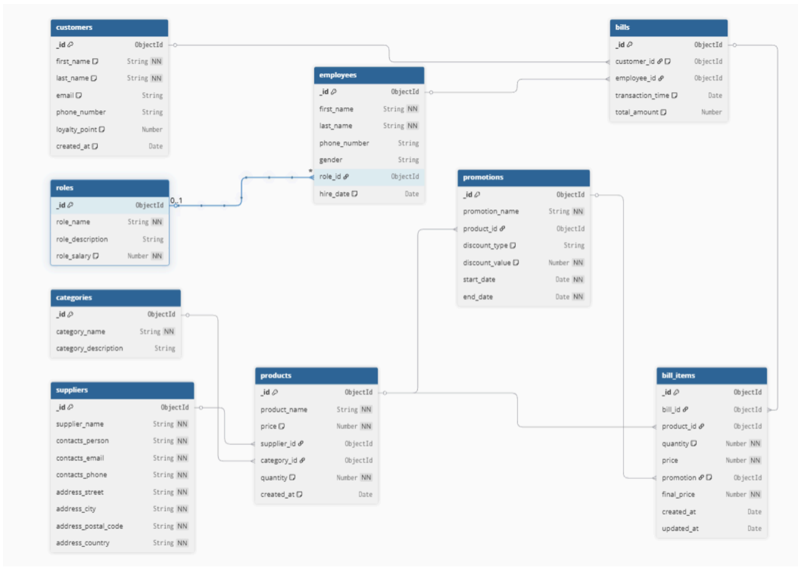
1.) Business selection and analysis

1. A supermarket manager can register new products and suppliers into the system to build the inventory catalog (CRUD).
2. Store the customer’s membership details, employee profiles, and current stock levels for operational management.
3. Tracking product quantity and low-stock status to ensure that inventory is replenished immediately before running out.
4. Track of the active promotions and discount rules so we can apply them to the appropriate products during a sale, make sure the calculation is accurate. Additionally, calculate the total bill amount based on the customer’s purchase and their loyalty status (for example, members will earn points for every purchase, and specific items will trigger automatic price deductions based on valid promotion dates).
5. Assign employees into specific roles (Cashier, Manager, Stock) to handle different store operations and manage permissions effectively.
6. Track the sales trends and revenue on specific dates to forecast future inventory needs using historical transaction data.
7.

● Necessity of Data Collection, Management, and Utilization in Activities and Processes

- The business can coordinate daily sales, manage inventory levels, and track employee performance effectively.
- To collect information about customers, products, and sales transactions to provide accurate service and ensure that no popular item is out of stock.
- To gather and analyze past data, such as the list of best-selling products and monthly revenue, so that we can prepare the right amount of stock for future demand.

2. Database design



1. Relation: Customer

Initial Design: `Customer(_id, first_name, last_name, email, phone_number, loyalty_point, created_at)`

- **1NF (First Normal Form):** The table is already in 1NF because all values are atomic. There are no multi-valued attributes or repeating groups (e.g., multiple phone numbers are not stored in a single cell).
- **2NF (Second Normal Form):** The primary key is `_id`, which is a single attribute. Since the key is not composite, there are no partial dependencies. Therefore, the table is in 2NF.
- **3NF (Third Normal Form):** There are no transitive dependencies; all non-key attributes (such as `first_name`, `email`, `loyalty_point`) depend solely on the primary key `_id`. The table is in 3NF.

---

2. Relation: Employee

Initial Design: `Employee(_id, first_name, last_name, phone_number, gender, role_id, hire_date)`

- **1NF (First Normal Form):** The table is in 1NF as all attributes hold atomic values.
- **2NF (Second Normal Form):** The primary key is `_id` (a single attribute), so there are no partial dependencies. Thus, the table is already in 2NF.
- **3NF (Third Normal Form):** There are no transitive dependencies. All non-key attributes depend solely on the primary key `_id`. While `role_id` is a foreign key, the specific assignment of a role depends on the employee `_id`, not another non-key attribute. The table is in 3NF.

---

3. Relation: Role

Initial Design: `Role(_id, role_name, role_description, role_salary)`

- **1NF (First Normal Form):** The table is in 1NF because all attributes contain atomic values.
- **2NF (Second Normal Form):** The primary key is `_id`, a single attribute. There are no partial dependencies, so the table is in 2NF.
- **3NF (Third Normal Form):** All non-key attributes (`role_name`, `role_description`, `role_salary`) depend directly on the primary key `_id`. There are no transitive dependencies, so the table is in 3NF.

---

4. Relation: Product

Initial Design: `Product(_id, product_name, price, supplier_id, category_id, quantity, created_at)`

- **1NF (First Normal Form):** The table is in 1NF because all attributes are atomic.
- **2NF (Second Normal Form):** The primary key is `_id`, which is a single attribute. No partial dependencies exist, so the table is in 2NF.
- **3NF (Third Normal Form):** There are no transitive dependencies; each non-key attribute (`product_name`, `price`, `quantity`) is dependent only on the primary key `_id`. Foreign keys (`supplier_id`, `category_id`) reference other entities but do not cause transitive dependency within this table. The table is in 3NF.

---

5. Relation: Category

Initial Design: `Category(_id, category_name, category_description)`

- **1NF (First Normal Form):** The table is in 1NF because all attributes contain atomic values.
- **2NF (Second Normal Form):** The primary key is `_id`, which is a single attribute. No partial dependencies exist, so the table is in 2NF.
- **3NF (Third Normal Form):** All non-key attributes (`category_name`, `category_description`) depend directly on the primary key `_id`. There are no transitive dependencies, so the table is in 3NF.

---

6. Relation: Supplier

Initial Design: `Supplier(_id, supplier_name, contacts_person, contacts_email, contacts_phone, address_street, address_city, address_postal, address_country)`

- **1NF (First Normal Form):** The table is in 1NF because all attributes are atomic. The nested objects for contacts and addresses have been flattened into distinct columns (e.g., `contacts_email`, `address_city`).
- **2NF (Second Normal Form):** The primary key is `_id`, a single attribute, so there are no partial dependencies. The table is in 2NF.
- **3NF (Third Normal Form):** All non-key attributes are fully dependent on the primary key `_id`, with no transitive dependencies. Therefore, the table is in 3NF.

---

7. Relation: Promotion

Initial Design: `Promotion(_id, promotion_name, product_id, discount_type, discount_value, start_date, end_date)`

- **1NF (First Normal Form):** The table is in 1NF as all attributes are atomic.
- **2NF (Second Normal Form):** The primary key is `_id`, which is a single attribute. There are no partial dependencies, so the table is in 2NF.
- **3NF (Third Normal Form):** All attributes depend directly on `_id`. The promotion details describe the specific promotion instance. While it references a product, the rules of the discount depend on the promotion ID. The table is in 3NF.

---

8. Relation: Bill

Initial Design: `Bill(_id, customer_id, employee_id, transaction_time, total_amount)`

- **1NF (First Normal Form):** The table is in 1NF because all attributes are atomic. The list of products purchased has been moved to the `BillItem` table to avoid repeating groups.
- **2NF (Second Normal Form):** The primary key is `_id`, a single attribute. Since there are no partial dependencies, the table is in 2NF.
- **3NF (Third Normal Form):** There are no transitive dependencies. All attributes (`customer_id`, `employee_id`, `transaction_time`, `total_amount`) depend solely on the primary key `_id`. Therefore, the table is in 3NF.

---

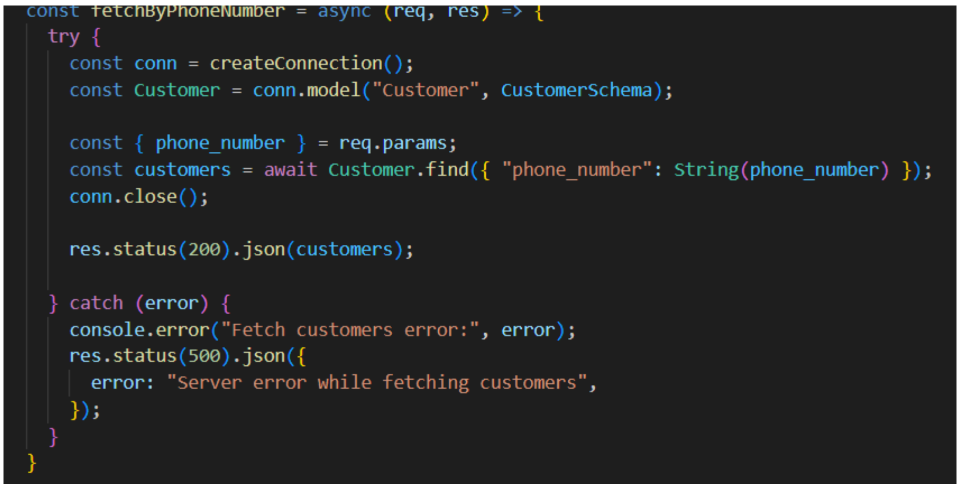
9. Relation: BillItem

Initial Design: `BillItem(_id, bill_id, product_id, quantity, price, promotion_id, final_price)`

- **1NF (First Normal Form):** The table is in 1NF because all attributes contain atomic values.
- **2NF (Second Normal Form):** The primary key is `_id` (a unique identifier for the line item). Since the key is not composite, there are no partial dependencies. The table is in 2NF.
- **3NF (Third Normal Form):** All non-key attributes (`quantity`, `price`, `final_price`) depend directly on the primary key `_id`. `Price` here represents the historical price at the moment of sale, ensuring it doesn't transitively depend on the `Product` table (which might change later). Therefore, the table is in 3NF.

3.) 9 queries

1. Customer Lookup by phone number



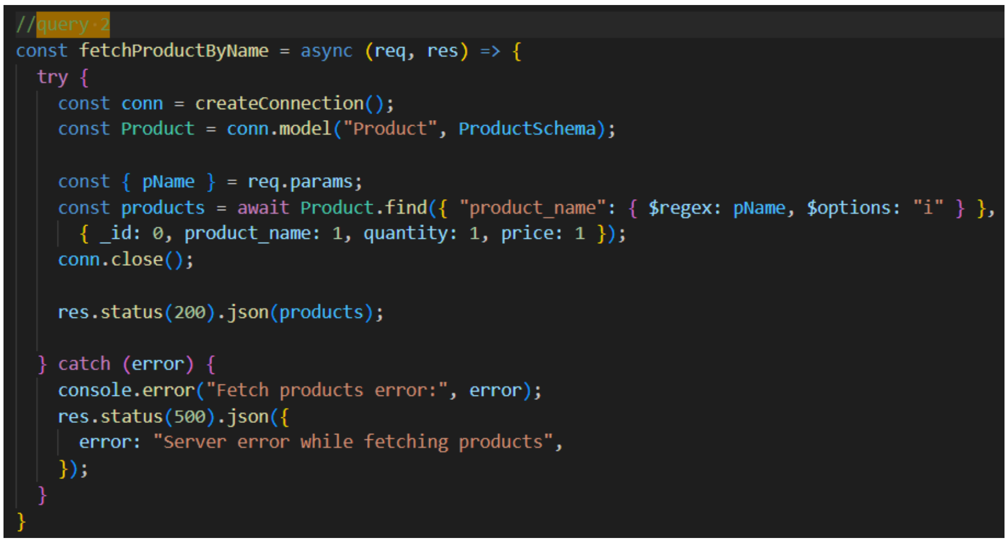
```
const fetchByPhoneNumber = async (req, res) => {
  try {
    const conn = createConnection();
    const Customer = conn.model("Customer", CustomerSchema);

    const { phone_number } = req.params;
    const customers = await Customer.find({ "phone_number": String(phone_number) });
    conn.close();

    res.status(200).json(customers);

  } catch (error) {
    console.error("Fetch customers error:", error);
    res.status(500).json({
      error: "Server error while fetching customers",
    });
  }
}
```

2 . Product Price Check



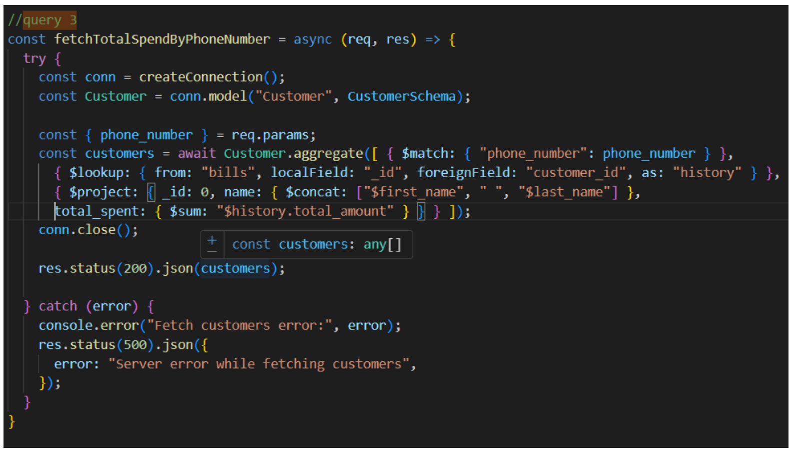
```
//query 2
const fetchProductByName = async (req, res) => {
  try {
    const conn = createConnection();
    const Product = conn.model("Product", ProductSchema);

    const { pName } = req.params;
    const products = await Product.find({ "product_name": { $regex: pName, $options: "i" } },
      { _id: 0, product_name: 1, quantity: 1, price: 1 });
    conn.close();

    res.status(200).json(products);

  } catch (error) {
    console.error("Fetch products error:", error);
    res.status(500).json({
      error: "Server error while fetching products",
    });
  }
}
```

3. Total spend by customer



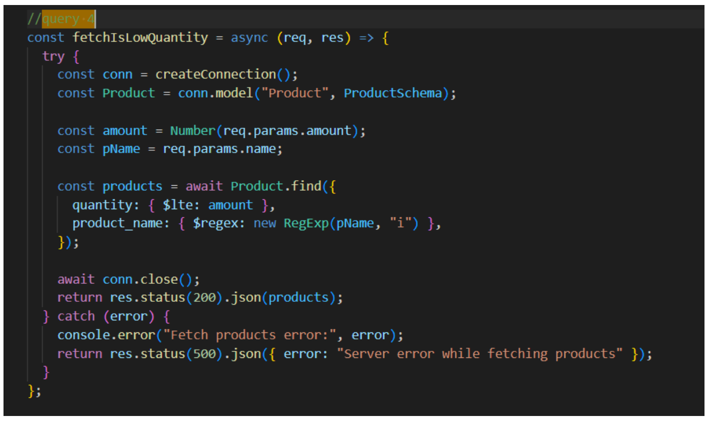
```javascript
//query 3
const fetchTotalSpendByPhoneNumber = async (req, res) => {
  try {
    const conn = createConnection();
    const Customer = conn.model("Customer", CustomerSchema);

    const { phone_number } = req.params;
    const customers = await Customer.aggregate([ { $match: { "phone_number": phone_number } },
      { $lookup: { from: "bills", localField: "_id", foreignField: "customer_id", as: "history" } },
      { $project: { _id: 0, name: { $concat: ["$first_name", " ", "$last_name"] },
      total_spent: { $sum: "$history.total_amount" } } } ]);
    conn.close();

    res.status(200).json(customers);

  } catch (error) {
    console.error("Fetch customers error:", error);
    res.status(500).json({
      error: "Server error while fetching customers",
    });
  }
}
```

4. Does that product have too low quantity



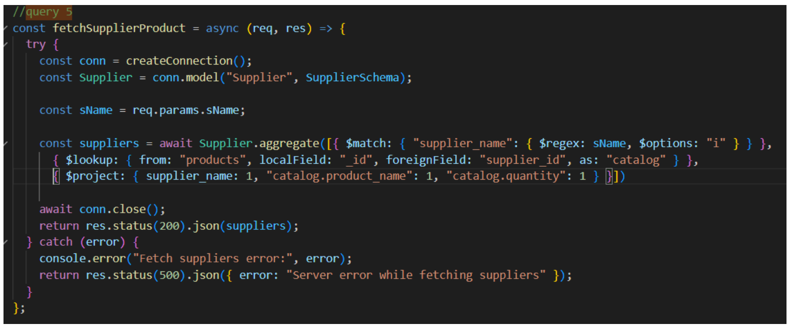
```javascript
//query 4
const fetchIsLowQuantity = async (req, res) => {
  try {
    const conn = createConnection();
    const Product = conn.model("Product", ProductSchema);

    const amount = Number(req.params.amount);
    const pName = req.params.name;

    const products = await Product.find({
      quantity: { $lte: amount },
      product_name: { $regex: new RegExp(pName, "i") },
    });

    await conn.close();
    return res.status(200).json(products);
  } catch (error) {
    console.error("Fetch products error:", error);
    return res.status(500).json({ error: "Server error while fetching products" });
  }
};
```

5. Supplier detail



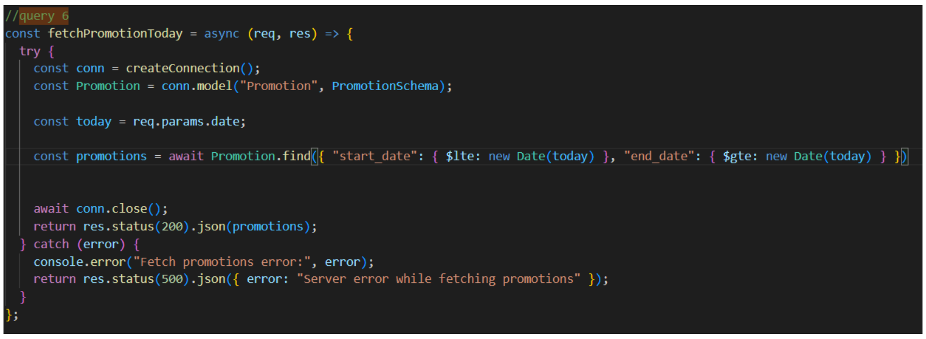
```javascript
//query 5
const fetchSupplierProduct = async (req, res) => {
  try {
    const conn = createConnection();
    const Supplier = conn.model("Supplier", SupplierSchema);

    const sName = req.params.sName;

    const suppliers = await Supplier.aggregate([{ $match: { "supplier_name": { $regex: sName, $options: "i" } } },
      { $lookup: { from: "products", localField: "_id", foreignField: "supplier_id", as: "catalog" } },
      { $project: { supplier_name: 1, "catalog.product_name": 1, "catalog.quantity": 1 } }])

    await conn.close();
    return res.status(200).json(suppliers);
  } catch (error) {
    console.error("Fetch suppliers error:", error);
    return res.status(500).json({ error: "Server error while fetching suppliers" });
  }
};
```

6. Is there promotion on that day



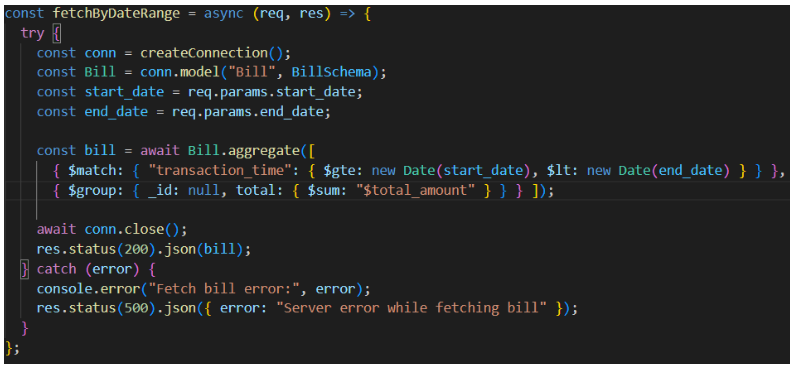
```javascript
//query 6
const fetchPromotionToday = async (req, res) => {
  try {
    const conn = createConnection();
    const Promotion = conn.model("Promotion", PromotionSchema);

    const today = req.params.date;

    const promotions = await Promotion.find({ "start_date": { $lte: new Date(today) }, "end_date": { $gte: new Date(today) } })


    await conn.close();
    return res.status(200).json(promotions);
  } catch (error) {
    console.error("Fetch promotions error:", error);
    return res.status(500).json({ error: "Server error while fetching promotions" });
  }
};
```

7. Cash flow in a duration

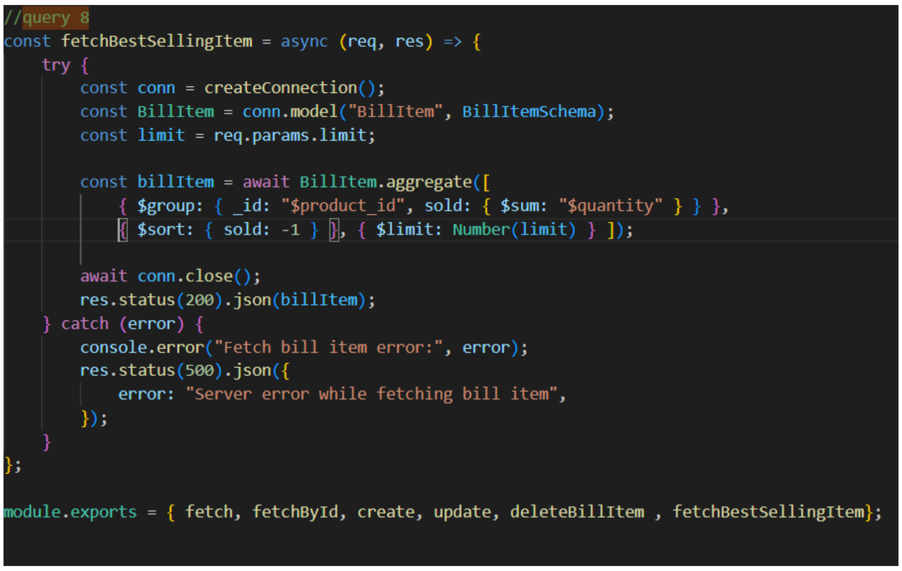


```
const fetchByDateRange = async (req, res) => {
  try {
    const conn = createConnection();
    const Bill = conn.model("Bill", BillSchema);
    const start_date = req.params.start_date;
    const end_date = req.params.end_date;

    const bill = await Bill.aggregate([
      { $match: { "transaction_time": { $gte: new Date(start_date), $lt: new Date(end_date) } } },
      { $group: { _id: null, total: { $sum: "$total_amount" } } } ]);

    await conn.close();
    res.status(200).json(bill);
  } catch (error) {
    console.error("Fetch bill error:", error);
    res.status(500).json({ error: "Server error while fetching bill" });
  }
};
```

8. Best selling item



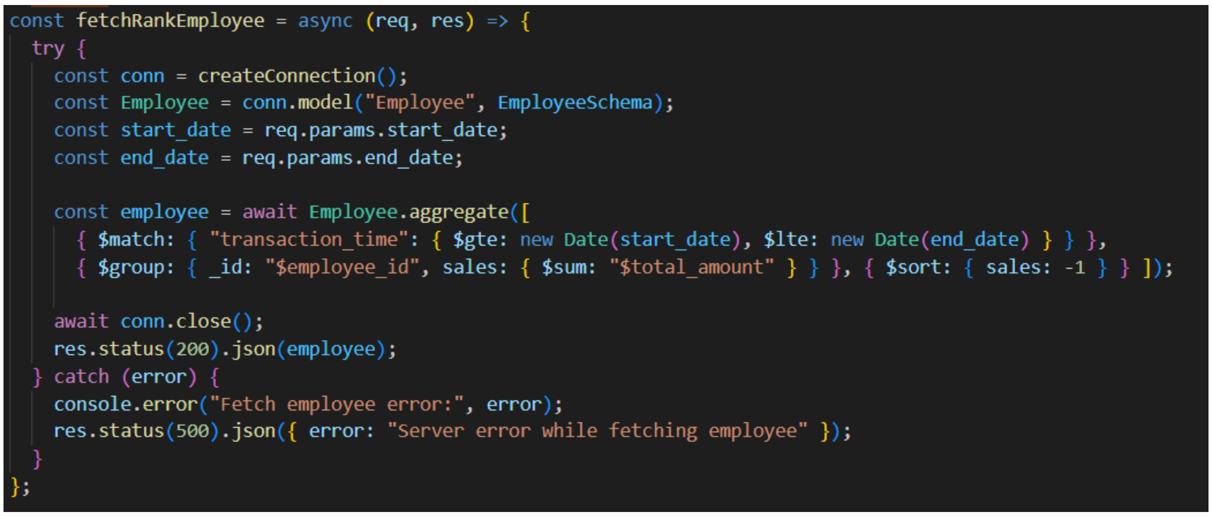
```
//query 8
const fetchBestSellingItem = async (req, res) => {
    try {
        const conn = createConnection();
        const BillItem = conn.model("BillItem", BillItemSchema);
        const limit = req.params.limit;

        const billItem = await BillItem.aggregate([
            { $group: { _id: "$product_id", sold: { $sum: "$quantity" } } },
            { $sort: { sold: -1 } }, { $limit: Number(limit) } ]);

        await conn.close();
        res.status(200).json(billItem);
    } catch (error) {
        console.error("Fetch bill item error:", error);
        res.status(500).json({
            error: "Server error while fetching bill item",
        });
    }
};

module.exports = { fetch, fetchById, create, update, deleteBillItem , fetchBestSellingItem};
```

9. Top working employee



```
const fetchRankEmployee = async (req, res) => {
  try {
    const conn = createConnection();
    const Employee = conn.model("Employee", EmployeeSchema);
    const start_date = req.params.start_date;
    const end_date = req.params.end_date;

    const employee = await Employee.aggregate([
      { $match: { "transaction_time": { $gte: new Date(start_date), $lte: new Date(end_date) } } },
      { $group: { _id: "$employee_id", sales: { $sum: "$total_amount" } } }, { $sort: { sales: -1 } } ]);

    await conn.close();
    res.status(200).json(employee);
  } catch (error) {
    console.error("Fetch employee error:", error);
    res.status(500).json({ error: "Server error while fetching employee" });
  }
};
```

Github link:
https://github.com/JirakornChaitanaporn/DB-project-mongo-supermarket.git